Translation Validation of Interprocedural Optimizations

Amir Pnueli\textsuperscript{1} and Ganna Zaks\textsuperscript{1}

New York University, New York, \{amir,ganna\}@cs.nyu.edu

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 (implementation) of the source code.

In this work, we extend the existing framework for translation validation of optimizing compilers to deal with procedural programs and define the notion of correct translation for reactive programs with intermediate inputs and outputs. We present an Interprocedural Translation Validation algorithm that automatically proves correct translation of the source program to the target program in presence of interprocedural optimizations such as global constant propagation, inlining, tail-recursion elimination, interprocedural dead code elimination, dead argument elimination, and cloning.

1 Introduction

The effort of program correctness verification is extensive. First, the programmer examines the code and tests it, usually with compiler optimizations turned off. Then, numerous verification tools and techniques can be applied to verify that the code satisfies the desired properties. After all the rigorous checks are complete, it is compiled by an optimizing compiler and released. Nevertheless, our verification effort should not stop here. Compilers are quite large applications, which are bound to have bugs. At present, the GCC Bug Database contains 3217 reported bugs. Clearly, it is highly desirable to ensure that the transformations performed by a compiler preserve the semantics of a program and do so automatically.

The methodology of compiler verification can be categorized by its intended customers (i.e., by the ultimate consumer). 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. A work in this direction is presented in [16], which introduces the notion of Credible Compiler. Each component of a credible compiler performs a specific transformation and produces a proof that the transformation is correct. Another approach is taken in [10, 9], which present languages for specification of compiler optimizations that can be automatically proved sound, meaning that their transformations are always semantics-preserving. This approach assumes the correctness of the execution engine.

Another group interested in compiler verification are compiler users who may need to work with a black box and require tools that accommodate minimal compiler cooperation. Good examples of such tools are presented in [12] and [3]. The tools are based on the technique of translation validation [13]; they check the result of each compilation against the source program, and rely on heuristics and available compiler annotations to detect the transformations that take place. The frameworks assume minimal cooperation from the compiler; however, they are also well suited for development of a self-certifying compiler, thus, contributing to the first direction described above as well.

To the best of our knowledge, existing translation validation approaches are not capable and were not designed to deal with interprocedural optimizations. For example, in [12] two executions are considered the same if both lead to the same sequence of function calls and returns. In this paper, we extend the deductive method for automatic translation validation presented in [3, 18, 19, 6] to programs with procedures.
In contrast to the previous work, which used transition systems as the formal model, we rely on the transition graphs that capture not only conditions and assignments but also procedure calls and read/write operations. Note that the set of represented programs is not limited to the deterministic programs, as before, but includes reactive systems driven by intermediate inputs. The notion of correct translation has also been extended accordingly. Generally, the target program $P_T$ is a correct translation (refinement) of program $P_S$ if every observation of $P_T$ is also an observation of $P_S$. 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 read and write instructions. Finally, we show how to generate the auxiliary invariants used for verification of context sensitive copy propagation and present Interprocedural Translation Validation algorithm that proves correct translation of $P_S$ to $P_T$ in presence of interprocedural optimizations like global constant propagation, inlining, tail-recursion elimination, interprocedural dead code elimination, dead argument elimination, and cloning. Our algorithm is strong enough to handle most, if not all, of the interprocedural optimizations described in literature [11, 1] and performed by compilers such as GCC, ORC, and LLVM [7, 8].

The rest of the paper is organized as following. Section 2 introduces our formal model. Section 3 presents inductive assertion network. We define the notion of correct translation in Section 4. Section 5 introduces the Translation Validation algorithm. Section 6 and Section 7 present the set of verification conditions used to prove correct translation. We conclude in Section 8.

2 Underlining Model of Programs with Procedures

Our formal model and the verification techniques are based on the ones introduced in [15] for verification of procedural programs.

2.1 Transition Graphs

A program (application) $A$ consists of $m + 1$ modules: $P_0, P_1, \ldots, P_m$, where $P_0$ represents the main procedure, and $P_1, \ldots, P_m$ are procedures which may be called from $P_0$ or from other procedures. The variables of each module $P_i$ are partitioned into $\vec{y} = (\vec{x}, \vec{z}, \vec{w})$, where $\vec{x}$ and $\vec{z}$ are the input parameters of a module and $\vec{w}$ denotes local(working) variables of the module. Call-by-value parameter passing method is used for $\vec{x}$, and call-by-reference is used for $\vec{z}$. For simplicity of this presentation, we assume that the main procedure $P_0$ does not take any parameters. Thus, the observable behavior of the program is expressed solely by the read and write instructions. We also assume that there are no variable name conflicts among the procedures.

Each module $P_i$ is presented as a transition graph with nodes (locations)

$$\mathcal{L}_i = \{ r^i = l_0, l_1, l_2, \ldots, l_k = t^i \}.$$ 

It must have distinct root node $r^i$ as its only entry point, distinct tail node $t^i$ as its only exit point, and every other node must be on a path from $r^i$ to $t^i$. Node $r^i$ should not have any incoming edges and node $t^i$ should not have any outgoing edges. Nodes of the graph are connected by directed edges labeled by instructions, which must be either guarded assignments, procedure calls, or read/write operations. Consider a procedure module $g(in: \vec{x}; \vec{z})$ with $\vec{y} = (\vec{x}, \vec{z}, \vec{w})$. Let $\vec{u} \subseteq \vec{y}$ 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 $c$ is a boolean expression. When the assignment part is empty, we abbreviate the label to a pure condition $c$?

- Read and write instructions are denoted by $\text{read}(\vec{u})$ and $\text{write}(\vec{u})$. 


**Procedure call** instruction \( f(E(\vec{y}), \vec{u}) \) denotes a call to module \( f(\text{in} : \vec{x}_f; \vec{z}_f) \), passing input parameters \( E(\vec{y}) \) and \( \vec{u} \).

