

# Validation of Interprocedural Optimizations

Amir Pnueli, Anna Zaks<sup>1,2</sup>

*Computer Science Department  
New York University  
New York, USA*

---

## Abstract

Translation validation is an approach of ensuring compilation correctness in which each compiler run is followed by a validation pass that proves that the target code produced by the compiler is a correct translation of the source code. We present a framework for translation validation of compiler optimization run that targets *reactive procedural* programs. Our algorithm is automatic, requires minor compiler cooperation, and accommodates most classical *interprocedural* optimizations such as global constant propagation, inlining, tail-recursion elimination, interprocedural dead code elimination, dead argument elimination, and cloning.

*Keywords:* compiler verification, translation validation, interprocedural optimizations, program equivalence, deductive verification, formal methods

---

## 1 Introduction

Compilers are quite large applications, which are bound to have bugs. At present, the GCC Bug Database contains 1370 unresolved bugs. Clearly, it is highly desirable to ensure that the transformations performed by a compiler preserve the semantics of a program. Translation validation [13] is an approach of ensuring compilation correctness in which each compiler run is followed by a validation pass that proves that the target code produced by the compiler is a correct translation of the source code. The methodology of compiler verification can be categorized by its intended customers. Compiler writers are interested in methods that lead to creation of a self-certifying compiler and may assume full knowledge of the inner workings of a particular compiler. The full control over the compiler allows for creation of powerful and efficient translation validation techniques; for example, [17], [5] describe methods targeted at specific compiler optimizations and phases. Another group interested in compiler verification are users who may need to work with an existing compiler and require tools that insist on minimal compiler cooperation. Good examples of such tools are presented in [11], [19,20], and [16]. They rely

---

<sup>1</sup> Email: [amir@cs.nyu.edu](mailto:amir@cs.nyu.edu), [ganna@cs.nyu.edu](mailto:ganna@cs.nyu.edu)

<sup>2</sup> This research has been supported in part by a grant from the Microsoft Phoenix Academic Program and the NSF CSR-EHS grant CNS-0720581.

on heuristics and available compiler annotations, usually the debug information, to detect the transformations that took place. The frameworks are also well suited for development of a self-certifying compiler contributing to the first direction as well.

Translation validation algorithms can be generally viewed as a special case of program equivalence checking. Furthermore, since the input programs are infinite state systems, deductive techniques are applied. Specifically, [19] is a generalization of the Floyd method [7] in which a set of assertions is associated with the locations of the source and target programs. Next, a set of verification conditions is generated. The validity of the verification conditions implies that the assertions hold and that the target is a correct translation of the source. To the best of our knowledge, the existing translation validation approaches are not capable and were not designed to deal with interprocedural optimizations. For example, in [11] two executions are considered the same if both lead to the same sequence of function calls and returns. [19] does not model programs with procedures and only considers deterministic programs with no intermediate inputs and outputs.

The main contribution of this paper is a novel translation validation algorithm capable of checking correctness of a compiler optimization run in presence of interprocedural optimizations. Specifically, our framework is an extension of [19] to programs with procedures. In contrast to [19], which used transition systems as the formal model, we rely on *transition graphs*, which capture not only conditions and assignments but also procedure calls and I/O operations and allows to verify optimizing phases of reactive programs with non-terminating runs. The notion of correct translation has also been generalized – the target program  $\mathcal{T}$  is a *correct translation* (refinement) of program  $\mathcal{S}$  if every observation of  $\mathcal{T}$  is also an observation of  $\mathcal{S}$ . An observation of a program is similar to a program computation. However, it only captures the essential information, for example, the values of the variables used in I/O instructions. Finally, the paper presents methodology for generation of auxiliary invariants used for verification of context sensitive copy propagation and presents *Interprocedural Translation Validation algorithm* that, in addition to the transformations covered by [19], is strong enough to handle most, if not all, of the interprocedural optimizations described in literature [10,4] and performed by optimizing compilers (GCC, ORC, LLVM [2]), like global constant propagation, inlining, tail-recursion elimination, interprocedural dead code elimination, dead argument elimination, and cloning. The main restriction of [19] and, consequently, of the extended approach the assumption that there exists a mapping from the loops(cut-points) of  $\mathcal{T}$  to the loops of  $\mathcal{S}$ . However, most of the classical compiler optimizations such as constant folding, induction variable optimizations, branch optimizations, common subexpression elimination as well as the mentioned above interprocedural optimizations preserve this property. Rules for loop reordering transformations presented in [20] can be additionally applied to verify transformations such as loop interchange, fusion, and tiling.

The paper is organized as following. Section 2 presents our formal model, the notion of correct translation, and inductive assertion networks. Section 3 describes the Translation Validation algorithm. Section 4 and Section 5 present the generation of the auxiliary invariants and the translation verification conditions, respectively. We give a comprehensive example in Section 6 and conclude in Section 7.

## 2 Preliminaries

### 2.1 Formal Model of Transition Graphs

Our model is similar to that presented in [12] for verification of procedural programs. A program (application)  $\mathcal{A}$  consists of  $m + 1$  procedures:  $main, f_1, \dots, f_m$ , where  $main$  represents the main procedure, and  $f_1, \dots, f_m$  are procedures which may be called from  $main$  or from other procedures. We use  $f_i(\vec{x}, \&\vec{z})$  to denote the signature of a procedure. Here, call-by-value parameter passing method is used for  $\vec{x}$ , and call-by-reference is used for  $\vec{z}$ . A procedure may return a result by means of  $\vec{z}$  variables. We use  $\vec{y}$  to denote the typed variables of a procedure.  $\vec{y} = (\vec{x}; \vec{z}; \vec{w})$ , i.e. the variables in  $\vec{y}$  are partitioned into  $\vec{x}, \vec{z}$ , and  $\vec{w}$ , where  $\vec{x}$  and  $\vec{z}$  are the *input* parameters and  $\vec{w}$  denotes the *local* variables of the procedure.

Each procedure is presented as a **transition graph**  $f_i := (\vec{y}, \mathcal{N}_i, \mathcal{E}_i)$  with variables  $\vec{y}$ , nodes (locations)  $\mathcal{N}_i = \{r^i = n_0^i, n_1^i, n_2^i, \dots, n_k^i = t^i\}$  and a set of labeled edges  $\mathcal{E}_i$ . It must have a distinct root node  $r^i$  as its only entry point, a distinct tail node  $t^i$  as its only exit point, and every other node must be on a path from  $r^i$  to  $t^i$ . Nodes of the graph are connected by directed edges labeled by instructions. There are four types of instructions: guarded assignments, procedure calls, reads, and writes. Consider a procedure  $f_i(\vec{x}; \&\vec{z})$  with  $\vec{y} = (\vec{x}, \vec{z}, \vec{w})$ . Let  $\vec{u}$  include variables from  $\vec{y}$ ; and  $E(\vec{y})$  be a list of expressions over  $\vec{y}$ .

- A *guarded assignment* is an instruction of the form  $c \rightarrow [\vec{u} := E(\vec{y})]$ , where guard  $c$  is a boolean expression. When the assignment part is empty, we abbreviate the label to a pure condition  $c?$ .
- *Procedure call* instruction  $g(E(\vec{y}), \vec{u})$  denotes a call to procedure  $g(\vec{x}_g; \&\vec{z}_g)$ , passing input parameters  $E(\vec{y})$  by value and  $\vec{u}$  by reference.
- *Read* and *write* instructions are denoted by  $read(\vec{u})$  and  $write(\vec{u})$ . They are used to express the interaction of the procedure with the outside world; e.g. I/O.

