# Certificate Translation in Abstract Interpretation

Gilles Barthe and César Kunz

INRIA Sophia Antipolis-Méditerranée, France
{Gilles.Barthe,Cesar.Kunz}@inria.fr

**Abstract.** A certificate is a mathematical object that can be used to establish that a piece of mobile code satisfies some security policy. Since in general certificates cannot be generated automatically, there is an interest in developing methods to reuse certificates. This article formalises in the setting of abstract interpretation a method to transform certificates of program correctness along program transformations.

## 1 Introduction

A certificate $c$ is a mathematical object that can be checked automatically against some property $\phi$ it intends to prove; certificates arise naturally in logic, in the context of proof checking (via the Curry-Howard isomorphism) and of result checking. Certificates are also used to carry evidence of innocuousness of components in mobile code: in a typical Proof Carrying Code (PCC) scenario [11], a piece of mobile code is downloaded together with a certificate that shows its adherence to the consumer policy. While certificate checking is reasonably understood, certificate generation remains a challenging problem: while it is possible to generate certificates automatically for properties that are enforceable by automated program analyses, and in particular type systems, certificate generation remains necessarily interactive in the general case. It is therefore of interest to develop methods that simplify the construction of certificates.

In this paper, we use the setting of abstract interpretation [8, 9] to describe a method for transforming certificates along program transformations. We provide sufficient conditions for transforming a certificate of a program $G$ into a certificate of a program $G'$, where $G'$ is derived from $G$ by a semantically justified program transformation, typically a program optimization. These results provide substantial leverage on our earlier work on certificate translation [3].

*Certificate Translation* The primary goal of certificate translation is to extend the scope of PCC to complex policies, by supporting the generation of certificates from interactive source code verification. The scenario is of interest in situations where the functional correctness of the downloaded code is essential, and where certificate issues such as size or checking time are not relevant, e.g. in wholesale PCC, where one code verifier checks the certificate prior to distributing a cryptographically signed version to code consumers.

Certificate translation is tightly bound to the compilation infrastructure: for compilers that do not perform any optimization, proof obligations are preserved

(up to syntactic equality), and hence it is possible to reuse directly certificates of source code programs for their compilation; see e.g. [4].

In contrast, program optimizations make certificate translation more challenging. In [3], we show in a simplified setting that one can define certificate transformers for common program optimizations, provided one can infer automatically certificates of correctness for the underlying program analyses, by means of certifying analyzers. The existence of certifying analyzers and certificate translators is shown individually for each optimization.

*Comparison with our previous work.* The lack of a framework in which to formulate the basic concepts of certificate translation was a clear limitation of our earlier work, and made it difficult to assess the generality of certificate translation. The present article overcomes this limitation: we capture the essence of certificate translation in an algebraic setting that abstracts away from the specifics of programming languages, program transformations, and of verification methods. In fact, our results provide a means to generate, for given verification settings and program transformations, a set of proof obligations that guarantee the existence of certificate translators. The results of [3, 4] can then be recovered by discharging these proof obligations.

## 2 Certified Solutions

This section extends the basic framework of abstract interpretation with certificate infrastructures, in order to introduce formally the notion of certified solution. Definition 7 provides a general definition of certified solution that is of independent interest from certificate transformation, and provides a unifying framework for existing *ad hoc* definitions, see Section 5. For the purpose of this article, one can think about certified solutions as:

- programs annotated with logical assertions, and bundled with a certificate of the correctness of the verification conditions, or;
- programs annotated with abstract values (or types), and bundled with a certificate that the program is correct with respect to the interpretation of the abstract values.

We view programs as flow graphs. Thus, programs are directed pointed graphs with a distinguished set of output nodes, from which execution may not flow.

**Definition 1 (Programs).** *A program is a pointed directed graph $G = \langle \mathcal{N}, \mathcal{E}, l_{\mathsf{sp}} \rangle$, where $\mathcal{N}$ is a set of nodes, $l_{\mathsf{sp}} \in \mathcal{N}$ is a distinguished initial node, and $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a finitely branching relation; elements of $\mathcal{E}$ are called edges. We let $\mathcal{O}$ be the set of nodes without successors.*

Throughout this section, we let $G = \langle \mathcal{N}, \mathcal{E}, l_{\mathsf{sp}} \rangle$ be a program.

The semantics of programs is specified as a transition relation between states. Although more general definitions could be used, we choose to model states as pairs consisting of a program point and of an environment.

```
c := 1
x′ := x
y′ := y
while (y′ ≠ 1) do
  if (y′ mod 2 = 1) then
    c := c × x′
  fi
done
x′ = x′ × c
```

Fig. 1: Fast exponentiation algorithm.

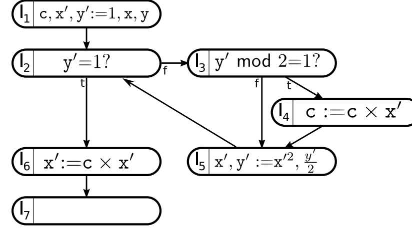


Fig. 2: Graph representation.

**Definition 2 (States, semantics).** *Let* $\mathsf{Env}$ *be an abstract set of environments. The set of states is defined as* $\mathsf{State} = \mathcal{N} \times \mathsf{Env}$. *The semantics of program* $G$ *is given by an abstract relation* $\rightsquigarrow \subseteq \mathsf{State} \times \mathsf{State}$.

*Example.* Consider as a running example (Fig. 1) a fast exponentiation algorithm. Its representation as a (labeled) graph is given in Figure 2; labels are either assignments of the form $x{:=}e$, in which case the node has exactly one successor, or conditional statements of the form $b?$, in which case the node has exactly two successor nodes, respectively corresponding to the true and false branch of the condition.

Both the analysis and verification frameworks are viewed as abstract interpretations. Note that, in contrast to standard abstract interpretation, our domains are pre-orders, rather than partial orders[1].

**Definition 3 (Abstract interpretation).** *Let* $G = \langle \mathcal{N}, \mathcal{E}, l_{\mathsf{sp}} \rangle$ *be a program. An abstract interpretation of* $G$ *is a triple* $I = \langle A, \{T_e\}_{e\in\mathcal{E}}, f \rangle$, *where*

- *$A$ is a pre-lattice*[2] $\langle D_A, \sqsubseteq_A, \sqsupseteq_A, \sqcup_A, \sqcap_A, \top_A, \bot_A \rangle$ *of abstract states. By abuse of notation, we write $A$ instead of $D_A$;*
- *$f$ is the flow sense, either forward ($f = \downarrow$), or backward ($f = \uparrow$);*
- $\{T_e\}_{e\in\mathcal{E}} : A \rightarrow A$ *is a family of monotone transfer functions.*

Thus, an abstraction of the program consists of an abstract domain, e.g. assertions or types, and a set of transfer functions, e.g. weakest precondition transformers.