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. The transition graphs represent a deterministic system when, for every node \(i\), the guards of all edges departing from \(i\) are mutually exclusive.

**Example 1** The program depicted on Fig. 1 reads in a natural number, computes its integer square root, and writes out the result. Note that the assignment instructions may assign to more than one variable at a time.

![Procedure diagram](image)

**Fig. 1.** Procedure \( P_1(\text{in} : x; z_1) \) computes square root of \( x \) and returns the result by reference.

### 2.2 States and Computations

For simplicity, we assume that all variables of a module range over the same domain \( D \) (say, the integers). We denote by \( \vec{d} = (d_1, \ldots, d_n) \) a tuple of \( D \)-values, which represents an interpretation (i.e., an assignment of values) of the module variables.

**Definition 1** A state of module \( P \) is a pair \( \langle l; \vec{d} \rangle \) consisting of a location \( l \) and a data interpretation \( \vec{d} \).

**Definition 2** A \((\xi, \zeta)\)-computation of module \( P \) is a maximal sequence of states and their labeled transitions:

\[ \sigma : \langle r; (\xi, \zeta, \vec{\top}) \rangle \xrightarrow{\lambda_1} \langle l_1; \vec{d}_1 \rangle \xrightarrow{\lambda_2} \langle l_2; \vec{d}_2 \rangle \ldots \]

The tuple \( \vec{\top} \) denotes uninitialized values. At the first state of the computation, the location is \( r \), the entry location of \( P \); the values of input variables \( \vec{x} \) and \( \vec{z} \) are set to \( \xi \) and \( \zeta \), respectively, and
the local variables $\bar{w}$ are not initialized. Labels in the transitions are either names of edges in the program or the special label $ret$. Each transition in a computation 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 Section A for the formal definition.

Computations of $P_0$ constitute the set of computations of a procedural program $\mathcal{A}$.

2.3 From Programs to Transition Graphs

In this section, we describe how to construct the formal model of a program. A set of cut-points is a set of program locations $C$ such that:

- At least one location in each loop belongs to $C$.
- For every procedure, both procedure entry and exit belong to $C$.
- For every procedure call edge $(i, j)$, locations $i$ and $j$ belong to $C$.
- The locations right before and after each read/write operation belong to $C$.

The choice of cut-point set can be generalized not to require at least one cut-point per each loop but to ensure that the transitions between every pair of cut-points are computable [6].

Each procedure whose implementation is given is represented by a transition graph. We choose the set of cut-points of a procedure $P_k$ to be the set of nodes for the corresponding transition graph. If there exists a path $\pi$ from cut-point $i$ to cut-point $j$, which does not pass through any other cut-point, we add edge $(i, j)$ 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, $\text{pow}$ function in C), the call is modeled by uninterpreted functions. Global variables and functions can be efficiently modeled in this framework. Consult Section B for the details.

At present, the framework has yet to be extended to incorporate some language features with most noticeable omissions of dynamic memory allocation and exceptions.

3 Inductive Assertion Network

3.1 Assertions

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

- For each module $P_k$, we denote $\varphi_{l\text{e}}$, the assertion associated with the entry location, by $p_k$. The input predicate $p_k = p_k(\bar{X}, \bar{Z}; \bar{x}, \bar{z})$ imposes constraints only on the input variables of the module. Since we assume that the main module $P_0$ does not have input parameters, $p_0 = \text{true}$.
- Similarly, we denote $\varphi_{l\text{e}}$, the assertion associated with the exit location, by $q_k$. The output predicate $q_k = q_k(\bar{X}, \bar{Z}; \bar{z})$ is the procedure summary: it specifies the relation between the input and output values.
- The assertions at all other locations of the procedure $\varphi_l(\bar{Y})$ may depend on any of the variables.

3.2 Verification Conditions

For each edge of the transition graph $e$ connecting cut-point $i$ and $j$, we form verification conditions. We consider five types of verification conditions (one for each edge type):
• **Guarded Assignment**: If $e$ is an assignment edge labeled by $c \rightarrow [\vec{u} := E(\vec{y})]$, 
\[ \forall C_e : \varphi_i(\vec{Y}) \land c(\vec{y}) \rightarrow \varphi_j(\vec{Y})[\vec{u} \mapsto E(\vec{y})], \]
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 $\text{read}(\vec{u})$, 
\[ \forall C_e : \varphi_i(\vec{Y}) \rightarrow \varphi_i(\vec{Y})[\vec{u} \mapsto \vec{u}'], \]
where $\vec{u}'$ are fresh variables. The invariant $\varphi_j$ must hold for all possible inputs.

• **Write**: If $e$ is a write edge labeled by $\text{write}(\vec{u})$, 
\[ \forall C_e : \varphi_i(\vec{Y}) \rightarrow \varphi_j(\vec{Y}). \]

• **Procedure call**: We associate the following two conditions with a procedure call $P_k(E(\vec{y}), \vec{u})$:
\[ \forall C_{\text{call}} : \varphi_i(\vec{Y}) \rightarrow p_k(E(\vec{y}), \vec{u}; E(\vec{y}), \vec{u}) \]
\[ \forall C_{\text{return}} : \varphi_i(\vec{Y}) \land q_k(E(\vec{y}), \vec{u}; \vec{z}_k) \rightarrow \varphi_j(\vec{Y})[\vec{u} \mapsto \vec{z}_k] \]

\[ P_k(\text{in} : \vec{x}_k; \vec{z}_k) \]
\[ p_k(\vec{X}_k, \vec{Z}_k; \vec{x}_k, \vec{z}_k) \text{……} \]
\[ \text{.....} \quad \varphi_i(\vec{Y}) \quad \text{.....} \]
\[ e \quad P_k(E(\vec{y}), \vec{u}) \quad j \]
\[ \varphi_j(\vec{Y}) \]

**Fig. 2. Verification Conditions**: procedure $P$ calls procedure $P_k$.

**Definition 3** An assertion network $\mathcal{N} = \{\varphi_0, \ldots, \varphi_n\}$ for a program $\mathcal{A}$ is said to be **inductive** if all the verification conditions for all edges in $\mathcal{A}$ are valid.