Consider a pair of nodes  $(i, j)$  connected by either procedure call, read, or write edge. With no loss of generality, we assume that this edge is the only edge connecting  $i$  and  $j$ . Note that deterministic and non-deterministic branching can be expressed through the use of the guarded assignment instruction.

Transition graphs can be used to model programs written in procedural languages. In order to construct a formal model of a program, we first choose a set of program **cut-points**  $C$  to be a set of program locations such that:

- At least one location in each loop belongs to  $C$ .
- For every procedure, both procedure entry and exit belong to  $C$ .
- The locations before and after read, write, and procedure call belong to  $C$ .

Each procedure (or function) whose implementation is given is represented by a transition graph. We choose the set  $C$  of a procedure  $f_i$  to be the set of nodes for the corresponding transition graph. For every pair of locations  $n, m$  in  $C$ , if there exists a path  $\pi$  from  $n$  to  $m$ , which does not pass through any other location from  $C$ , we add edge  $(n, m)$  to the graph and label it by the instruction that summarizes the effect of executing the path  $\pi$ . Each call to a procedure whose implementation is

hidden can be modeled by read/write instructions. If a hidden procedure is stateless and does not perform I/O operations (for example, *pow* function in C), the call is modeled by uninterpreted functions. Note that global variables and functions also can be modeled in this framework.

For example, the procedure *MAIN* depicted on Fig. 1 reads in a natural number  $A$  and writes out the expression  $3 * A! + 5$ . It calls a recursive procedure *FAC* to compute the factorial. *FAC* takes argument  $N$  by value and  $R$  by reference and computes  $R * N!$ , which is returned to the caller by reference.

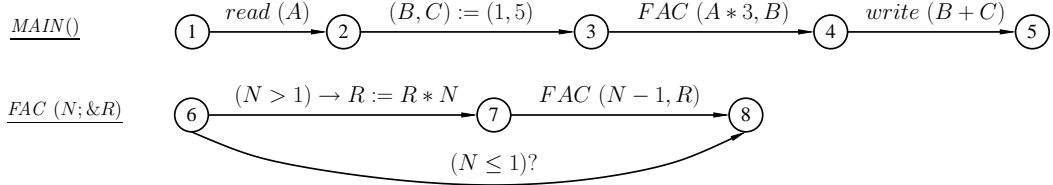


Fig. 1. Transition graphs for the program that on input  $A$ , outputs  $3 * A! + 5$ .

## 2.2 States and Computations

We denote by  $\vec{d} = (d^{\vec{x}}; d^{\vec{z}}; d^{\vec{w}})$  a tuple of values, which represents an interpretation (i.e., an assignment of values) of the procedure variables  $\vec{y} = (\vec{x}; \vec{z}; \vec{w})$ . A **state** of a procedure  $f$  is a pair  $\langle n; \vec{d} \rangle$  consisting of a node  $n$  and a data interpretation  $\vec{d}$ . A  $(\vec{\xi}, \vec{\zeta})$ -**computation** of procedure  $f$  is a maximal sequence of states and labeled transitions:

$$\sigma : \langle r; (\vec{\xi}, \vec{\zeta}, \vec{\top}) \rangle \xrightarrow{\lambda_1} \langle n_1; \vec{d}_1 \rangle \xrightarrow{\lambda_2} \langle n_2; \vec{d}_2 \rangle \dots$$

The tuple  $\vec{\top}$  denotes uninitialized values. At the first state of the computation, the location is  $r$ , the entry location of  $f$ ; the values of input variables  $\vec{x}$  and  $\vec{z}$  are set to  $\vec{\xi}$  and  $\vec{\zeta}$ , respectively, and the local variables  $\vec{w}$  are not initialized. Labels of the transitions are either labels of edges in the program or the special label *ret*. Each transition must be justified by either an intra-procedural transition, a call transition, or a return transition such that the call and return transitions are *balanced*. See our technical report [18] for the formal definition. We define a set of computations of program  $\mathcal{A}$  to be the set of computations of *main*.

## 2.3 Correct Translation

The notion of correct translation used in this work is similar to those used in [14], [8] and is based on the general notion of refinement between source(abstract) program  $\mathcal{S}$  and target(concrete) program  $\mathcal{T}$ . We define the correctness of translation via equivalence of program behaviors that can be observed by the user. Intuitively, given the same input both, the source program  $\mathcal{S}$  and the target program  $\mathcal{T}$ , must produce the same set of outputs.

Given a computation, we define  $V_s$  – the set of **observable variables** at a state  $s = \langle n, d \rangle$ , to be the minimal set satisfying the following two conditions. First, if  $s$  is a state immediately after transition  $read(\vec{u})$ ,  $V_s \supseteq \vec{u}$ . Second, if  $s$  is a state immediately before transition  $write(\vec{u})$ ,  $V_s \supseteq \vec{u}$ . (Above, we use  $V_s \supseteq \vec{u}$  to denote  $V_s \supseteq \{v : \forall v \text{ in } \vec{u}\}$ .) We associate an **observation function**  $\mathcal{O}$  with each program, mapping the source and target states and transition labels into a common

domain. The observation function needs to ensure that read and write transitions of the source and target computations match. Formally, given a state  $s = \langle n, d \rangle$ , an observation function  $\mathcal{O}(s)$  is defined as following. Let  $V_s$  be the set of observable variables at  $s$ . If  $V_s = \emptyset$  then  $\mathcal{O}(s) = \perp$ , else  $\mathcal{O}(s) = \vec{d}_{V_s}$ . We obtain  $\vec{d}_{V_s}$  by restricting  $\vec{d}$  only to the values that correspond to the variables in  $V_s$ . Given a transition label  $\lambda$ , an observation function  $\mathcal{O}(\lambda)$  is defined as follows. If  $\lambda$  is a label of a transition that is a read or a write,  $\mathcal{O}(\lambda)$  is equal to *read* or *write*, respectively. Otherwise,  $\mathcal{O}(\lambda) = \perp$ .

An **observation** of a computation  $\sigma$ , denoted  $o(\sigma)$ , is obtained by applying the observation function  $\mathcal{O}$  to each state and each transition label in  $\sigma$ . That is, for  $\sigma : s_1 \xrightarrow{\lambda_1} s_2 \xrightarrow{\lambda_2} s_3 \dots$ , we get  $o(\sigma) : \mathcal{O}(s_1) \xrightarrow{\mathcal{O}(\lambda_1)} \mathcal{O}(s_2) \xrightarrow{\mathcal{O}(\lambda_2)} \mathcal{O}(s_3) \dots$ .

Computations  $\sigma$  and  $\sigma'$  are **stuttering equivalent**, denoted  $\sigma \sim_{st} \sigma'$ , if their observations  $o(\sigma)$ ,  $o(\sigma')$  only differ from each other by finite sequences of pairs  $\perp \xrightarrow{\perp}$  or  $\xrightarrow{\perp} \perp$ . Stuttering equivalence is used to ensure that even though the programs may have to execute a different number of instructions to get to an observable state, the difference is always finite. Our assumption is that the user is not time-sensitive so this finite delta cannot be observed. For example,  $\beta \sim_{st} \beta'$ :

$$\begin{aligned} o(\beta) : & \perp \xrightarrow{read} (5, 22) \xrightarrow{\perp} \perp \xrightarrow{\perp} \perp \xrightarrow{\perp} (110) \xrightarrow{write} \perp \\ o(\beta') : & \perp \xrightarrow{read} (5, 22) \xrightarrow{\perp} \perp \xrightarrow{\perp} (110) \xrightarrow{write} \perp \xrightarrow{\perp} \perp \end{aligned}$$

In both computations, first two numbers: 5 and 22, are read; and then, after a finite number of steps, their product: 110, is written out.