[1] A binary relation $\sqsubseteq$ on a set $A$ is a pre-order if it is reflexive and transitive. A pre-order is a partial order if it is also antisymmetric. One natural domain for the verification infrastructure is that of propositions; we do not want to view it as a partial order since it would later imply (in Definition 6) that any formulas $\phi_1$ and $\phi_2$, if logically equivalent (i.e. if $\phi_1 \sqsubseteq \phi_2$ and $\phi_2 \sqsubseteq \phi_1$), by antisymmetry will have the same certificates (since $\phi_1 = \phi_2$), which is not desirable.

[2] Although it is sufficient to consider meet or join semi-lattices, depending on the flow of the interpretation, we find it more convenient to require our domains to be pre-lattices, since we deal both with forward and backwards analyses.

Furthermore, for every abstract domain $A$, we assume that $\models_A \subseteq \mathsf{Env} \times A$ is a satisfaction relation, s.t. $\sqsubseteq$ is an approximation order, i.e., that for all $\eta \in \mathsf{Env}$, $a_1, a_2 \in A$, if $\models_A \eta : a_1$ and $a_1 \sqsubseteq a_2$ then $\models_A \eta : a_2$. In the following, we simply write $\models$ omitting the subscript $A$.

**Definition 4 (Consistency).** *We say that $I$ is consistent with the semantics of $G$ w.r.t. $\models$ iff for all states $\langle l, \eta\rangle, \langle l', \eta'\rangle \in \mathsf{State}$ such that $\langle l, \eta\rangle \rightsquigarrow \langle l', \eta'\rangle$, and for all $a \in A$:*

- *if $f = \downarrow$ and $\models \eta : a$, then $\models \eta' : T_e(a)$;*
- *if $f = \uparrow$ and $\models \eta : T_e(a)$, then $\models \eta' : a$.*

A common means to verify program properties is to consider (pre- or post-) fixpoints of the transfer functions.

**Definition 5 (Solution).** *A labeling $S : \mathcal{N} \to A$ is a solution of $I$ if*

- *$f = \uparrow$ and for every $l$ in $\mathcal{N}$, $S(l) \sqsubseteq \bigsqcap_{\langle l,l'\rangle \in \mathcal{E}} T_{\langle l,l'\rangle}(S(l'))$;*
- *$f = \downarrow$ and for every node $l$ in $\mathcal{N}$, $S(l) \sqsupseteq \bigsqcup_{\langle l',l\rangle \in \mathcal{E}} T_{\langle l',l\rangle}(S(l'))$.*

**Lemma 1.** *Let $S$ be a solution of the abstract interpretation $I = \langle A, \{T_e\}, f\rangle$ and assume $I$ consistent with the semantics of $G$. Then, if $\langle l, \eta\rangle \rightsquigarrow^{\star} \langle l', \eta'\rangle$ and $\models \eta : S(l)$ then $\models \eta' : S(l')$.*

In order to capture the notion of certified solution at an appropriate level of abstraction, we rely on a general notion of certificate infrastructure.

**Definition 6 (Certificate infrastructure).** *A certificate infrastructure for $G$ consists of an abstract interpretation $I = \langle A, \{T_e\}_{e\in\mathcal{E}}, f\rangle$ for $G$, and a proof algebra $\mathcal{P}$ that assigns to every $a, a' \in A$ a set of certificates $\mathcal{P}(\vdash a \sqsubseteq a')$ s.t.:*

- *$\mathcal{P}$ is closed under the operations of Figure 3, where $a, b, c \in A$;*
- *$\mathcal{P}$ is sound, i.e. for every $a, a' \in A$, if $a \not\sqsubseteq a'$, then $\mathcal{P}(\vdash a \sqsubseteq a') = \emptyset$.*

*In the sequel, we write $c :\vdash a \sqsubseteq a'$ or $c :\vdash a' \sqsupseteq a$ instead of $c \in \mathcal{P}(\vdash a \sqsubseteq a')$.*

In the context of standard proof carrying code, the underlying pre-lattice is that of logical assertions, with logical implication $\Rightarrow$ as pre-order, and the transfer functions are the predicate transformers (based on weakest precondition or strongest postcondition) induced by instructions at any given program point. The particular form of certificates is irrelevant for this paper. It may nevertheless be helpful for the reader to think about certificates in terms of the Curry-Howard isomorphism and consider that $\mathcal{P}$ is given by the typing judgment in a dependently typed $\lambda$-calculus, i.e. $\mathcal{P}(\phi) = \{e \in \mathcal{E} \mid \langle\rangle \vdash e : \phi\}$, where $\mathcal{E}$ is the set of expressions of the type theory. Under such assumptions, one can provide an obvious type-theoretical interpretation to the functions of Figure 3; for example, $\mathsf{intro}_{\sqcap}$ is given by the $\lambda$-term $\lambda f.\ \lambda g.\ \lambda a.\ \langle fa, ga\rangle$.

In the sequel, we let $I = \langle A, \{T_e\}, f\rangle$ be a certificate infrastructure for $G$.

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

Fig. 3: Proof Algebra

**Definition 7 (Certified solution).** *A certified solution for $I$ is a pair $\langle S, \mathbf{c} \rangle$, where $S : \mathcal{N} \to A$ is a labeling and $\mathbf{c} = (c_l)_{l \in \mathcal{N}}$ is a family of certificates s.t. for every $l \in \mathcal{N}$,*

- *if $f = \uparrow$ then $c_l : \vdash S(l) \sqsubseteq \bigsqcap_{\langle l,l' \rangle \in \mathcal{E}} T_{\langle l,l' \rangle}(S(l'))$;*
- *if $f = \downarrow$ then $c_l : \vdash \bigsqcup_{\langle l',l \rangle \in \mathcal{E}} T_{\langle l',l \rangle}(S(l')) \sqsubseteq S(l)$.*

*It follows that $S$ is a solution for $I$.*

Many techniques, including lightweight bytecode verification and abstraction carrying code, 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 remaining of this section relates the construction of a (certified) solution from a partial labeling.

**Definition 8 (Labeling).** *A partial labeling is a partial function $S : \mathcal{N} \rightharpoonup A$ s.t. entry and output nodes are annotated, i.e. $\mathcal{O} \cup \{l_{\mathsf{sp}}\} \subseteq \mathsf{dom}(S)$, and such that the program is sufficiently annotated, i.e. the restriction $G_{\mathcal{N} \setminus \mathsf{dom}(S)}$ of $G$ to nodes that are not annotated is acyclic. A labeling $S$ is total if $\mathsf{dom}(S) = \mathcal{N}$.*

In a partial labeling $\mathsf{annot}$, annotations on entry and output nodes serve as specification, whereas we need sufficient annotations to reconstruct a total labeling $\overline{\mathsf{annot}}$ from the partial one.