**Definition 4** Network $\mathcal{N}$ is said to be **invariant** if for every execution state $(l; \vec{d})$ occurring in a computation, $d \models \varphi_l$. That is, on every visit of a computation of node $l$, the visiting data state satisfies the corresponding assertion $\varphi_l$ associated with $l$.

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

### 4 Correct Translation

Similarly to [14], the notion of correct translation used in this work is based on the general notion of refinement between source (abstract) program $\mathcal{S}$ and target (concrete) program $\mathcal{T}$.

We associate **observation function** $\mathcal{O}$ with each program, mapping the source and target transition labels and states into a common domain $R$. The observation function needs to ensure that read and write transitions of the source and target computations match. Formally, given a transition label $\lambda$, an observation function $\mathcal{O}(\lambda)$ is defined as following:
- If $\lambda$ is a name of a read edge, $\mathcal{O}(\lambda) = \text{read};$
- If $\lambda$ is a name of a write edge, $\mathcal{O}(\lambda) = \text{write};$
- Otherwise, $\mathcal{O}(\lambda) = \top.$
Given a state \( s = (l, d) \), an observation function \( \mathcal{O}(s) \) is defined as following:

- If \( s \) is a state immediately after a read transition \( \text{read}(\vec{u}) \) and before a write transition \( \text{write}(\vec{v}) \), \( \mathcal{O}(s) = \vec{d}_{\vec{u},\vec{v}} \). We obtain \( \vec{d}_{\vec{u},\vec{v}} \) by restricting \( \vec{d} \) only to the values that correspond to the variables in \( \vec{v} \) and \( \vec{u} \).
- If \( s \) is a state immediately after a read transition \( \text{read}(\vec{u}) \), \( \mathcal{O}(s) = \vec{d}_{\vec{u}} \).
- If \( s \) is a state immediately before a write transition \( \text{write}(\vec{v}) \), \( \mathcal{O}(s) = \vec{d}_{\vec{v}} \).
- For all other states, \( \mathcal{O}(s) = \top \).

An observation of a program \( \mathcal{A} \) is a (possibly finite) sequence of \( R \)-elements and their \( R \)-labeled transitions which can be obtained by applying the observation function \( \mathcal{O} \) to each state and each transition label in the computation of \( \mathcal{A} \). That is, for \( \sigma : s_1 \xrightarrow{\lambda_1} s_2 \xrightarrow{\lambda_2} s_3 \xrightarrow{\lambda_4} s_4 \ldots \), some computation of \( \mathcal{A} \), we get the following observation:

\[
o : \mathcal{O}(s_1) \xrightarrow{\mathcal{O}(\lambda_1)} \mathcal{O}(s_2) \xrightarrow{\mathcal{O}(\lambda_2)} \mathcal{O}(s_3) \xrightarrow{\mathcal{O}(\lambda_4)} \mathcal{O}(s_4) \ldots
\]

We denote by \( \mathcal{O}(\mathcal{A}) \) the set of all observations of program \( \mathcal{A} \).

**Definition 5** Program \( \mathcal{T} \) is a refinement of program \( \mathcal{S} \), denoted \( \mathcal{T} \sqsubseteq \mathcal{S} \), if \( \mathcal{O}(\mathcal{T}) \subseteq \mathcal{O}(\mathcal{S}) \); that is, if every observation of \( \mathcal{T} \) is also an observation of \( \mathcal{S} \).

**Definition 6** Program \( \mathcal{T} \) is stuttering equivalent to program \( \mathcal{T}' \) if for every observation \( o_{\mathcal{T}'} \in \mathcal{O}(\mathcal{T}') \), there exists a corresponding observation \( o_{\mathcal{T}} \in \mathcal{O}(\mathcal{T}) \) such that \( o_{\mathcal{T}'} \) and \( o_{\mathcal{T}} \) only differ from each other by finite sequences of pairs \( \top \xrightarrow{\mathcal{T}} \top \) or \( \top \xrightarrow{\mathcal{T}} \top \).

For example, \( o \) and \( o' \) are stuttering equivalent:

\[
o : \top \xrightarrow{\text{read}(5,22)} \top \xrightarrow{\text{write}(110)} \top \xrightarrow{\text{read}(5,22)} \top \xrightarrow{\text{write}(110)} \top
\]

\[
o' : \top \xrightarrow{\text{read}(5,22)} \top \xrightarrow{\text{write}(110)} \top \xrightarrow{\text{write}(110)} \top
\]

**Definition 7** We say that program \( \mathcal{T} \) is a correct translation of program \( \mathcal{S} \), if \( \mathcal{T} \sqsubseteq \mathcal{S} \) or \( \mathcal{T}' \sqsubseteq \mathcal{S} \) and \( \mathcal{T}' \) is stuttering equivalent to \( \mathcal{T} \).

For deterministic programs \( \mathcal{S} \) and \( \mathcal{T} \), in which the input sequence uniquely determines the computation, \( \mathcal{T} \sqsubseteq \mathcal{S} \) if \( \mathcal{S} \sqsubseteq \mathcal{T} \). Thus, the notion of correct translation for deterministic programs is an equivalence relation.

## 5 Translation Validation

Given two procedural programs \( \mathcal{S} \) and \( \mathcal{T} \), Interprocedural Translation Validation algorithm, presented in Fig. 3, proves that target program \( \mathcal{T} \) is a correct translation of source program \( \mathcal{S} \). It is an extension of the rule Validate [19] to programs with procedures and intermediate inputs/outputs. The algorithm is based on the computational induction approach [5], originally introduced for proving properties of a single program. Its main restriction is that it assumes that there exists a mapping from the nodes(cut-points) of the target program \( \mathcal{T} \) to the nodes of the source program \( \mathcal{S} \). Most of the classical compiler optimizations such as constant folding, induction variable optimizations, branch optimizations, common subexpression elimination, inlining, tail recursion elimination, and others, fall into this category. Rules for loop reordering transformations [19, 6] can be additionally applied to verify transformations such as loop interchange, fusion, distribution and tiling.