**Definition 1** We say that the target program  $\mathcal{T}$  is a **correct translation** of the source program  $\mathcal{S}$ , denoted  $\mathcal{T} \sqsubseteq \mathcal{S}$ , if for every target computation  $\sigma_{\mathcal{T}}$ , there exists a source computation  $\sigma_{\mathcal{S}}$  such that  $\sigma_{\mathcal{T}} \sim_{st} \sigma_{\mathcal{S}}$ . For deterministic programs  $\mathcal{S}$  and  $\mathcal{T}$ , in which the input sequence uniquely determines the computation,  $\mathcal{T} \sqsubseteq \mathcal{S}$  if and only if  $\mathcal{S} \sqsubseteq \mathcal{T}$ .

#### 2.4 Inductive Assertion Network

We introduce *virtual* variables  $\vec{X}$  and  $\vec{Z}$  to represent the values of the input variables  $\vec{x}$  and  $\vec{z}$  at the procedure entry and denote the extended vector of variables by  $\vec{Y} = (\vec{X}, \vec{Z}, \vec{x}, \vec{z}, \vec{w})$ . An **assertion network** associates an assertion  $\varphi_l$  with each program location  $l$ .

- For each procedure  $f$  with the entry location  $r$ , we denote  $\varphi_r$  by  $p_f$ . The **input predicate**  $p_f = p_f(\vec{X}, \vec{Z}; \vec{x}, \vec{z})$  imposes constraints only on the input variables of the procedure. Since we assume that the main procedure *main* does not have input parameters,  $p_0 = true$ .
- Similarly, we denote  $\varphi_t$ , the assertion associated with the exit location of  $f$ , by  $q_f$ . The **output predicate**  $q_f = q_f(\vec{X}, \vec{Z}; \vec{z})$  is the procedure summary: it specifies the relation between the input and output values.
- The assertions at all other locations  $\varphi_l(\vec{Y})$  may depend on any of the variables.

For each edge of the transition graph  $e$  connecting a node  $i$  to a node  $j$ , we form **verification conditions**, which represent different edge types:

- *Guarded Assignment*: If  $e$  is an assignment edge labeled by  $c \rightarrow [\vec{u} := E(\vec{y})]$ ,  

$$\mathcal{VC}_e : \varphi_i(\vec{Y}) \wedge c(\vec{y}) \rightarrow \varphi_j(\vec{Y})[\vec{u} \mapsto E(\vec{y})],$$
 where  $\varphi_j(\vec{Y})[\vec{u} \mapsto E(\vec{y})]$  is obtained from  $\varphi_j(\vec{Y})$  by replacing variables in  $\vec{u}$  by the corresponding expressions in  $E(\vec{y})$ .
- *Read*: If  $e$  is a read edge labeled by  $read(\vec{u})$ ,  

$$\mathcal{VC}_e : \varphi_i(\vec{Y}) \rightarrow \varphi_j(\vec{Y})[\vec{u} \mapsto \vec{u}'],$$
 where  $\vec{u}'$  is a vector of fresh variables. Intuitively, the assertion  $\varphi_j$  must hold for all possible inputs.
- *Write*: If  $e$  is a write edge labeled by  $write(\vec{u})$ ,  

$$\mathcal{VC}_e : \varphi_i(\vec{Y}) \rightarrow \varphi_j(\vec{Y}).$$
- *Procedure call*: We associate the following two conditions with a procedure call  $f(E(\vec{y}), \vec{u})$ , which calls the procedure with signature  $f(\vec{x}_f; \&\vec{z}_f)$ :

$$\mathcal{VC}_{call} : \varphi_i(\vec{Y}) \rightarrow p_f(E(\vec{y}), \vec{u}; E(\vec{y}), \vec{u})$$

$$\mathcal{VC}_{return} : \varphi_i(\vec{Y}) \wedge q_f(E(\vec{y}), \vec{u}; \vec{z}_f) \rightarrow \varphi_j(\vec{Y})[\vec{u} \mapsto \vec{z}_f]$$

Note that  $p_f$  and  $q_f$  are the input and output predicates of  $f$ . Thus,  $\mathcal{VC}_{call}$  checks that the assertion associated with the location before the call,  $\varphi_i$ , implies the input predicate of the callee.  $\mathcal{VC}_{return}$  checks that the assertion at the location reached immediately after the procedure return is implied by the output predicate and  $\varphi_i$ . The conditions generally use variables of the caller procedure with the only exception of the variables passed by reference  $\vec{z}_f$ . This exception allows to disregard the old information about the variables passed by reference, stored by  $\varphi_i(\vec{Y})$ , and instead rely on  $q_f$ .

An assertion network  $\mathcal{N} = \{\varphi_0, \dots, \varphi_n\}$  for a program  $\mathcal{A}$  is said to be **inductive** if all the verification conditions for all edges in  $\mathcal{A}$  are valid. Network  $\mathcal{N}$  is said to be **invariant** if for every execution state  $\langle l; \vec{d} \rangle$  occurring in a computation, the visiting data state  $d$  satisfies the corresponding assertion  $\varphi_l$  associated with  $l$ .

**Claim 1** *Every inductive network is invariant.*

### 3 Interprocedural Translation Validation Algorithm

The Interprocedural Translation Validation algorithm is an extension of the rule Validate [20] to reactive procedural programs. Given two procedural programs  $\mathcal{S}$  and  $\mathcal{T}$ , the algorithm generates a proof that the target program  $\mathcal{T}$  is a correct translation of the source program  $\mathcal{S}$ . Let  $C^{\mathcal{T}}$  and  $C^{\mathcal{S}}$  denote the sets of nodes (cut-points) of  $\mathcal{T}$  and  $C^{\mathcal{S}}$  respectively. We follow the five steps below to check if  $\mathcal{T} \sqsubseteq \mathcal{S}$ .

**Step 1:** Establish *control abstraction*  $\kappa : C^{\mathcal{T}} \rightarrow C^{\mathcal{S}}$ , mapping the target nodes to the source nodes, such that  $r$  is the initial location (root of the main procedure) of  $\mathcal{T}$  if and only if  $\kappa(r)$  is the initial location of  $\mathcal{S}$ . The mapping  $\kappa$  is total but can be neither surjective nor injective. For example, we allow a non-surjective mapping to handle a situation when a loop is eliminated as part of dead code elimination. Optimizations such as inlining result in a non-injective control abstraction. Note that the control abstraction not only specifies the mapping between the program locations but also imposes many-to-one correspondence between target and source

procedures. For example, consider target procedure  $g^T$  with the root node  $r$  and the tail node  $t$ .  $g^T$  corresponds to source procedure  $G^S$  with the root node  $\kappa(r)$  and the tail node  $\kappa(t)$ .

**Step 2:** Construct sets of target and source auxiliary assertions that form inductive networks  $\mathcal{N}^T = \{\varphi_0^T, \dots, \varphi_{|C^T|}^T\}$  and  $\mathcal{N}^S = \{\varphi_0^S, \dots, \varphi_{|C^S|}^S\}$  for programs  $\mathcal{T}$  and  $\mathcal{S}$ , respectively. Form verification conditions showing that the networks are invariant, following rules from Section 2.4. Add the generated conditions to the set of verification conditions  $\mathcal{VC}$ .