**Definition 9 (Annotation propagation, verification condition).** *Let $\mathsf{annot}$ be a partial labeling. The labeling $\overline{\mathsf{annot}}$ is defined by the clause:*

- *if $f = \uparrow$, $\overline{\mathsf{annot}}(l) = \begin{cases} \mathsf{annot}(l) & \text{if } l \in \mathsf{dom}(\mathsf{annot}) \\ \bigsqcap_{\langle l,l' \rangle \in \mathcal{E}} T_{\langle l,l' \rangle}(\overline{\mathsf{annot}}(l')) & \text{otherwise} \end{cases}$*
- *if $f = \downarrow$, $\overline{\mathsf{annot}}(l) = \begin{cases} \mathsf{annot}(l) & \text{if } l \in \mathsf{dom}(\mathsf{annot}) \\ \bigsqcup_{\langle l',l \rangle \in \mathcal{E}} T_{\langle l',l \rangle}(\overline{\mathsf{annot}}(l')) & \text{otherwise} \end{cases}$*

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

- *$\mathsf{vc}(l) := \mathsf{annot}(l) \sqsubseteq \bigsqcap_{\langle l,l' \rangle \in \mathcal{E}} T_{\langle l,l' \rangle}(\overline{\mathsf{annot}}(l'))$ if $f = \uparrow$;*
- *$\mathsf{vc}(l) := \bigsqcup_{\langle l',l \rangle \in \mathcal{E}} T_{\langle l',l \rangle}(\overline{\mathsf{annot}}(l')) \sqsubseteq \mathsf{annot}(l)$ if $f = \downarrow$.*

Given a partial labeling $\mathsf{annot}$, one can build a certificate for $\overline{\mathsf{annot}}$ from certificates for the verification conditions on $\mathsf{dom}(\mathsf{annot})$.

**Lemma 2.** *Let* $\mathsf{annot}$ *be a partial labeling for* $I$ *and assume given* $c_l :\vdash \mathsf{vc}(l)$ *for every* $l \in \mathsf{dom}(\mathsf{annot})$*. Then there exists* $\boldsymbol{c}'$ *s.t.* $\langle\overline{\mathsf{annot}}, \boldsymbol{c}'\rangle$ *is a certified solution.*

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

**Corollary 1.** *Let* $\langle\mathsf{annot}, \boldsymbol{c}\rangle$ *be a certified partial labeling of* $\langle I, \mathcal{P}\rangle$ *and assume* $I$ *consistent with the semantics of* $G$*. Then, if* $\langle l_{\mathsf{sp}}, \eta\rangle \rightsquigarrow^* \langle l_o, \eta'\rangle$ *with* $l_o \in \mathcal{O}$ *and* $\models \eta : \mathsf{annot}(l_{\mathsf{sp}})$ *then* $\models \eta' : \mathsf{annot}(l_o)$*.*

*Example.* The verification infrastructure to certify the running example is built from a weakest precondition calculus over first-order formulae. That is, the backward transfer functions are defined, for any assertion $\phi$, as $T_{\langle l,l'\rangle}(\phi) = \phi[^e/_x]$ in case the node $l$ contains the assignment $x{:=}e$, and as $b \Rightarrow \phi$ or $\neg b \Rightarrow \phi$ respectively for the positive and negative branch of a jump statement conditioned by the boolean expression $b$. We assume given a certificate of functional correctness for the program, i.e. we assume given a certified solution $\langle\mathsf{annot}, \boldsymbol{c}\rangle$ of $I = \langle A, \{T_e\}, \uparrow\rangle$, where $\mathsf{annot}$ (as shown in Figure 5) is the partial labeling s.t. the precondition is trivial, i.e. $\mathsf{annot}(l_1) = \mathsf{true}$, the invariant is $\mathsf{annot}(l_2) = c \times x'^{y'} = x^y$ and the postcondition is $\mathsf{annot}(l_7) = x'{=}x^y$.

## 3 Certifying Analyzers

The certificate transformations studied in the next section require that the analyzers upon which the program transformation is based are certifying, i.e. produce certificates which justify their results. In this section, we thus provide sufficient conditions under which every solution may be certified. Proposition 1 below generalizes a previous result of Chaieb [7], who only considered the case where $f = \uparrow$ and $f^\sharp = \downarrow$.

Let $G$ be a program, $I^\sharp = \langle A^\sharp, \{T_e^\sharp\}, f^\sharp\rangle$ be an abstract interpretation, $I = \langle A, \{T_e\}, f\rangle$ a certificate infrastructure of program $G$, and $\gamma : A^\sharp \to A$ a concretization function.

**Proposition 1 (Existence of certifying analyzers).** *For every solution* $S$ *of* $I^\sharp$*, one can compute* $\boldsymbol{c}$ *s.t.* $\langle\gamma \circ S, \boldsymbol{c}\rangle$ *is a certified solution for* $I$*, provided there exist:*

- *for every* $a, a' \in A^\sharp$ *s.t.* $a \sqsubseteq^\sharp a'$*, a certificate* $\mathsf{monot}_\gamma(a, a') :\vdash \gamma(a) \sqsubseteq \gamma(a')$*;*
- *for every* $x \in A^\sharp$*, a certificate* $\mathsf{cons}(x) :\vdash \phi(x)$*, where* $\phi(x)$ *is defined in Figure 4 according to the flows of the interpretations.*

*Proof. For space reasons, we only show how to construct a certificate for the analysis in case* $f = f^\sharp = \downarrow$*. Let* $\mathsf{hyp}$ *stand for* $T^\sharp_{\langle l',l\rangle}(S(l')) \sqsubseteq S(l)$ *in*

$$p_1{:=}\mathsf{monot}_\gamma(\mathsf{hyp}) :\vdash \gamma(T^\sharp_{\langle l',l\rangle}(S(l'))) \sqsubseteq \gamma(S(l))$$
$$p_2{:=}\mathsf{cons}(S(l')) :\vdash T_{\langle l',l\rangle}(\gamma(S(l'))) \sqsubseteq \gamma(T^\sharp_{\langle l',l\rangle}(S(l')))$$
$$p_3{:=}\mathsf{weak}_\sqcap(-, p_1) :\vdash \gamma(T^\sharp_{\langle l',l\rangle}(S(l'))) \sqcap T_{\langle l',l\rangle}(\gamma(S(l'))) \sqsubseteq \gamma(S(l))$$
$$p_4{:=}\mathsf{elim}_\sqcap(p_3, p_2) :\vdash T_{\langle l',l\rangle}(\gamma(S(l'))) \sqsubseteq \gamma(S(l))$$
$$\boldsymbol{c}_l{:=}\mathsf{intro}_\sqcup(\{p_4\}_{\langle l',l\rangle\in\mathcal{E}}) :\vdash \bigsqcup\nolimits_{\langle l',l\rangle\in\mathcal{E}} T_{\langle l',l\rangle}(\gamma(S(l'))) \sqsubseteq \gamma(S(l))$$