As the first step, we create control abstraction \( \kappa \), a mapping from the set of target nodes to the set of source nodes. Note that the control abstraction not only specifies the mapping between
To prove that program $T$ with the set of nodes (cut-points) $C^T$ is a correct translation of program $S$ with nodes $C^S$, follow the steps below.

1. Establish control abstraction $\kappa : C^T \rightarrow C^S$ mapping the target nodes to the source nodes, such that $r$ is the initial location (root of the main module) of $T$ iff $\kappa(r)$ is the initial location of $S$.

2. Construct sets of target and source invariants that form inductive networks
$$N^T = \{\phi^T_0, \ldots, \phi^T_{|C^T|}\}$$
and
$$N^S = \{\phi^S_0, \ldots, \phi^S_{|C^S|}\}$$
for programs $T$ and $S$, respectively. Form verification conditions that prove inductiveness of the source and target networks. Add the generated conditions to the set $VC$.

3. Form data abstraction $D = \{\alpha_0, \ldots, \alpha_{|C^T|}\}$ by defining each $\alpha_l(V^S; V^T)$ as a conjunction of equalities of the form $v^S = E(v^T)$ at each target node $l \in C^T$, where $V^S$ and $V^T$ denote the sets of variables that belong to programs $S$ and $T$, respectively. The data abstraction must be valid at the initial location of $T$, $\alpha_r = \text{true}$. Form translation verification conditions for every edge of the target program and add them to the set of verification conditions $VC$. If there exists an edge of the target program that does not contribute any verification conditions, generate Error.

4. Establish validity of every verification condition in $VC$; generate Error otherwise.

Fig. 3. INTERPROCEDURAL TRANSLATION VALIDATION

the program locations but also imposes many-to-one correspondence between target and source procedures. For example, consider target procedure $P^T_g$ with the root node $r$ and the tail node $t$. $P^T_g$ corresponds to source procedure $P^S_g$ with the root node $\kappa(r)$ and the tail node $\kappa(t)$.

We also construct the data abstraction $D$ that maps the values of target variables at location $l$ to the values of source variables at location $\kappa(l)$. Similar to the verification conditions used to prove the assertion network inductive, translation verification conditions, presented in Section 6 and Section 7, 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.

In case any conditions in $VC$ are not valid, or if there exists a target transition that does not have a corresponding transition on the source, we generate Error, which signifies that either an error in translation is detected or we ran into a transformation that is not currently supported.

5.1 Generation of Control Abstraction, Data Abstraction, and Inductive Networks

The methods for data abstraction construction and invariant generation are presented in [3, 4]. Construction of $D$ is based on refining a candidate data abstraction, obtained from the compiler annotations. Generally, each invariant is based on the set of reachable definitions (definitions that must hold at a particular location). We use the data abstraction and invariants constructed by these methods as the foundation and show how to extend them when necessary.

Control Abstraction $\kappa$ is a mapping from the nodes of the target program to the nodes of the source program. The mapping is total but does not have to 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. To construct the control abstraction, we first generate the set of source cut-points $C^S$ such that they satisfy the minimal requirements stated in Section 2.3. Then, we rely on the compiler annotations to assist in computation of the control abstraction $\kappa$ and $C^T$. Finally, we check the $C^T$ for completeness with respect to the requirements of Section 2.3.

The compiler annotations that the methods depend on are also required for debugging, so they are provided by most compilers.
5.2 Strengthening Source Inductive Network

The inductive network $N^S = \{\varphi^S_0, \ldots, \varphi^S_{(S)}\}$ has to be augmented so that it incorporates the information essential to proving interprocedural optimizations. We are going to use [17] 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. In this section, we show how to generate the source invariant network that is strong enough for context sensitive copy constant propagation. Linear constant propagation can be handled in a similar fashion.

As a first try, it appears that any precise solution to the interprocedural constant-propagation problem should suffice. For example, $\varphi^S_l$ should be extended with conjunct $x = 17$ if $x$ always evaluates to constant $17$ at location $l$. However, the resulting network $N^S$ 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. Intuitively, 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 $L = \mathbb{Z}_+^T$ be the integer constant propagation lattice. We denote the meet operator by $\cap$. The set $Env(V, L)$ of environments is the set of functions from $V$ to $L$. A mapping $T : Env(V, L) \hookrightarrow Env(V, L)$ is called an environment transformer. A transformer $T$ is distributive iff for every variable $v \in V$, $(T(\cap_i env_i))(v) = \cap_i (T(env_i))(v)$. The algorithm in [17] essentially computes a transformer $T_{(t_k,l)}$ between the root of each procedure $P_k$ and every location in $L_k$. Note that the transformer $T_{(t_k,l)}$ 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 [17]. Every distributive transformer $T$ can be represented using a set of functions $F_T = \{f_{v,v'} | v,v' \in V \cup \{A\}\}$, each of type $L \rightarrow L$. Function $f_{v,v'}$ 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 $f_{v,v'} = \lambda l. T$. Function $f_{A,v'}$ is used to represent the effects of on the variable $v$ that are 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: $T(env)(v') = f_{A,v'} \cap (\cap_{v \in V} f_{v,v'}((env)(v)))$. Since we are only concerned with constant copy propagation analysis, all the functions in $F_T$ will be either identities or constants.

Example 2 Consider the example in Fig. 4. Below is the list of environment transformers computed by [17] for procedure $foo$. We omit all the functions that evaluate to top $f_{(v,v')} = \lambda l. T$.