**Step 3:** Let  $V^S$  and  $V^T$  denote the sets of variables that belong to programs  $\mathcal{S}$  and  $\mathcal{T}$ , respectively. Form *data abstraction*  $\{\alpha_0, \dots, \alpha_{|C^T|}\}$  by defining each  $\alpha_l(V^S; V^T)$  as a conjunction of equalities of the form  $E(v^S) = E(v^T)$  at each target node  $l \in C^T$ . The data abstraction must be valid at the initial location of  $\mathcal{T}$ :  $\alpha_r = \text{true}$ . Intuitively, the data abstraction maps the values of target variables at location  $l$  to the values of source variables at location  $\kappa(l)$ .

**Step 4:** Form *Translation Verification Conditions*, presented in Section 5, for every edge of the target program and add them to the set of verification conditions  $\mathcal{VC}$ . If there exists an edge of the target program that does not contribute a verification condition, generate ERROR.

**Step 5:** Establish validity of the conditions in  $\mathcal{VC}$ ; generate ERROR otherwise. The ERROR signifies that either an error in translation is detected or we ran into a transformation that is not currently supported.

The methods for construction of the data and control abstractions are presented in [19]. These methods rely on compiler annotations that are usually required for debugging compiled code, so they are provided by most compilers. Construction of the data abstraction is based on refining the mapping between source and target program variables. To construct the control abstraction  $\kappa$ , we first generate the set of source cut-points  $C^S$  such that they satisfy the minimal requirements stated in Section 2.1. Then, we rely on the compiler annotations to assist in computation of the control abstraction  $\kappa$  and  $C^T$  by providing the mapping from the source program locations to the target program locations. Finally, we check the  $C^T$  for completeness with respect to the requirements of Section 2.1.

## 4 Generating the Source Inductive Assertion Network

We use the inductive assertion networks from [6] as the foundation. Generally, the method generates the assertions based on the set of reachable definitions (variable definitions that must hold at a particular location) and is applied in the intraprocedural setting. However, the source network has to be extended so that it incorporates the information essential for proving interprocedural optimizations. In this section, we show how to generate the source assertion network that is strong enough for context sensitive copy constant propagation. Linear constant propagation can be handled in a similar fashion. We are going to use [15] as our interprocedural dataflow analysis algorithm. The algorithm is *precise* and has an efficient representation for the internal data that we can use to our advantage.

As a first try, it appears that any precise solution to the interprocedural constant-propagation problem should suffice. For example,  $\varphi_l^S$  should be extended with

conjunct  $x = 17$  if  $x$  always evaluates to constant 17 at location  $l$ . However, the resulting network  $\mathcal{N}^S$  may not be inductive. Fortunately, the fixpoint based dataflow analysis algorithm not only provides a solution, but also finds a fixpoint for the corresponding set of dataflow equations. We are going to use the information about the fixpoint itself to strengthen our network so it would be inductive.

Let  $V$  be the finite set of program variables. Let  $L = Z_{\perp}^{\top}$  be the integer constant propagation lattice. We denote the meet operator by  $\sqcap$ . The set  $Env(V, L)$  of **environments** is the set of functions from  $V$  to  $L$ . A mapping  $T : Env(V, L) \mapsto Env(V, L)$  is called *an environment transformer*. A transformer  $T$  is **distributive** iff for every variable  $v \in V$ ,  $(T(\sqcap_i env_i))(v) = \sqcap_i (T(env_i))(v)$ . The algorithm in [15] essentially computes a transformer  $T_{(r_k, l)}$  between the root of each procedure  $P_k$  and every location of the procedure. Note that the transformer  $T_{(r_k, t_k)}$  between the root and the tail of  $P_k$  is essentially a procedure summary that is represented in our framework by the invariant  $q_k$ .

Since  $T$  needs to operate on functions with infinite domains, the following succinct representation for distributive transformers is used in [15]. Every distributive transformer  $T$  can be represented using a set of functions  $\Omega^T = \{\rho_{v, v'} \mid v, v' \in V \cup \{\Lambda\}\}$ , each of type  $L \mapsto L$ . Function  $\rho_{v, v'}$  captures the effect that the value of variable  $v$  in the argument environment has on the value of  $v'$  in the result environment; if  $v'$  does not depend on  $v$ , then  $\rho_{v, v'} = \lambda l. \top$ . Function  $\rho_{\Lambda, v'}$  is used to represent the effect on the variable  $v$  that is independent of the argument environment. For any symbol  $v'$ , the value  $T(env)(v')$  can be determined by taking the meet of the values of  $|V| + 1$  individual function applications:  $T(env)(v') = \rho_{\Lambda, v'} \sqcap (\sqcap_{v \in V} \rho_{v, v'}(env)(v))$ . Since we are only concerned with constant copy propagation, all the functions in  $\Omega^T$  will be either identities or constants.

**Example 1** Consider the example in Fig. 2. Below is the list of environment transformers computed by [15] for procedure  $f_{oo}$ . We omit all the functions that evaluate to top  $\rho_{(v, v')} = \lambda l. \top$ .

$$\Omega_{(2,2)} = \{ \rho_{x,x} = \lambda l.l, \rho_{c,c} = \lambda l.l, \rho_{y,y} = \lambda l.l, \rho_{z,z} = \lambda l.l \}$$

$$\Omega_{(2,3)} = \{ \rho_{c,c} = \lambda l.l, \rho_{y,y} = \lambda l.l, \rho_{y,z} = \lambda l.l \}$$

$$\Omega_{(2,4)} = \{ \rho_{c,c} = \lambda l.l, \rho_{c,z} = \lambda l.l, \rho_{y,z} = \lambda l.l \}$$

Given all the dataflow facts (constants) and the transformer represented by  $\Omega_{(i,j)}$ , we follow the following rules to compute an invariant  $\varphi_l$  at location  $l$  of  $P_k$ :

- We ignore all functions of the form  $\rho_{(v, v')} = \lambda l. \top$ .
- For each variable  $v'$  that is not set to  $\perp$  by  $\rho_{(\Lambda, v')} \in \Omega_{(r_k, l)}$  we add the following conjunct to  $\varphi_l$ :

$$\bigvee_{\rho_{v, v'} \in \Omega_{(r_k, l)}} v' = \rho_{v, v'}(V), \text{ where } V \text{ is the value of } v \text{ at the procedure entry.}$$

We use disjunction to model the effect of the meet operator. In our example, we use fictitious variables  $X, C, Y, Z$  to store the the initial values of  $x, c, y, z$ .

- We also add the conjunct  $x = const$  if  $x$  was determined to evaluate to constant  $const$  at location  $l$ . We need this addition since  $T_{(r_k, l)}$  does not propagate the information from the callers.

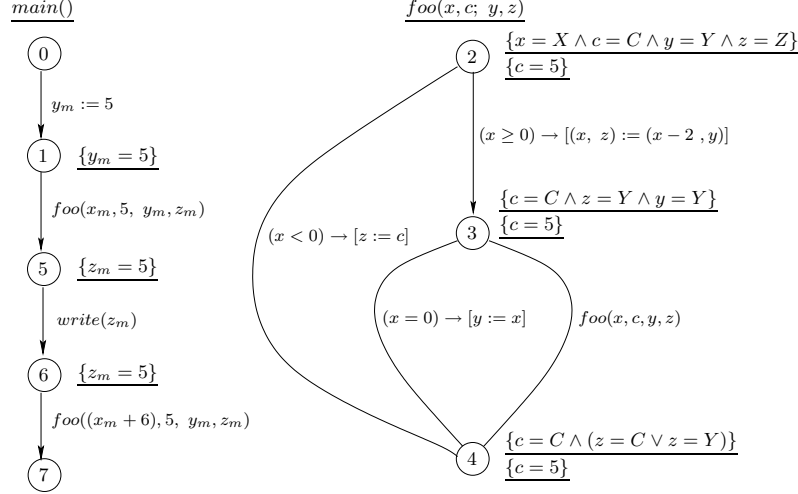


Fig. 2. Inductive network for Constant Copy Propagation.