While Proposition 1 provides a means to construct certifying analyzers, it is sometimes of interest to rely on more direct methods to generate certificates: in [3], we show how to construct compact certificates for constant propagation and common sub-expression elimination in an intermediate language.

| | |
|---|---|
| $f = f^\sharp =\downarrow$ | $T_e(\gamma(x)) \sqsubseteq \gamma(T_e^\sharp(x))$ |
| $f = f^\sharp =\uparrow$ | $T_e(\gamma(x)) \sqsupseteq \gamma(T_e^\sharp(x))$ |
| $f =\uparrow,\ f^\sharp =\downarrow$ | $T_e(\gamma(T_e^\sharp(x))) \sqsupseteq \gamma(x)$ |
| $f =\downarrow,\ f^\sharp =\uparrow$ | $T_e(\gamma(T_e^\sharp(x))) \sqsubseteq \gamma(x)$ |

Fig. 4: Definition of $\phi(x)$

## 4 Certificate Translation

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

In a first instance, Section 4.1 generalizes program transformations by allowing $G'$ to contain additional nodes that arise from duplicating fragments of $G$, as is the case for transformations such as loop unrolling. In a second instance, certificate transformation as defined in Section 4.2 requires that the transformed program $G'$ is a subgraph of the original program $G$. This is the case, for example, when $G'$ is derived from $G$ by applying optimizations such as constant propagation or common sub-expression elimination. In a third instance, in Section 4.3, we provide a notion of program skeleton, which abstracts away some of the structure of the program, to deal with transformations that do not preserve so tightly the structure of programs, such as code motion. Finally, in Section 4.4 we generalize certificate translation, covering optimizations such as dead variable elimination.

Throughout this section, we assume given two programs: an initial program $G = \langle \mathcal{N}, \mathcal{E}, l_{\mathsf{sp}} \rangle$ and a transformed program $G' = \langle \mathcal{N}', \mathcal{E}', l_{\mathsf{sp}} \rangle$. Furthermore, we assume given the required infrastructure to certify these programs; more concretely, consider the two abstract interpretations $I = \langle A, \{T_e\}_{e \in \mathcal{E}}, f \rangle$ and $I' = \langle A, \{T'_e\}_{e \in \mathcal{E}'}, f \rangle$ over $G$ and $G'$, and a proof algebra $\mathcal{P}$ over $A$. Note that the abstract interpretations share the same underlying domain and flow sense.

### 4.1 Code Duplication

In this section, we consider the case where some subgraphs of the initial program are duplicated in the transformed program, with the aim to trigger further program optimizations. Typical cases of code duplication are loop unrolling and function inlining.

**Definition 10 (Node replication).** *A program $G^+ = \langle \mathcal{N} \cup \mathcal{N}^+, \mathcal{E}^+, l_{\mathsf{sp}} \rangle$ is a result of replicating nodes of program $G = \langle \mathcal{N}, \mathcal{E}, l_{\mathsf{sp}} \rangle$ if $\mathcal{N}^+ \subseteq \{l^+ \mid l \in \mathcal{N}\}$ and $\mathcal{E} = \{\langle l_1, l_2 \rangle \mid \langle l, l' \rangle \in \mathcal{E}^+ \wedge \langle l, l' \rangle \in \{l_1, l_1^+\} \times \{l_2, l_2^+\}\}$.*

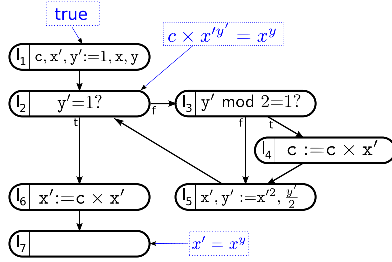


Fig. 5: Annotated program

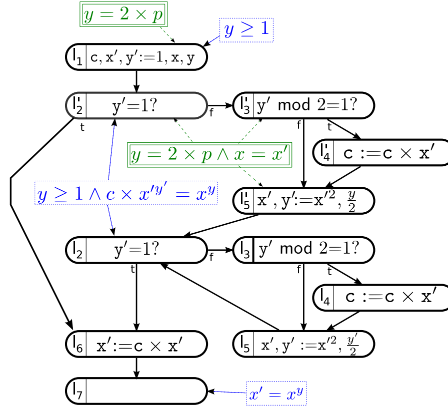


Fig. 6: Program after loop unrolling

Let $\langle I, \mathcal{P} \rangle$ be a certificate infrastructure with $I = \langle A, \{T_e\}_{e \in \mathcal{E}}, f \rangle$. Then, we define an extended certificate infrastructure $I^+ = \langle A, \{T_e\}_{e \in \mathcal{E}^+}, f \rangle$ for program $G^+$, the transfer functions $T_e$ for $e \in \mathcal{E}^+ \setminus \mathcal{E}$ being such that for all $\langle \bar{l}_1, \bar{l}_2 \rangle \in \mathcal{E}^+$, with $\bar{l}_i \in \{l_i, l_i^+\}$, $T_{\langle l_1, l_2 \rangle} = T_{\langle \bar{l}_1, \bar{l}_2 \rangle}$.

**Proposition 2.** *Assume the certificates of Fig. 7 exist for every $a_1, a_2, b_1, b_2 \in A$. Then every certified solution $\langle S, \boldsymbol{c} \rangle$ for $G$ can be transformed into a certified solution $\langle S^+, \boldsymbol{c}' \rangle$ for $G^+$, s.t. $S^+(l^+) = S(l)$ for all $l \in \mathsf{dom}(S)$.*

*Example.* Figure 6 shows the result of applying loop unrolling. Formally, it consists in duplicating a subset of nodes as defined in Section 4.1. In the graph, nodes $l_2$, $l_3$, $l_4$ and $l_5$ are respectively duplicated into the nodes $l'_2$, $l'_3$, $l'_4$, $l'_5$ and a new subset of edges is defined accordingly. A certified labeling $\langle \mathsf{annot}^+, \boldsymbol{c}^+ \rangle$, where $\mathsf{annot}^+(l'_2) = \mathsf{annot}(l_2)$, is generated for the program in Figure 6, by application of Proposition 2.

### 4.2 Subgraph Transformation

In this section, we assume that $G'$ is a subgraph of $G$, i.e. $\mathcal{N}' \subseteq \mathcal{N}$ and $\mathcal{E}' \subseteq \mathcal{E}$. Furthermore, we assume given an abstract interpretation $I = \langle A, \{T_e\}_{e \in \mathcal{E}}, f \rangle$ of $G$ and a labelling $S$ that justifies the transformation from $G$ to $G'$.