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

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

$F_{(2,4)} = \{ f_{c,c} = \lambda l. l, f_{x,z} = \lambda l. l, f_{y,z} = \lambda l. l \}$

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

- We ignore all functions of the form $f_{(v,v')} = \lambda l. T$.
- For each variable $v'$ that is not set to $\bot$ by $f_{(A,v')} \in F_{(t_k,l)}$ we add the following conjunct to $\varphi_l$: 
  \[ \bigvee_{f_{v,v'} \in F_{(t_k,l)}} v' = f_{v,v'}(V), \] where $V$ represents the value of $v$ at the procedure entry.

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

- We also add the conjunct $x = \text{const}$ if $x$ was determined to evaluate to constant $\text{const}$ at location $l$. We need this addition since $T_{(t_k,l)}$ does not propagate the information from the callers.
The resulting invariants, denoted in Fig. 4 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.

\[
\forall \text{ret}: \phi_1 \land \phi_4[(C, Y) \mapsto (5, y_m)] \rightarrow \phi_3[z_m \mapsto z] \iff \\
y_m = 5 \land c = 5 \land (z = 5 \lor z = y_m) \land c = 5 \rightarrow z = 5
\]

6 Translation Verification Conditions in Presence of Structure Preserving Optimizations

This section gives a recipe of generating translation verification conditions when the structure of the transformed program is preserved. This means that, for every edge of the target program \(e^T\) connecting nodes \(i\) and \(j\), there exist the corresponding source edges between nodes \(\kappa(i)\) and \(\kappa(j)\).

For every edge of the target program \(e^T\) connecting nodes \(i\) and \(j\) and \(e^S = (\kappa(i), \kappa(j))\), form translation verification conditions according to one of the following rules.

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

  \[
  \alpha_i \land \varphi_i^S \land \varphi_i^T \land \rho_{e^T} \rightarrow (\bigvee_{e^S \in \text{Edges}(\kappa(i), \kappa(j))} \text{Cond}_{e^S}).
  \]

  \[
  \alpha_i \land \varphi_i^S \land \varphi_i^T \land \rho_{e^T} \land (\bigvee_{e^S \in \text{Edges}(\kappa(i), \kappa(j))} \rho_{e^S}) \rightarrow \alpha_j.
  \]

In the assertions above, \(\rho_{e^S}\) and \(\rho_{e^T}\) are defined as follows: for edge \(e \in \{e^S, e^T\}\) labeled by guarded assignment \(c \rightarrow [\bar{u} := E(\bar{y})]\),

\[
\rho_e = c \land (\bar{u}' = E(\bar{y}')) \land \bar{v}' = \bar{u},
\]

where \(\bar{v}'\) are all variables of \(\alpha_j\) with the exception of those in \(\bar{u}\).

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 \( \text{read}(\bar{u}^T) \) and \( \text{read}(\bar{u}^S) \),
  \[ \alpha_i \land \varphi_i^T \land \varphi_{\nu(i)}^S (\bar{u}^T = \bar{u}^S) \implies \alpha_j. \]

- **Write**: If \( e^T \) and \( e^S \) are both labeled by write instructions \( \text{write}(E^T) \) and \( \text{write}(E^S) \),
  \[ \alpha_i \land \varphi_i^T \land \varphi_{\nu(i)}^S \implies \alpha_j \land (E^T = E^S). \]

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

![Call Verification Conditions](image)

**Fig. 5. Call Verification Conditions**: procedure \( P_i^S \) calls procedure \( P_j^S \) in the source program and procedure \( P_i^T \) calls procedure \( P_j^T \) in the target program.

- **Procedure Call**: If both \( e^T \) and \( e^S \) are call edges labeled by \( P_i^T (E_j^T; \bar{u}_j^T) \) and \( P_i^S (E_j^S; \bar{u}_j^S) \), respectively, where \( P_i^T \) is mapped to \( P_j^S \), we generate Call Verification Conditions presented in Fig. 5, which check that the data abstraction is preserved by stepping through the procedure calls. The entry condition checks that the data mapping holds at the entry to the procedure. The exit condition guarantees that it holds after the procedure return. Note that if procedure \( P_i^T \) is not mapped to procedure \( P_j^S \), *Error* should be generated.

## 7 Translation Verification Conditions in Presence of Inlining and TRE

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 \( P^S_P(E^S_G; \vec{u}^S_G) \); \( \kappa(a) \) is the root of \( P^S_P \); and there exists the corresponding node \( b \) in the target such that \( \kappa(b) \) is the tail of the procedure \( P^S_P \). If \((b, j)\) is an unconditional assignment transition of \( T \), proceed with generating Inlining Verification Conditions; otherwise, consider TRE.

7.1 Inlining

\[
P^S_P(\text{in} : x^S_P; \vec{x}^S_P) \quad \text{and} \quad P^T_T(\text{in} : x^T_T; \vec{x}^T_T)
\]

\[
\begin{array}{c}
\kappa(a) \\
\kappa(i) \\
P^S_P(E^S_G; \vec{u}^S_G) \\
\kappa(j) \\
\kappa(b)
\end{array}
\]

\[
\begin{array}{c}
\alpha_i((\vec{u}^S_G; \vec{x}^S_P); \vec{y}^S_T) & \land \varphi^T(\vec{y}^T) & \land \varphi^S_\kappa(i)(\vec{y}^S_T) \\
\rightarrow \alpha_a((\vec{u}^S_G; \vec{v}^S_G; \vec{x}^S_P); \vec{y}^T) \\
\varphi^S_\kappa(i)(\vec{y}^S_G) & \land \varphi^T(\vec{y}^T) & \land \varphi^S_\kappa(j)(\vec{y}^S_G) & \land \varphi^G(\vec{E}^S_G; \vec{u}^S_G; \vec{x}^S_P) \\
\rightarrow \alpha_j((\vec{y}^S_G; \vec{u}^S_G; \vec{w}^S_G); \vec{y}^T) \\
\text{where} \quad \vec{v}^S_G = \vec{y}^S_G \setminus \vec{u}^S_G
\end{array}
\]

Fig. 6. **Inlining Verification Conditions**: a call to procedure \( P^S_P \) is inlined.