The resulting invariants, denoted in Fig. 2 by curly brackets, form an inductive network. For example, let's show that the return verification condition for call edge (1, 5) of our example holds.

$$\begin{aligned} \mathcal{VC}_{ret}: \quad \varphi_1 \quad \wedge \quad \varphi_4[(C, Y) \mapsto (5, y_m)] \quad \rightarrow \quad \varphi_5[z_m \mapsto z] \quad \Leftrightarrow \\ y_m = 5 \wedge c = 5 \wedge (z = 5 \vee z = y_m) \wedge c = 5 \rightarrow z = 5 \end{aligned}$$

## 5 Translation Verification Conditions

Similarly to the verification conditions used to prove the assertion network inductive, Translation Verification Conditions prove that the data abstraction is inductive on the computations of the target program. They also ensure that source and target observations match given the consistent input. We first give a recipe for generating translation verification conditions when the structure of the transformed program is preserved: for every edge of the target program  $e^T$  connecting nodes  $i$  and  $j$ , there exist the corresponding source edges  $e^S$  between nodes  $\kappa(i)$  and  $\kappa(j)$ :

- *Guarded Assignment*: If the target edge  $e^T$  is a guarded assignment edge of  $\mathcal{T}$ ; and  $\kappa(i), \kappa(j)$  are also connected by one or more assignment edges in  $\mathcal{S}$ , we generate the following conditions.

$$\begin{aligned} \alpha_i \wedge \varphi_i^S \wedge \varphi_i^T \wedge \rho_{e^T} \rightarrow \left( \bigvee_{e^S \in \text{Edges}(\kappa(i), \kappa(j))} c_{e^S} \right). \\ \alpha_i \wedge \varphi_i^S \wedge \varphi_i^T \wedge \rho_{e^T} \wedge \left( \bigvee_{e^S \in \text{Edges}(\kappa(i), \kappa(j))} \rho_{e^S} \right) \rightarrow \alpha_j. \end{aligned}$$

In the formulas above, for an edge  $e \in \{e^S, e^T\}$  labeled by  $c \rightarrow [\vec{u} := E(\vec{y})]$ ,  $c_e$  stands for the condition  $c$  and  $\rho_e$  for the expression  $c \wedge (\vec{u}' = E(\vec{y})) \wedge \vec{v}' = \vec{v}$ , where  $\vec{v}$  are all variables of  $\alpha_j$  with the exception of those in  $\vec{u}$ . The first implication checks that whenever the target transition is enabled, at least one of the corresponding source transitions is also enabled. The second verification condition checks that the data abstraction is preserved by the matching target and source transitions. Invariants  $\varphi_i^S$  and  $\varphi_i^T$  are used to strengthen the left-hand-side of the implication.

- *Read*: If  $e^T$  and  $e^S$  are both labeled by read instructions  $read(\vec{u}^T)$  and  $read(\vec{u}^S)$ ,  
 $\alpha_i \wedge \varphi_i^T \wedge \varphi_{\kappa(i)}^S \wedge (\vec{u}^T = \vec{u}^S) \rightarrow \alpha_j$ .
- *Write*: If  $e^T$  and  $e^S$  are both write edges, labeled by  $write(E^T)$  and  $write(E^S)$ ,  
 $\alpha_i \wedge \varphi_i^T \wedge \varphi_{\kappa(i)}^S \rightarrow \alpha_j \wedge (E^T = E^S)$ .

Read and write verification conditions ensure that the data mapping implies matching source and target output given the consistent input.

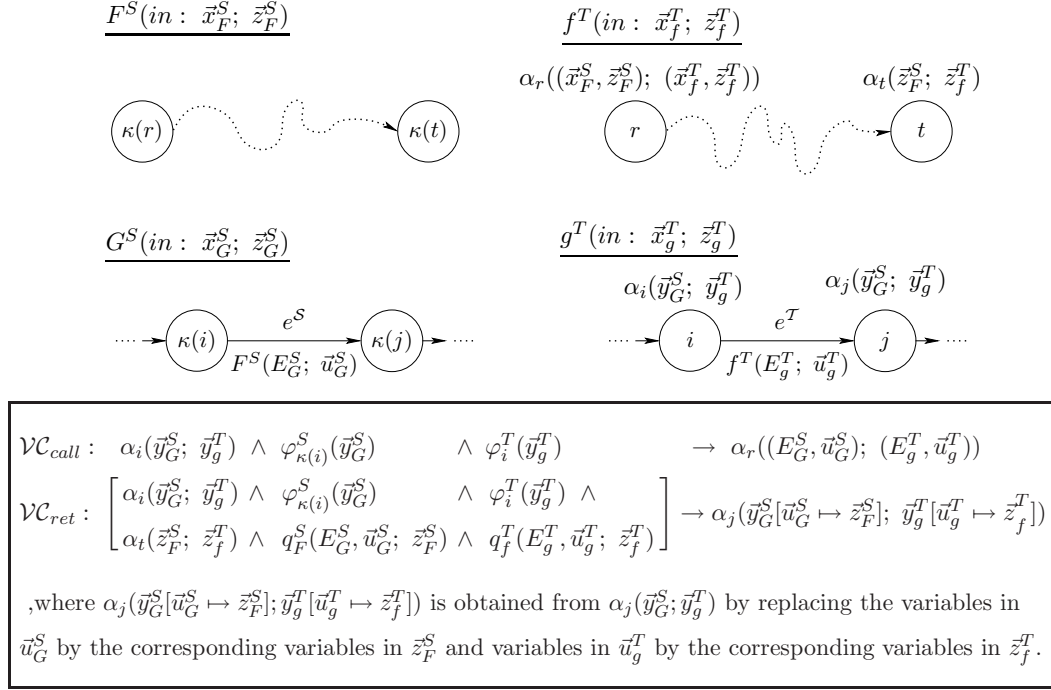


Fig. 3. CALL VERIFICATION CONDITIONS: procedure  $G^S$  calls procedure  $F^S$  in the source program and procedure  $g^T$  calls procedure  $f^T$  in the target program.

- *Procedure Call*: If both  $e^T$  and  $e^S$  are call edges labeled by  $f^T(E_g^T; \vec{u}_g^T)$  and  $F^S(E_G^S; \vec{u}_G^S)$ , respectively, where  $f^T$  is mapped to  $F^S$ , we generate Call Verification Conditions presented in Fig. 3, which check that the data abstraction is preserved by stepping through the procedure calls. Similarly to the call conditions of Section 2.4, the  $\mathcal{V}\mathcal{C}_{call}$  condition checks that the data mapping holds at the entry to the procedure; and the  $\mathcal{V}\mathcal{C}_{ret}$  condition guarantees that it holds after the procedure return. The right-hand-sides of the implications are strengthened by the auxiliary invariants of the source and target systems; recall that  $q_F^S$  and  $q_f^T$  are the output predicates of  $F^S$  and  $f^T$ , respectively. If procedure  $f^T$  is not mapped to procedure  $F^S$ , the algorithm raises the **Error**.