**Proposition 3 (Existence of certificate translators).** *Let $\langle S, \boldsymbol{c}^S \rangle$ be a certified solution for $I$ such that for every $\langle l_1, l_2 \rangle \in \mathcal{E}'$ and $a \in A$:*

- *if $f = \uparrow$ then $\mathsf{justif}(l_1, l_2) : \vdash S(l_1) \sqcap T_{\langle l_1, l_2 \rangle}(a) \sqsubseteq T'_{\langle l_1, l_2 \rangle}(a)$;*
- *if $f = \downarrow$ then $\mathsf{justif}(l_1, l_2) : \vdash T'_{\langle l_1, l_2 \rangle}(a) \sqsubseteq S(l_2) \sqcap T_{\langle l_1, l_2 \rangle}(a)$*

*Then, provided the certificates in Fig. 7 are given for every $a_1, a_2, b_1, b_2 \in A$, one can transform every certified labeling $\langle \mathsf{annot}, \boldsymbol{c} \rangle$ for $G$ into a certified labeling $\langle \mathsf{annot}', \boldsymbol{c}' \rangle$ for $G'$, where $\mathsf{annot}'(l) = \mathsf{annot}(l) \sqcap S(l)$ for every node $l$ in $\mathsf{dom}(\mathsf{annot}') = \mathsf{dom}(\mathsf{annot}) \cap \mathcal{N}'$.*

$$
\begin{array}{l}
\mathsf{monot}_T : \mathcal{P}(\vdash a_1 \sqsubseteq a_2) \to \mathcal{P}(\vdash T(a_1) \sqsubseteq T(a_2)) \\
\mathsf{distr}^{\leftarrow}_{(T,\sqcap)} :\vdash T(a_1) \sqcap T(a_2) \sqsubseteq T(a_1 \sqcap a_2) \\
\mathsf{distr}^{\rightarrow}_{(T,\sqcap)} :\vdash T(a_1 \sqcap a_2) \sqsubseteq T(a_1) \sqcap T(a_2) \\
\mathsf{assoc}^{\leftarrow}_{\sqcap} : \mathcal{P}(\vdash a_1 \sqcap (b_1 \sqcap b_2) \sqsubseteq (a_1 \sqcap b_1) \sqcap b_2) \\
\mathsf{assoc}^{\rightarrow}_{\sqcap} : \mathcal{P}(\vdash (a_1 \sqcap b_1) \sqcap b_2 \sqsubseteq a_1 \sqcap (b_1 \sqcap b_2)) \\
\mathsf{commut}_{\sqcap} : \mathcal{P}(\vdash a_1 \sqcap a_2 \sqsubseteq a_2 \sqcap a_1)
\end{array}
$$

Fig. 7: Requirements for certificate translation.

Using the results of Proposition 1, Proposition 3 can be instantiated to prove the existence of certificate transformers for many common optimizations, including constant propagation and common sub-expression elimination. In a nutshell, one first runs the certifying analyzer, which provides the solution $S$, then performs the optimization, and finally one provides a justification $\mathsf{justif}(l_1, l_2)$ for each edge (instruction) that has been modified by the optimization. This process is further illustrated in the following example.

*Example.* Suppose that we know (e.g. from the execution context) that the program is called with an even $y$; such knowledge is formalized by a precondition $y = 2 \times p$. Then, one can consider a forward abstract interpretation that analyses parity of variables and which variables are modified. A certifying analyzer for such an abstract interpretation exists by Proposition 1 and will produce a certified solution $\langle S, \boldsymbol{c}^S \rangle$ such that $S$ (shown inside double squared boxes in Fig. 6) associates the assertion $y = 2 \times p$ to the node $l_1$, the assertion $y' = 2 \times p \wedge x = x'$ to the nodes $\{l'_2, l'_3, l'_5\}$ and $\mathsf{true}$ to any other node.

Figure 8 contains an optimized version of the program of Figure 6, where jump statements whose conditions can be determined statically have been eliminated (nodes $l'_2$ and $l'_3$) and unreachable nodes have been removed (node $l'_4$), and where assignments have been simplified by propagating the results of the analysis (node $l'_5$). By Proposition 3, one can build a certificate for the optimized program, with labeling $\mathsf{annot}'(l) = \mathsf{annot}(l) \sqcap S(l)$ for all nodes $l \in \mathsf{dom}(\mathsf{annot})$ (in squared boxes in the figure), provided there exists, for every $a \in A$ and for every modified edge, i.e. for every $\langle l, l' \rangle \in \{\langle l'_2, l'_3 \rangle, \langle l'_3, l'_5 \rangle, \langle l'_5, l_2 \rangle\}$, a certificate:

$$
\mathsf{justif}_{\langle l, l' \rangle} :\vdash y' = 2 \times p \wedge x = x' \sqcap T_{\langle l, l' \rangle}(a) \sqsubseteq T'_{\langle l, l' \rangle}(a)
$$

The remaining certificates $\mathsf{justif}(l, l')$ for $\langle l, l' \rangle \notin \{\langle l'_2, l'_3 \rangle, \langle l'_3, l'_5 \rangle, \langle l'_5, l_2 \rangle\}$ are trivially generated since $T'_{\langle l, l' \rangle} = T_{\langle l, l' \rangle}$.

We conclude this section with a proof sketch of the existence of certificate transformers in the case of a backward certificate infrastructure. The idea is to build for every $l$ in $\mathcal{N}'$ the certificate

$$
\mathsf{goal}(l) :\vdash S(l) \sqcap \overline{\mathsf{annot}}(l) \sqsubseteq \overline{\mathsf{annot}}'(l)
$$

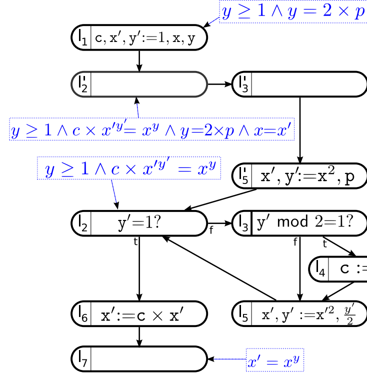


Fig. 8: Program after optimizing transformations

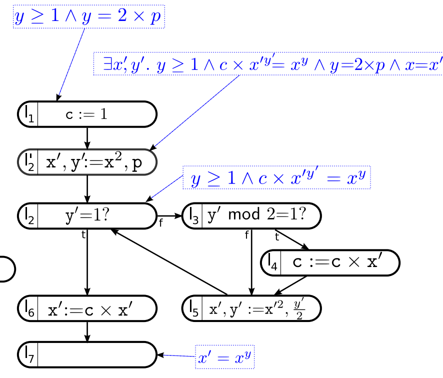


Fig. 9: Node coalescing and dead assignment elimination

from which the existence of a certificate for $\mathsf{annot}'$ follows. We proceed by induction, using the principle derived from the fact that $\mathsf{annot}$ 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, i.e. 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 Figure 10, where the application of certificates $\mathsf{assoc}_{\sqcap}^{\leftarrow}$, $\mathsf{assoc}_{\sqcap}^{\rightarrow}$ and $\mathsf{commut}_{\sqcap}$ is omitted for readability.