Consider a case when a source call edge \( e^S = (\kappa(i), \kappa(j)) \), labeled by \( P^S_P(E^S_G, \vec{u}^S_G) \), has been inlined. Suppose that the target locations \( i \) and \( j \) belong to some procedure \( P^T_T \). To simplify this presentation, we assume that there is no nested inlining, so \( e^S \) belongs to \( P^T_T \) such that \( P^T_T \) is mapped to \( P^S_P \). The target procedure should contain unconditional assignment transitions \((i, a)\) and \((b, j)\) that correspond to the call to and return from procedure \( P^S_P \) on the source. Assume, \((i, a)\) is labeled by \([\vec{y}^T_g := EC^T_g]\) and \((b, j)\) is labeled by \([\vec{y}^T_g := ER^T_g]\), as depicted in Fig. 6.

Define a set of target locations \( L \subseteq C^T \) 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 \( P^S_P \). It is required that \( \alpha_l, l \in L \) does not depend on \( \vec{w}^S_G \), the variables whose references are passed to \( P^S_P \). However, we do allow the dependence on the corresponding formal parameters. This restriction comes from the fact that \( \vec{w}^S_G \) may change during the execution of \( P^S_P \), but, for efficiency reasons, the verification conditions from Section 6 work over variables of one procedure at a time.

Inlining Verification Conditions, presented in Fig. 6, are generated for each pair of target locations \((i, j)\) that correspond to the inlined call edge \((\kappa(i), \kappa(j))\). The call condition checks that the data abstraction holds at location \( a \) after the assignment on the target and the call on the source; the return condition checks that the data abstraction holds at location \( j \) after the corre-
sponding assignment and the return. Note that both of the target edges \((i, a)\) and \((b, j)\) contribute a verification condition to the set \(\mathcal{VC}\).

Example 3 In Fig. 7, program \(T\) is obtained from \(S\) after \(sqrt\) is inlined and the value of constant \(c\) is propagated. Notice our notation: we use capital letters to denote the source variables and procedure names; we use the lowercase equivalents in the target program. Let us apply the Translation Validation algorithm from Fig. 3 to prove the correct translation of \(S\) to \(T\).

1. The identity control abstraction \(\kappa\) maps the target procedure \(MAIN\) to source procedure \(main\).
2. The invariants \(\varphi^T_l : (c = 3)\) for \(l \in \{1, 2, 6, 7\}\) form the inductive network \(\mathcal{L}S\). We omit the set of the verification conditions that prove inductiveness of \(\mathcal{L}S\) since they are straightforward.
3. The following data abstraction is generated:

\[
\begin{align*}
\alpha_1 & : \text{true} \\
\alpha_2 & : (a = A) \land (d = D) \\
\alpha_3 & : (x = A \star 3) \land (a = A) \land (d = D) \\
\alpha_4 & : (x = A \star 3) \land (z_1 = Z) \land (u = U) \land (w = W) \land (a = A) \land (d = D) \\
\alpha_5 & : (x = A \star 3) \land (z_1 = Z) \land (u = U) \land (w = W) \land (a = A) \land (d = D) \\
\alpha_6 & : (z_0 = Z) \land (a = A) \land (d = D) \\
\alpha_7 & : (z_0 = Z) \land (a = A) \land (d = D) \\
\end{align*}
\]

Below we list the selected translation verification conditions from the set \(\mathcal{VC}\):
The corresponding target computation are performed. Note, that 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 we still satisfy the requirements of the correct translation as defined in Section 4. Consider a source computation $\sigma_S$ that contains $m$ recursive calls to $P^S_G$ and the corresponding target computation $\sigma_T$.

### Definition 8

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

Let $e^T = (i, r)$ be an unconditional assignment edge of the target procedure $P^T_g$ such that there exists a TRE candidate source edge $(\kappa(i), \kappa(t))$ labeled by $P^S_g(E^S_g(\bar{y}^T_g); \bar{z}^T_g)$, where the target procedure $P^T_g$ is mapped to the source procedure $P^S_g$. Under these conditions, we guess that TRE optimization occurred and generate TRE Verification Condition, presented in Fig. 8. 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). Note, that 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 we still satisfy the requirements of the correct translation as defined in Section 4. Consider a source computation $\sigma_S$ that contains $m$ recursive calls to $P^S_G$ and the corresponding target computation $\sigma_T$:

#### 7.2 Tail Recursion Elimination

![TRE Verification Conditions](image)

**Fig. 8.** TRE Verification Conditions: a tail recursive call is eliminated in procedure $P$. 

$\forall C_{\{4,5\}}: \alpha_4 \land \rho_{\{4,5\}} \rightarrow Cond_{\{4,5\}} \Leftrightarrow
\{(x = A * 3) \land (z_1 = Z) \land (u = U) \land (w = W) \land (a = A) \land (d = D)\} \land
\{(W > A * 3) \land (Z' = Z) \land (U' = U) \land (W' = W) \land (A' = A) \land (D' = D)\} \rightarrow (w > x)
\forall C_{\{6,7\}}: \alpha_6 \rightarrow \alpha_7 \land \alpha_7 \land (z_0 + d) = (Z + D) \Leftrightarrow
\{(z_0 = Z) \land (a = A) \land (d = D)\} \rightarrow \{(z_0 = Z) \land (a = A) \land (d = D)\} \land (z_0 + d) = (Z + D)

Inlining $\forall C_{\text{call}}: \alpha_2 \rightarrow \alpha_3[(a * 3) \rightarrow x]$; $\forall C_{\text{return}}: \alpha_5 \rightarrow \alpha_6[z_1 \rightarrow z_0] \Leftrightarrow
\forall C_{\text{call}}: \{(a = A) \land (d = D)\} \rightarrow \{(a * 3 = A * 3) \land (a = A) \land (d = D)\};
\forall C_{\text{return}}: \{(x = A * 3) \land (z_1 = Z) \land (u = U) \land (w = W) \land (a = A) \land (d = D)\} \rightarrow \{(z_1 = Z) \land (a = A) \land (d = D)\}