Inlining and Tail-Recursion Elimination(TRE) introduce situations in which the source code contains a call edge that corresponds to a subgraph in the target. In this case, we prove the translation by “stepping into” the procedure call on the source. Let  $e^T = (i, a)$  be an unconditional assignment edge of the target such that there exists a source call edge  $(\kappa(i), \kappa(j))$ , labeled by  $F^S(E_G^S; \vec{u}_G^S)$ ;  $\kappa(a)$  is the entry node of  $F^S$ ; and there exists the corresponding node  $b$  in the target such that  $\kappa(b)$  is the exit node of the procedure  $F^S$ . If  $(b, j)$  is an unconditional assignment of  $\mathcal{T}$ ,

proceed with inlining verification conditions; otherwise, consider TRE.

### 5.1 Inlining

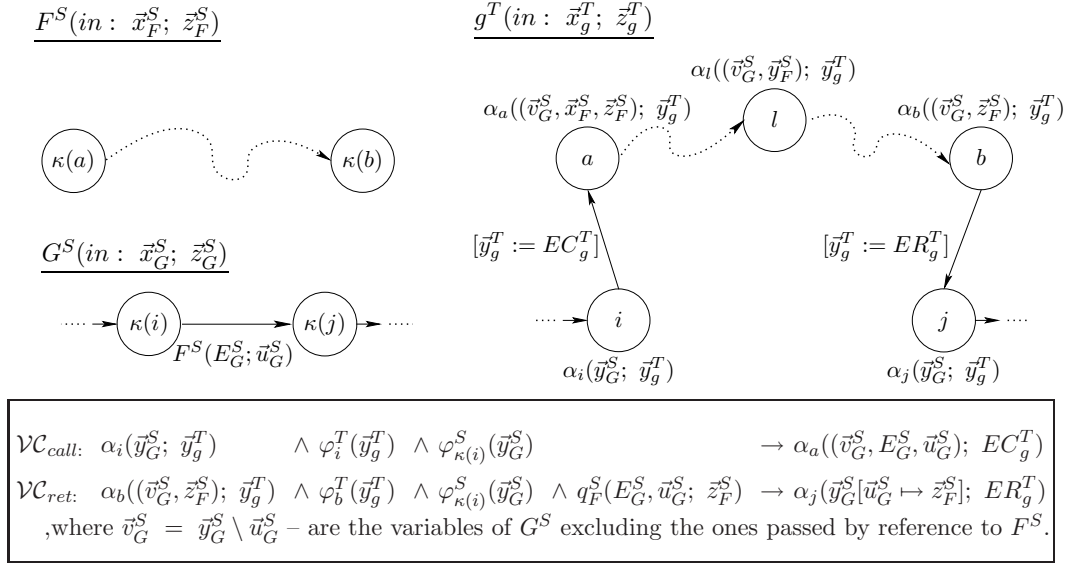


Fig. 4. INLINING VERIFICATION CONDITIONS: a call to procedure  $F^S$  is inlined.

Consider a case when a source call edge  $e^S = (\kappa(i), \kappa(j))$ , labeled by  $F^S(E_G^S, \vec{u}_G^S)$ , has been inlined. Suppose that the target locations  $i$  and  $j$  belong to some procedure  $g^T$ . To simplify this presentation, we assume that there is no nested inlining, so  $e^S$  belongs to  $G^S$  such that  $g^T$  is mapped to  $G^S$ . The target procedure should contain unconditional assignment transitions  $(i, a)$  and  $(b, j)$  that correspond to the call to and return from procedure  $F^S$  on the source. Assume,  $(i, a)$  is labeled by  $[\vec{y}_g^T := EC_g^T]$  and  $(b, j)$  is labeled by  $[\vec{y}_g^T := ER_g^T]$ , as depicted in Fig. 4.

Define a set of target locations  $L \subset C^T$  such that it includes all locations on every path from  $i$  to  $j$ . Note that all the locations in this set will be mapped to the nodes of the source procedure  $F^S$ . It is required that  $\alpha_l, l \in L$  does not depend on  $\vec{u}_G^S$ , the variables whose references are passed to  $F^S$ . However, we do allow the dependance on the corresponding formal parameters. This restriction comes from the fact that  $\vec{u}_G^S$  may change during the execution of  $F^S$ . Inlining Verification Conditions, presented in Fig. 4, are generated for each pair of target locations  $(i, j)$  that correspond to the inlined call edge  $(\kappa(i), \kappa(j))$ . The  $\mathcal{VC}_{call}$  condition checks the data abstraction associated with locations  $a$  and  $\kappa(a)$ , which are reached after the assignment on the target and the call on the source; the  $\mathcal{VC}_{ret}$  condition checks that the data abstraction holds at locations  $j$  and  $\kappa(j)$  – after the corresponding assignment and the return. This ensures that both of the target edges,  $(i, a)$  and  $(b, j)$ , contribute a condition to the set  $\mathcal{VC}$ .

### 5.2 Tail Recursion Elimination

A call edge  $(i, t)$  of a procedure  $f(\vec{x}; \&\vec{z})$  is a **TRE candidate** if it is a recursive call labeled by  $f(E(y); \vec{z})$  and  $t$  is the tail node of procedure  $f$ . Note that the formal input parameters  $\vec{z}$  are passed as the actual parameters in the tail call. Let

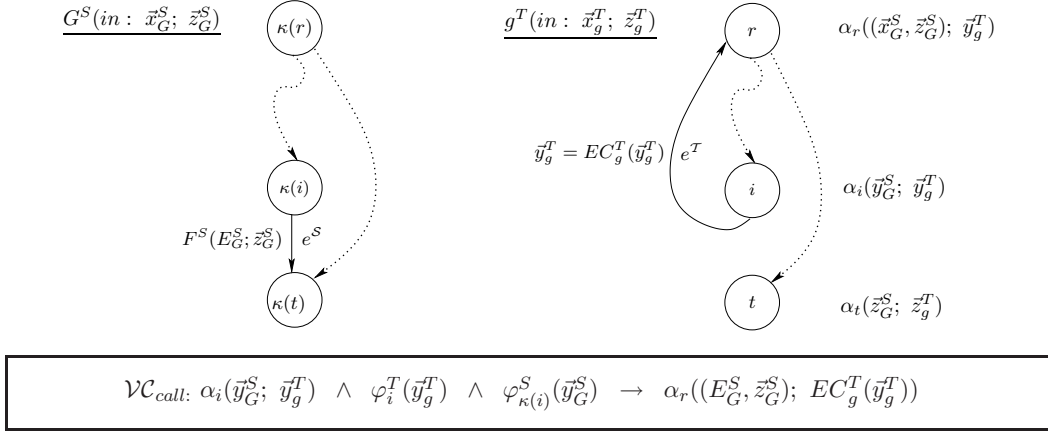


Fig. 5. TRE VERIFICATION CONDITIONS: a tail recursive call is eliminated in the procedure  $f$ .