### 4.3 Program Skeletons

Proposition 3 requires that the transformation is justified for each edge of the program; this rules out several well known optimizations such as instruction swapping or code motion, whose justification involves more than one instruction. To overcome this limitation, one can abandon the intuitive representation of programs, where each edge represents one instruction, and cluster several instructions into a single edge. The purpose of this section is to capture formally this idea of clustering, and use it to strengthen our basic result.

Throughout this section, we assume that $\mathcal{N}_0 \subseteq \mathcal{N}$ is a set of nodes such that $G_{|\mathcal{N} \setminus \mathcal{N}_0}$ and $G'_{|\mathcal{N}' \setminus \mathcal{N}_0}$ are acyclic. We define $\mathcal{E}_0 = \mathcal{E}^\star \cap \mathcal{N}_0 \times \mathcal{N}_0$ where $\mathcal{E}^\star$ denote the transitive closure of $\mathcal{E}$. Let $\langle I, \mathcal{P} \rangle$ be a certificate infrastructure with $I = \langle A, \{T_e\}, f \rangle$. The transfer functions $\hat{T}$ are defined for every $\langle l, l' \rangle \in \mathcal{E}^0$ and $a \in A$ as $\check{T}_{\langle l, l' \rangle}(a)$, where $\check{T}_e$ is defined for every $e \in \mathcal{E}$ as:

- if $f = \uparrow$, $\begin{cases} \check{T}_{\langle l,l' \rangle} = T_{\langle l,l' \rangle} & \langle l, l' \rangle \in \mathcal{E} \\ \check{T}_{\langle l,l' \rangle}(a) = \bigsqcap_{\{\langle l,l'' \rangle \in \mathcal{E} \mid \mathsf{reaches}(l'',l')\}} T_{\langle l,l'' \rangle}(\check{T}_{\langle l'',l' \rangle}(a)) & \langle l, l' \rangle \notin \mathcal{E} \end{cases}$
- if $f = \downarrow$, $\begin{cases} \check{T}_{\langle l',l \rangle} = T_{\langle l',l \rangle} & \langle l', l \rangle \in \mathcal{E} \\ \check{T}_{\langle l',l \rangle}(a) = \bigsqcup_{\{\langle l'',l \rangle \in \mathcal{E} \mid \mathsf{reaches}(l',l'')\}} T_{\langle l'',l \rangle}(\check{T}_{\langle l',l'' \rangle}(a)) & \langle l', l \rangle \notin \mathcal{E} \end{cases}$

Let $a = S(l)$, $a' = S(l')$, $T = T_{\langle l,l'\rangle}$ and $T' = T'_{\langle l,l'\rangle}$ in:

$$
\begin{array}{rl}
\mathsf{hyp}_1 := \mathsf{monot}_T & : \mathcal{P}(\vdash b_1 \sqsubseteq b_2) \rightarrow \mathcal{P}(\vdash T(b_1) \sqsubseteq T(b_2)) \\
\mathsf{hyp}_2 := \mathsf{distrib}_T & : \mathcal{P}(\vdash T(b_1) \sqcap T(b_2) \sqsubseteq T(b_1 \sqcap b_2)) \\
p_1 := \mathsf{goal}(l') & : \vdash a' \sqcap \overline{\mathsf{annot}}(l') \sqsubseteq \overline{\mathsf{annot}}'(l') \\
p_2 := \mathsf{hyp}_1(p_1) & : \vdash T'(a' \sqcap \overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l')) \\
p_3 := \mathsf{justif}(l, l') & : \vdash 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) & : \vdash a \sqcap T(a' \sqcap \overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l')) \\
p_6 := \mathsf{hyp}_2 & : \vdash T(a') \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq T(a' \sqcap \overline{\mathsf{annot}}(l')) \\
p_7 := \mathsf{axiom} & : \vdash a \sqsubseteq a \\
p_8 := \mathsf{weak}_\sqcap(p_6) & : \vdash a \sqcap T(a') \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq T(a' \sqcap \overline{\mathsf{annot}}(l')) \\
p_9 := \mathsf{intro}_\sqcap(p_8, \mathsf{weak}_\sqcap(p_6)) & : \vdash : \sqsubseteq a \sqcap T(a') \sqcap T(\overline{\mathsf{annot}}(l'))a \sqcap T(a' \sqcap \overline{\mathsf{annot}}(l')) \\
p_{10} := \mathsf{elim}_\sqcap(\mathsf{weak}_\sqcap(p_5), p_9) & : \vdash a \sqcap T(a') \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l')) \\
p_{11} := \boldsymbol{c}_l^S & : \vdash a \sqsubseteq T(a') \\
p_{12} := \mathsf{elim}_\sqcap(p_{10}, p_{11}) & : \vdash a \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l')) \\
p_{13} := \mathsf{weak}_\sqcap(p_{12}) & : \vdash a \sqcap \bigsqcap_{\langle l,l'\rangle \in \mathcal{E}} T(\overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l')) \\
\mathsf{goal}(l) := \mathsf{intro}_\sqcap(\{p_{12}\}_{\langle l,l'\rangle \in \mathcal{E}}) & : \vdash a \sqcap \bigsqcap_{\langle l,l'\rangle \in \mathcal{E}} T(\overline{\mathsf{annot}}(l')) \sqsubseteq \bigsqcap_{\langle l,l'\rangle \in \mathcal{E}} T'(\overline{\mathsf{annot}}'(l'))
\end{array}
$$

Fig. 10: Definition of $\mathsf{goal}(l)$ for certificate translation (case $f = \uparrow$)

where the condition $\mathsf{reaches}(l, l')$ stands for the existence of a sequence of labels $l_1, \ldots, l_k$ with $l_1 = l$ and $l_k = l'$ s.t. $\langle l_i, l_{i+1}\rangle \in \mathcal{E}$, for all $i \in \{1, \ldots, k-1\}$. The set $\mathcal{E}'_0$ and the transfer functions $\hat{T}'$ are defined in a similar fashion.

The results of the previous sections extend immediately to program skeletons.