4. The generated conditions are valid so we conclude that $T$ is the correct translation of $S$. 

<table>
<thead>
<tr>
<th>$P^S_G$ (in: $x^S_G; z^S_G$)</th>
<th>$P^T_g$ (in: $x^T_g; z^T_g$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\kappa(i)$</td>
<td>$\alpha_i((x^S_G; z^S_G); y^T_g)$</td>
</tr>
<tr>
<td>$\kappa(t)$</td>
<td>$\alpha_i(y^S_g; y^T_g)$</td>
</tr>
<tr>
<td>$e^S$</td>
<td>$\alpha_i(z^S_G; z^T_g)$</td>
</tr>
</tbody>
</table>

$\forall C_{\text{call}}: \alpha_i(y^S_g; y^T_g) \land \phi^T_g(y^T_g) \land \phi^S_i(y^S_g) \rightarrow \alpha_i((E^S_g, z^S_G); EC^T_g(y^T_g))$
\[ \sigma_S = \ldots \]
\[ \langle \kappa(r), D_i^0 \rangle \rightarrow \ldots \rightarrow \langle \kappa(i), D_i^0 \rangle \xrightarrow{e_S} \]
\[ \langle \kappa(r), D_i^1 \rangle \rightarrow \ldots \rightarrow \langle \kappa(i), D_i^1 \rangle \xrightarrow{e_S} \]
\[ \ldots \]
\[ \langle \kappa(r), D_i^{m-1} \rangle \rightarrow \ldots \rightarrow \langle \kappa(i), D_i^{m-1} \rangle \xrightarrow{e_S} \]
\[ \langle \kappa(r), D_i^m \rangle \rightarrow \ldots \rightarrow \langle \kappa(t), D_i^m \rangle \xrightarrow{ret} \langle \kappa(t), D_i^1 \rangle \xrightarrow{ret} \langle \kappa(t), D_i^0 \rangle \]
\[ \ldots \]
\[ \sigma_T = \ldots \]
\[ \langle r, d_r^0 \rangle \rightarrow \ldots \rightarrow \langle i, d_r^0 \rangle \xrightarrow{e_T} \]
\[ \langle r, d_r^1 \rangle \rightarrow \ldots \rightarrow \langle i, d_r^1 \rangle \xrightarrow{e_T} \]
\[ \ldots \]
\[ \langle r, d_r^{m-1} \rangle \rightarrow \ldots \rightarrow \langle i, d_r^{m-1} \rangle \xrightarrow{e_T} \]
\[ \langle r, d_r^m \rangle \rightarrow \ldots \rightarrow \langle t, d_r^m \rangle \]

\[ D_i^k \] and \[ d_r^k \] 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 \( \alpha_t(\vec{z}_G; \vec{z}_T) \) holds. Validity of the verification conditions generated for all target edges that end in \( t \) prove that \( \alpha_t(D_i^m; d_r^m) \) holds: \( \alpha_t \) holds before we take the very first return transition in the source. Note that the only source variables effecting \( \alpha_t(\vec{z}_G; \vec{z}_T) \) are the formal parameters passed by reference that match the actual parameters used for the tail call. Therefore, popping the stack does not change \( \vec{z}_G \), 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 \( \alpha_S \) and \( \alpha_T \) that can be obtained by applying the observation function \( \mathcal{O} \) to the computations \( \sigma_S \) and \( \sigma_T \):

\[ \alpha_S = \ldots \]
\[ \mathcal{O}(\langle \kappa(r), D_i^0 \rangle) \rightarrow \ldots \rightarrow \mathcal{O}(\langle \kappa(i), D_i^0 \rangle) \rightarrow \top \]
\[ \mathcal{O}(\langle \kappa(r), D_i^1 \rangle) \rightarrow \ldots \rightarrow \mathcal{O}(\langle \kappa(i), D_i^1 \rangle) \rightarrow \top \]
\[ \ldots \]
\[ \mathcal{O}(\langle \kappa(r), D_i^{m-1} \rangle) \rightarrow \ldots \rightarrow \mathcal{O}(\langle \kappa(i), D_i^{m-1} \rangle) \rightarrow \top \]
\[ \mathcal{O}(\langle \kappa(r), D_i^m \rangle) \rightarrow \ldots \rightarrow \mathcal{O}(\langle \kappa(t), D_i^m \rangle) \rightarrow \top \ldots \]

\[ \alpha_T = \ldots \]
\[ \mathcal{O}(\langle r, d_r^0 \rangle) \rightarrow \ldots \rightarrow \mathcal{O}(\langle i, d_r^0 \rangle) \rightarrow \top \]
\[ \mathcal{O}(\langle r, d_r^1 \rangle) \rightarrow \ldots \rightarrow \mathcal{O}(\langle i, d_r^1 \rangle) \rightarrow \top \]
\[ \ldots \]
\[ \mathcal{O}(\langle r, d_r^{m-1} \rangle) \rightarrow \ldots \rightarrow \mathcal{O}(\langle i, d_r^{m-1} \rangle) \rightarrow \top \]
\[ \mathcal{O}(\langle r, d_r^m \rangle) \rightarrow \ldots \rightarrow \mathcal{O}(\langle t, d_r^m \rangle) \rightarrow \ldots \]

The verification conditions that we generate for each target edge ensure that for every target transition in \( \alpha_T \) there exists a corresponding source transition in \( \alpha_S \). Furthermore, the read/write transitions (and the associated data) match. Thus, the source observation \( \alpha_S \) can be obtained from the target observation \( \alpha_T \) by adding exactly \( m \) pairs \( \rightarrow \top \rightarrow \top \).
8 Conclusion and Future Work

In this work, we presented a framework for automatic translation validation of reactive programs in presence of interprocedural optimizations. Since all mentioned translation validation approaches 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 CVC Lite [2] as the back end validity checker. In addition, the framework has yet to be extended to incorporate more language features such as aliasing, 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 the transformation specification language that provides enough information for automatic translation validation. This idea is similar to [10]. However, since our approach is translation validation and we do not insist that all the transformations are provably correct for all the programs, we hope to be more flexible and capture a larger set of program transformations.