$e^T = (i, r)$  be an unconditional assignment edge of the target procedure  $g^T$  such that there exists a TRE candidate source edge  $(\kappa(i), \kappa(t))$  labeled by  $G^S(E_G^S(\bar{y}_G^S), \bar{z}_G^S)$ , where the target procedure  $g^T$  is mapped to the source procedure  $G^S$ . Under these conditions, we guess that TRE optimization occurred and generate TRE Verification Condition, shown in Fig. 5. The condition checks that the data abstraction holds at the entry to the procedure: after a call on the source and the assignment on the target are performed. There is no target edge that corresponds to the return from the recursive call on the source and, consequently, the exit verification condition is not generated. Next, we explain why the requirements of the correct translation, as defined in Section 2.3, are still satisfied. Consider a source computation  $\sigma_S$  that contains  $m$  recursive calls to  $G^S$  and the corresponding target computation  $\sigma_T$ :

$$\begin{aligned} \sigma_S = & \dots \\ & \langle \kappa(r), D_r^0 \rangle \longrightarrow \dots \longrightarrow \langle \kappa(i), D_i^0 \rangle \xrightarrow{e^S} \\ & \dots \\ & \langle \kappa(r), D_r^{m-1} \rangle \longrightarrow \dots \longrightarrow \langle \kappa(i), D_i^{m-1} \rangle \xrightarrow{e^S} \\ & \langle \kappa(r), D_r^m \rangle \longrightarrow \dots \longrightarrow \langle \kappa(t), D_t^m \rangle \\ & \xrightarrow{ret} \langle \kappa(t), D_t^{m-1} \rangle \dots \xrightarrow{ret} \langle \kappa(t), D_t^1 \rangle \xrightarrow{ret} \langle \kappa(t), D_t^0 \rangle \dots \end{aligned}$$

$$\begin{aligned} \sigma_T = & \dots \\ & \langle r, d_r^0 \rangle \longrightarrow \dots \longrightarrow \langle i, d_i^0 \rangle \xrightarrow{e^T} \\ & \dots \\ & \langle r, d_r^{m-1} \rangle \longrightarrow \dots \longrightarrow \langle i, d_i^{m-1} \rangle \xrightarrow{e^T} \\ & \langle r, d_r^m \rangle \longrightarrow \dots \longrightarrow \langle t, d_t^m \rangle \\ & \dots \end{aligned}$$

$D_i^k$  and  $d_i^k$  denote source and target data interpretation, where  $k$  stands for the recursion level in the source program and the iteration level in the target program.

First, we want to show that  $\alpha_t(\bar{z}_G^S; \bar{z}_g^T)$  holds. Validity of the verification conditions generated for all target edges that end in  $t$  prove that  $\alpha_t(D_t^m; d_t^m)$  holds:  $\alpha_t$  holds before we take the very first return transition in the source. Note that the only source variables effecting  $\alpha_t(\bar{z}_G^S; \bar{z}_g^T)$  are the formal parameters passed by reference that match the actual parameters used for the tail call. Therefore, popping

the stack does not change  $\bar{z}_G^S$ , and  $\alpha_t$  is preserved by the return transitions.

Second, we show that for every target observation, there exists a stuttering equivalent source observation. Consider the observations of the source and target programs  $o_S$  and  $o_T$  that can be obtained by applying the observation function  $\mathcal{O}$  to the computations  $\sigma_S$  and  $\sigma_T$ :

$$\begin{array}{l}
 o_S = \dots \\
 \mathcal{O}(\langle \kappa(r), D_r^0 \rangle) \longrightarrow \dots \longrightarrow \mathcal{O}(\langle \kappa(i), D_i^0 \rangle) \xrightarrow{\top} \\
 \dots \\
 \mathcal{O}(\langle \kappa(r), D_r^{m-1} \rangle) \longrightarrow \dots \longrightarrow \mathcal{O}(\langle \kappa(i), D_i^{m-1} \rangle) \xrightarrow{\top} \\
 \mathcal{O}(\langle \kappa(r), D_r^m \rangle) \longrightarrow \dots \longrightarrow \mathcal{O}(\langle \kappa(t), D_t^m \rangle) \\
 \underbrace{\xrightarrow{\top} \top \dots \xrightarrow{\top} \top \xrightarrow{\top} \top \dots}_{m \text{ return transitions}} \\
 \\
 o_T = \dots \\
 \mathcal{O}(\langle r, d_r^0 \rangle) \longrightarrow \dots \longrightarrow \mathcal{O}(\langle i, d_i^0 \rangle) \xrightarrow{\top} \\
 \dots \\
 \mathcal{O}(\langle r, d_r^{m-1} \rangle) \longrightarrow \dots \longrightarrow \mathcal{O}(\langle i, d_i^{m-1} \rangle) \xrightarrow{\top} \\
 \mathcal{O}(\langle r, d_r^m \rangle) \longrightarrow \dots \longrightarrow \mathcal{O}(\langle t, d_t^m \rangle) \\
 \dots
 \end{array}$$

The verification conditions that we generate for each target edge ensure that for every target transition in  $\sigma_T$  there exists a corresponding source transition in  $\sigma_S$ . Furthermore, the I/O transitions (and the associated data) match. Thus, the source observation  $o_S$  can be obtained from the target observation  $o_T$  by adding exactly  $m$  pairs  $\xrightarrow{\top} \top$ .

## 6 Example

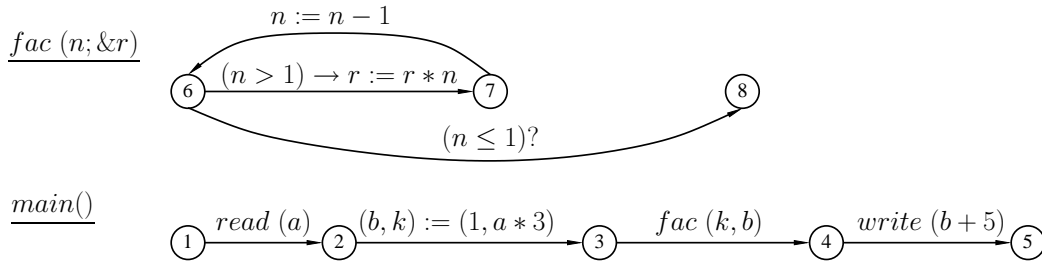


Fig. 6. The program from Fig. 1 after compiler optimizations.

The target program depicted in Fig. 6 is obtained from the source program shown earlier in Fig. 1 after TRE is applied to the procedure  $fac$ , the value of constant  $c$  is propagated, and the computation of the expression  $a * 3$  is moved due to instruction scheduling. Note our notation: we use capital letters to denote the source variables and procedure names; we use the lowercase counterparts for the target program. Let us apply the Translation Validation algorithm from Section 3 to prove that the target is the correct translation of the source.

**Step 1:** The control abstraction  $\kappa$  is identity.

**Step 2:** The inductive assertion network  $\mathcal{N}^S$  associates assertion  $(C = 3)$  with locations  $l \in \{3, 4, 5\}$  and assertion *true* with  $l \in \{1, 2, 6, 7, 8\}$ . All the assertions in  $\mathcal{N}^T$  are *true*. We omit the set of the verification conditions that prove inductiveness of  $\mathcal{N}^S$  and  $\mathcal{N}^T$  since they are straightforward.