**Lemma 3.** *Let $\langle S, \boldsymbol{c}_I\rangle$ be a certified solution for $I$ s.t $\mathsf{dom}(S) \subseteq \mathcal{N}_0$. Then $\langle \hat{S}, \hat{\boldsymbol{c}}_{\hat{I}}\rangle = \langle S, \boldsymbol{c}_{I|\mathcal{N}_0}\rangle$ is a certified solution of $\hat{I} = \langle A, \hat{T}_e, f\rangle$.*

**Proposition 4.** *Let $\langle \hat{S}, \hat{\boldsymbol{c}}_{\hat{I}}\rangle = \langle S, \boldsymbol{c}_{I|\mathcal{N}_0}\rangle$ be a certified solution of $\hat{I} = \langle A, \hat{T}_e, f\rangle$. Suppose that for every $\langle l_1, l_2\rangle \in \mathcal{E}'_0$ and $a \in A$:*

- *if $f = \uparrow$ then $\mathsf{justif}(l_1, l_2) : \vdash \hat{S}(l_1) \sqcap \hat{T}_{\langle l_1,l_2\rangle}(a) \sqsubseteq \hat{T}'_{\langle l_1,l_2\rangle}(a)$;*
- *if $f = \downarrow$ then $\mathsf{justif}(l_1, l_2) : \vdash \hat{T}'_{\langle l_1,l_2\rangle}(a) \sqsubseteq \hat{S}(l_2) \sqcap \hat{T}_{\langle l_1,l_2\rangle}(a)$*

*Then every certified labeling $\langle \mathsf{annot}, \boldsymbol{c}\rangle$ for $G$ such that $\mathsf{dom}(\mathsf{annot}) \subseteq \mathcal{N}_0$ can be transformed into a certified labeling $\langle \mathsf{annot}', \boldsymbol{c}'\rangle$ for $G'$, where $\mathsf{annot}'(l)$ is defined as $\mathsf{annot}(l) \sqcap S(l)$ for all $l \in \mathsf{dom}(\mathsf{annot}') = \mathsf{dom}(\mathsf{annot}) \cap \mathcal{N}'$.*

*Example.* A further simple transformation consists of coalescing the nodes $l'_2$, $l'_3$ and $l'_5$ to simplify the graph representation. Formally, we use the program skeletons to cluster the sub-graph constituted by the nodes $l'_2$, $l'_3$ and $l'_5$ into a single node $l'_2$. Then, we define the transfer function $\hat{T}_{\langle l'_2,l_2\rangle} = T'_{\langle l'_5,l_2\rangle}$ (formally, one should have $T_{\langle l'_2,l_2\rangle} = T'_{\langle l'_2,l'_3\rangle} \circ T'_{\langle l'_3,l'_5\rangle} \circ T'_{\langle l'_5,l_2\rangle}$ but $T'_{\langle l'_2,l'_3\rangle}$ and $T'_{\langle l'_3,l'_5\rangle}$ are the identity function). Hence, by a trivial application of Proposition 4, there exists a certified solution $\langle \hat{\mathsf{annot}}, \hat{\boldsymbol{c}}\rangle$, for the collapsed program representation $\langle \mathcal{N}_0, \mathcal{E}_0, l_{\mathsf{sp}}\rangle$, s.t. $\hat{\mathsf{annot}}(l) = \mathsf{annot}(l)$ for all $l \in \mathcal{N}_0$.

Proposition 4 can be used to prove preservation of proof obligations for non-optimizing compilers. Indeed, non-optimizing compilation transforms a graph representation of a program by splitting each node into a subgraph of more basic nodes, preserving the overall program structure. Thus, one can coalesce back the generated subgraphs into a skeleton structure similar to the source program. If we assume that transfer functions of the skeleton representation are equal to those of the source program (it is not sufficient that the functions are equivalent w.r.t. $\sqsubseteq$; equality is essential), then proof obligations are preserved and certificates can be reused without modification.

### 4.4 Second-Order Analysis-Based Optimizations

Proposition 3 does not cover optimizations that rely on analyses such as variable liveness to justify their result. This motivates the following mild generalization, in which the transformation is justified w.r.t. a composition operator.

**Proposition 5.** *Let* $\centerdot : A \times A \to A$ *be a composition operator s.t. for every* $a_1, a_2, b_1, b_2 \in A$ *there exists a certificate*

$$\mathsf{monot}_{\centerdot} : \mathcal{P}(\vdash a_1 \sqsubseteq a_2) \to \mathcal{P}(\vdash b_1 \sqsubseteq b_2) \to \mathcal{P}(\vdash a_1 \centerdot b_1 \sqsubseteq a_2 \centerdot b_2)$$

*Let* $\langle S, \boldsymbol{c}^S \rangle$ *be a certified solution for* $I$ *s.t. for every* $\langle l_1, l_2 \rangle \in \mathcal{E}'$ *and* $a \in A$*:*

- *if* $f = \uparrow$ *then* $\mathsf{justif}(l_1, l_2) : \vdash S(l_1) \centerdot T_{\langle l_1, l_2 \rangle}(a) \sqsubseteq T'_{\langle l_1, l_2 \rangle}(a \centerdot S(l_2))$*;*
- *if* $f = \downarrow$ *then* $\mathsf{justif}(l_1, l_2) : \vdash T'_{\langle l_1, l_2 \rangle}(a \centerdot S(l_1)) \sqsubseteq S(l_2) \centerdot T_{\langle l_1, l_2 \rangle}(a)$

*Then, provided the certificate* $\mathsf{monot}_T$ *defined in Fig. 7 exist for all* $a_1, a_2 \in A$*, every certified labeling* $\langle \mathsf{annot}, \boldsymbol{c} \rangle$ *for* $G$ *can be transformed into a certified labeling* $\langle \mathsf{annot}', \boldsymbol{c}' \rangle$ *for* $G'$*, where* $\mathsf{annot}'(l) = \mathsf{annot}(l) \centerdot S(l)$ *for every node* $l$ *in* $\mathsf{dom}(\mathsf{annot}') = \mathsf{dom}(\mathsf{annot}) \cap \mathcal{N}'$*.*

*Example.* Finally, we perform liveness analysis on program variables and remove assignments to dead variables. The resulting program is given in Figure 9. The remaining of this subsection is devoted to an explanation of the analysis, and to a justification of the transformation.

Assuming a standard program semantics, we say that a variable is live at a certain program point if its value will be needed in the future. An intensional definition classifies a variable $x$ as live at a program node $l$ if there is a path from $l$ that reaches an expression referring to $x$, without traversing an assignment to $x$. We prefer to use a more extensional interpretation of liveness, inspired by Benton's Relational Hoare Logic [5], identifying a declaration of a set of live variables as a relational proposition. To this end, 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. Formally, the extension consists on partitioning the domain of variables by attaching to each of them an index $_{\langle 1 \rangle}$

or $_{\langle 2\rangle}$. The set of transfer functions is also modified accordingly; for instance, the substitution $\phi[{}^{e}\!/_{x}]$ corresponding to the assignment $x{:=}e$ at node $l$, is replaced by 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 $_{\langle i\rangle}$.