**Step 3:** The following data abstraction is generated:

$$\begin{aligned}
 \alpha_1 &: \textit{true} \\
 \alpha_2 &: (A = a) & \alpha_6 &: (R = r) \wedge (N = n) \\
 \alpha_3 &: (A * 3 = k) \wedge (B = b) & \alpha_7 &: (R = r) \wedge (N = n) \\
 \alpha_4 &: (A * 3 = k) \wedge (B = b) & \alpha_8 &: (R = r) \\
 \alpha_5 &: \textit{true}
 \end{aligned}$$

**Step 4:** Below, we list selected translation verification conditions from the set  $\mathcal{VC}$ . We are going to omit the invariants that evaluate to *true*:

$$\underline{\textit{Read}} : \mathcal{VC}_{(1,2)} : \alpha_1 \wedge (A = a) \rightarrow \alpha_2 \Leftrightarrow \textit{true} \wedge (a = A) \rightarrow (a = A)$$

$$\begin{aligned}
 \underline{\textit{Assign}} : \mathcal{VC}_{(2,3)} : \alpha_2 \wedge \rho_{(2,3)}^T \rightarrow c_{(2,3)}^S; \alpha_2 \wedge \rho_{(2,3)}^T \wedge \rho_{(2,3)}^S \rightarrow \alpha'_3 \Leftrightarrow \\
 (A = a) \wedge ((b' = 1) \wedge (k' = a * 3) \wedge (a' = a)) \rightarrow \textit{true}; \\
 (A = a) \wedge ((b' = 1) \wedge (k' = a * 3) \wedge (a' = a)) \wedge \\
 ((B' = 1) \wedge (C' = 5) \wedge (A' = A)) \rightarrow ((A' * 3 = k') \wedge (B' = b'))
 \end{aligned}$$

$$\begin{aligned}
 \underline{\textit{Call}} : \mathcal{VC}_{(3,4)} : \mathcal{VC}_{\textit{call}} : \alpha_3 \wedge \varphi_3^S \rightarrow \alpha_6[(N, R) \mapsto (A * 3, B); (n, r) \mapsto (k, b)]; \\
 \mathcal{VC}_{\textit{ret}} : \alpha_3 \wedge \varphi_3^S \wedge \alpha_8 \rightarrow \alpha_4[B \mapsto R; b \mapsto r] \Leftrightarrow \\
 ((A * 3 = k) \wedge (B = b)) \wedge (C = 5) \rightarrow ((A * 3 = k) \wedge (B = b)); \\
 ((A * 3 = k) \wedge (B = b)) \wedge (C = 5) \wedge (R = r) \rightarrow ((A * 3 = k) \wedge (R = r))
 \end{aligned}$$

$$\begin{aligned}
 \underline{\textit{Write}} : \mathcal{VC}_{(4,5)} : \alpha_4 \wedge \varphi_4^S \rightarrow \alpha_5 \wedge (B + C = b + 5) \Leftrightarrow \\
 ((A * 3 = k) \wedge (B = b)) \wedge (C = 5) \rightarrow \textit{true} \wedge (B + C = b + 5)
 \end{aligned}$$

$$\begin{aligned}
 \underline{\textit{TRE}} : \mathcal{VC}_{(7,6)} : \alpha_7 \rightarrow \alpha_6[N \mapsto (N - 1); n \mapsto (n - 1)] \Leftrightarrow \\
 ((R = r) \wedge (N = n)) \rightarrow ((R = r) \wedge (N - 1 = n - 1))
 \end{aligned}$$

**Step 5:** We use an automatic theorem prover, such as [3,1], to check the generated conditions for validity. Since all the conditions in  $\mathcal{VC}$  are valid, we conclude the correctness of the translation.

## 7 Conclusion and Future Work

We presented a novel framework for automatic translation validation of reactive programs in presence of interprocedural optimizations. Since all the translation validation approaches mentioned in Section 1 deal with infinite state systems, they cannot hope to have a complete method for proving correct translation in general. However, because the focus is only on compiler optimizations, the number of false alarms can be drastically minimized or even eliminated. Intuitively, since we are aware of the analysis used by the optimizing compilers, we are optimistic in creation of a strong enough set of auxiliary invariants.

We are currently developing a tool that verifies the optimizations performed by LLVM compiler and uses CVC3 [1] as the back end validity checker. In addition,

the framework has yet to be extended to incorporate more language features such as dynamic memory allocation and exceptions.

Another long-term goal of ours is development of a self-certifying compiler or program transformation engine that would allow third parties to specify the desired program transformations. The need for verification is even more obvious here. Therefore, an interesting research direction is construction of a transformation specification language that provides enough information for automatic translation validation. This idea is similar to [9]. However, since our approach is translation validation, we do not insist that all the transformations are provably correct for all the programs. Therefore, we can be more flexible and capture a larger set of program transformations.

## References

- [1] *CVC3: An Automatic Theorem Prover for Satisfiability Modulo Theories (SMT)*, <http://www.cs.nyu.edu/acsys/cvc3/>.
- [2] *The LLVM Compiler Infrastructure Project*, <http://llvm.org>.
- [3] *The YICES SMT Solver*, <http://yices.csl.sri.com>.
- [4] Aho, A. V., R. Sethi and J. D. Ullman, “Compilers: Principles, Techniques, and Tools,” Addison-Wesley Longman Publishing Co., Inc., 1986.
- [5] Blech, J. O. and A. Poetsch-Heffter, *A certifying code generation phase*, in: *Proceedings of the 6<sup>th</sup> International Workshop on Compiler Optimization meets Compiler Verification*, 2007.
- [6] Fang, Y. and L. Zuck, *Generating invariants for translation validation*, in: *Proceedings of the 5<sup>th</sup> International Workshop on Compiler Optimization meets Compiler Verification*, 2006.
- [7] Floyd, R. W., *Assigning meanings to programs*, , **19:19-32**, 1967.
- [8] Goos, G. and W. Zimmermann, *Verification of compilers*, in: *Correct System Design*, 1999, pp. 201–230.
- [9] Lerner, S., T. Millstein, E. Rice and C. Chambers, *Automated soundness proofs for dataflow analyses and transformations via local rules*, in: *POPL*, 2005.
- [10] Muchnick, S. S., “Advanced Compiler Design and Implementation,” Morgan Kaufmann, 1997.
- [11] Necula, G. C., *Translation validation for an optimizing compiler.*, in: *Programming Language Design and Implementation* (2000), pp. 83–95.
- [12] Pnueli, A., *Verification of procedural programs*, in: *We Will Show Them! Essays in Honour of Dov Gabbay, Volume Two* (2005), pp. 543–590.
- [13] Pnueli, A., O. Shtrichman and M. Siegel, *Translation validation for synchronous languages*, Lecture Notes in Computer Science **1443** (1998), pp. 235–250.
- [14] Pnueli, A., M. Siegel and O. Shtrichman, *Translation validation: From SIGNAL to C*, in: *Correct System Design*, LNCS State-of-the-Art Survey **1710** (1999), pp. 231–255.
- [15] Reps, T., S. Horwitz and M. Sagiv, *Precise interprocedural dataflow analysis with applications to constant propagation*, in: *Proc. of the Sixth International Joint Conference CAAP/FASE*, Aarhus, Denmark, 1995.
- [16] Rival, X., *Symbolic transfer function-based approaches to certified compilation*, in: *31st Symposium on Principles of Programming Languages* (2004), pp. 1–13.
- [17] Tristan, J.-B. and X. Leroy, *Formal verification of translation validators: a case study on instruction scheduling optimizations*, in: *POPL* (2008).
- [18] Zaks, A. and A. Pnueli, *Translation validation of interprocedural optimizations*, Technical report, NYU (2007), <http://www.cs.nyu.edu/~ganna/home/TVI0TR.pdf>.
- [19] Zuck, L., A. Pnueli, Y. Fang and B. Goldberg, *Voc: A methodology for the translation validation of optimizing compilers*, in: *Journal of Universal Computer Science*, 2003.
- [20] Zuck, L., A. Pnueli, B. Goldberg, C. Barrett, Y. Fang and Y. Hu, *Translation and run-time validation of loop transformations*, , **27(3)**, 2005, pp. 335–360.