Then, we define $\gamma(X) = \bigwedge_{v\in X} v_{\langle 1\rangle} = v_{\langle 2\rangle}$ as an interpretation of the fact that all variables in $X$ are live. In order to generate a certificate for the optimized program, we apply Proposition 5, using as composition operator over relational propositions the function $\centerdot$ defined as

$$\phi \centerdot \psi = \exists x^1, \ldots, x^k.\ \phi[{}^{x^1_{\langle 2\rangle}}\!/_{x}] \ldots [{}^{x^k_{\langle 2\rangle}}\!/_{x}] \wedge \psi[{}^{x^1_{\langle 1\rangle}}\!/_{x}] \ldots [{}^{x^k_{\langle 1\rangle}}\!/_{x}]$$

where $\{x^1, \ldots, x^k\}$ are the set of variables in $\phi$ or $\psi$. The interpretation of the composition operator is that if $X$ declares the set of live variables, then $\gamma(X) \centerdot \phi$ is the result of existentially quantifying away from $\phi$ the variables that are not live.

By Proposition 1, we know that a certified solution $\langle \gamma \circ \mathsf{live}, \boldsymbol{c}''\rangle$ exists s.t. $\mathsf{live}(l_1) = \{x, y\}$, $\mathsf{live}(l'_2) = \{x, y, c\}$ and $\mathsf{live}(l) = \{x, y, c, x', y'\}$ for $l \notin \{l_1, l'_2\}$. Since node $l_1$ contains an assignment to variables $x'$ and $y'$ and these variables are not live in node $l'_2$, we may safely simplify the statement by removing such assignments. From Proposition 5 we can transform the current certified solution by assuming the certificate

$$\mathsf{justif}(l_1, l'_2) : \vdash \gamma(\mathsf{live}(l_1)) \centerdot T_{\langle l_1, l'_2\rangle}(\phi) \sqsubseteq T'_{\langle l_1, l'_2\rangle}(\gamma(\mathsf{live}(l'_2)) \centerdot \phi)\ .$$

For readability, if $\phi$ is a non-relational proposition, $\gamma(X) \centerdot \phi$ is equivalently denoted $\exists y_1, \ldots, y_m.\ \phi$ where $\{y_1, \ldots, y_m\} = \mathsf{Var} - X$. Then, the goal of the certificate $\mathsf{justif}(l_1, l'_2)$ can be interpreted as $\vdash \phi[{}^{1}\!/_{c}][{}^{x}\!/_{x'}][{}^{y}\!/_{y'}] \sqsubseteq (\exists x', y'.\ \phi)[{}^{1}\!/_{c}]$.

## 5 Related Work

*Certified solutions.* Abstraction Carrying Code (ACC) is an instance of PCC where programs come with a solution in an abstract interpretation that can be used to specify the consumer policy [1]. ACC is closely related to our notion of certified solution; in fact, one may view the latter as a natural extension of ACC to settings where the pre-order relation is either undecidable, or expensive to compute, and where the use of certificates is required in order to check solutions. Besson *et al* [6] have recently developed a program analysis framework in which certificates are used to verify inclusions between elements of the abstract domain of polyhedra. Their analysis is also an instance of a certified solution. Rival [12, 13] proposed a method to translate the result of a static analysis along program compilation. Result validation is restricted to post-fixpoint checking, i.e. there is no notion of certificate.

*Certifying analyzers.* We are aware of two previous works on certifying, or proof-producing, program analyses. Both consider the backwards case. Seo, Yang and

Yi [15] 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 [7] 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 technique was applied in the context of a certified PCC infrastructure [16].

*Certificate translation.* Müller and co-workers [2, 10] define a proof transforming compiler for sequential Java. They consider Hoare logics for source and bytecode programs, and transform a correct derivation for a Java program into a correct derivation for the JVM program obtained by non-optimizing compilation.

Saabas and Uustalu [14] develop type-based methods to establish the existence of certifying analyzers and certificate transformers. They illustrate the feasibility of their method by explaining in detail two particular transformations: common subexpression elimination and dead variable elimination. They demonstrate the correctness of both transformations, by derivability of Hoare logic proofs, and provide an algorithm to transform a Hoare proof of the original program to a Hoare proof of the transformed program.

## 6 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. In order to further demonstrate the benefits of our framework, we show that certificate translation scales to concurrent languages, to relational program logics, which have been used to prove information flow properties, and that similar techniques can be used to justify hybrid certificates, that combine simultaneously several verification methods.

*Acknowledgments* We are grateful to David Pichardie, Tamara Rezk and the referees for their constructive comments. This work is partially supported by the EU project MOBIUS.

## References

1. E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-carrying code. In F. and A. Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning*, volume 3452 of *Lecture Notes in Computer Science*, pages 380–397. Springer-Verlag, 2005.
2. F. Y. Bannwart and P. Müller. A program logic for bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 255–273. Elsevier, 2005.

3. G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In K. Yi, editor, *Static Analysis Symposium*, number 4134 in Lecture Notes in Computer Science, Seoul, Korea, August 2006. Springer-Verlag.
4. G. Barthe, T.Rezk, and A. Saabas. Proof obligations preserving compilation. In T. Dimitrakos, F. Martinelli, P. Ryan, and S. Schneider, editors, *Proceedings of FAST'05*, volume 3866 of *Lecture Notes in Computer Science*, pages 112–126. Springer-Verlag, 2005.
5. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In N. D. Jones and X. Leroy, editors, *Principles of Programming Languages*, pages 14–25. ACM Press, 2004.
6. F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Result certification for relational program analysis. Technical report, IRISA, 2007.
7. A. Chaieb. Proof-producing program analysis. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *ICTAC*, volume 4281 of *Lecture Notes in Computer Science*, pages 287–301. Springer, 2006.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.
9. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages*, pages 269–282, 1979.
10. P. Müller and M. Nordio. Proof-transforming compilation of programs with abrupt termination. Technical Report 565, ETH Zurich, 2007.
11. G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.
12. X. Rival. Abstract Interpretation-Based Certification of Assembly Code. In L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Verification, Model Checking and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 41–55, 2003.
13. X. Rival. Symbolic Transfer Functions-based Approaches to Certified Compilation. In *Principles of Programming Languages*, pages 1–13. ACM Press, 2004.
14. A. Saabas and T. Uustalu. Type systems for optimizing stack-based code. In M. Huisman and F. Spoto, editors, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 190(1) of *Electronic Notes in Theoretical Computer Science*, pages 103–119. Elsevier, 2007.
15. S. Seo, H. Yang, and K. Yi. Automatic Construction of Hoare Proofs from Abstract Interpretation Results. In A. Ohori, editor, *Asian Programming Languages and Systems Symposium*, volume 2895 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2003.
16. M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2005.