References

A States and Computations

For simplicity, we assume that all variables of a module range over the same domain \( D \) (say, the integers). We denote by \( \vec{d} = (d^1, \ldots, d^n) \) a tuple of \( D \)-values, which represents an interpretation (i.e., an assignment of values) of the module variables.

Definition 9 A state of module \( P \) is a pair \( \langle l; \vec{d} \rangle \) consisting of a location \( l \) and a data interpretation \( \vec{d} \).

Definition 10 A \((\xi, \zeta)\)-computation of module \( P \) is a maximal sequence of states and their labeled transitions:

\[
\sigma : \langle r; (\xi, \zeta, \vec{\tau}) \rangle \xrightarrow{\lambda_1} \langle l_1; \vec{d}_1 \rangle \xrightarrow{\lambda_2} \langle l_2; \vec{d}_2 \rangle \ldots
\]

The tuple \( \vec{\tau} \) denotes uninitialized values. At the first state of the computation, the location is \( r \), the entry location of \( P \); the values of input variables \( \vec{x} \) and \( \vec{z} \) are set to \( \xi \) and \( \zeta \), respectively, and the local variables \( \vec{w} \) are not initialized. Labels in the transitions are either names of edges in the program or the special label \( \text{ret} \). Each transition in a computation must be justified by one of the following cases:

- **Intra-procedural transition** \( \langle l; \vec{d} \rangle \xrightarrow{e} \langle l'; \vec{d}' \rangle \):
  - **Guarded Assignment**: There exists an edge \( e \) from node \( l \) to node \( l' \) in the program \( A \) (not necessarily in \( P_k \)) labeled by \( c \mapsto [\vec{u} := E(\vec{y})] \) such that \( \vec{d}' = c \vec{d} \) and \( \vec{d}' = (\vec{d} \text{ with } \vec{u} = E(\vec{d})) \), i.e. \( \vec{d}' \) is obtained from \( \vec{d} \) by replacing the values corresponding to the variables \( \vec{u} \) by \( E(\vec{d}) \).
  - **Read**: There exists an edge \( e \) in the program \( A \) from node \( l \) to node \( l' \) labeled by \( \text{read}(\vec{u}) \) such that \( \vec{d}'_v = \vec{d}_v \), where \( \vec{v} = \vec{y} \setminus \vec{u} \). \( \vec{d}'_v \) is obtained from \( \vec{d} \) by restricting it only to the values that correspond to the variables \( \vec{v} \). The values of all variables but the ones in \( \vec{u} \) are preserved by the read transition.
  - **Write**: There exists an edge \( e \) in the program \( A \) from node \( l \) to node \( l' \) labeled by \( \text{write}(\vec{u}) \). Since write instruction does not change the values of the variables, \( \vec{d}' = \vec{d} \).
- **Procedure call**: To justify transition \( \langle l; \vec{d} \rangle \xrightarrow{e} \langle r^k; (E(\vec{d}), \vec{d}_{\text{a}}, \vec{\tau}) \rangle \), there must exist a call edge \( e = (l, l') \) in the program \( A \) labeled by \( P_k(E(\vec{y}), \vec{u}) \), as depicted in Fig. 9. The location of the new state \( r^k \) is the first location in the called procedure \( P_k \). \( E(\vec{d}) \) and \( \vec{d}_{\text{a}} \) are the values of the input variables \( \vec{x}_k \) and \( \vec{z}_k \) on entry to \( P_k \). We assume that the working variables are uninitialized.
- **Procedure return**: Finally we consider transition \( \langle t^k; (\xi_k, \zeta_k, \eta_k) \rangle \xrightarrow{\text{ret}} \langle l'; \vec{d} \rangle \). To justify such a transition, there must exist a module \( P_k \) (the module from which we return), such that \( t^k \) is the terminal location of \( P_k \), and we should be able to identify a suffix of the current computation of the form

\[
\langle l; \vec{d} \rangle \xrightarrow{e} \langle r^k; (\xi_k, \zeta_k, \vec{\tau}) \rangle \xrightarrow{\xi_1} \ldots \xrightarrow{\xi_m} \langle t^k; (\xi_k', \zeta_k, \eta_k) \rangle \xrightarrow{\text{ret}} \langle l'; \vec{d} \rangle
\]

such that the segment \( \hat{\sigma} \) is balanced (has an equal number of calls and returns). We also require that \( e \) is a procedure call edge from node \( l \) to node \( l' \) labeled by \( P_k(E(\vec{y}), \vec{u}) \) and \( \vec{d}' = (\vec{d} \text{ with } \vec{u} = \zeta_k') \).

Computations of \( P_0 \) constitute the set of computations of a procedural program \( A \).
B From Programs to Transition Graphs

In this section, we describe how to construct the formal model of a program specified in a standard imperative languages such as C. We also show how to represent the basic language features like global variables and function calls. At present, the framework has yet to be extended to incorporate some language features with most noticeable omissions of dynamic memory allocation and exceptions.

B.1 Module Nodes as Cut-Points

A set of cut-points is a set of program locations $C$ such that:

- At least one location in each loop belongs to $C$.
- For every procedure, both procedure entry and exit belong to $C$.
- For every procedure call edge $(i, j)$, locations $i$ and $j$ belong to $C$.
- The locations right before and after each read/write operation belong to $C$.

The choice of cut-point set can be generalized not to require at least one cut-point per each loop but to ensure that the transitions between every pair of cut-points are computable [6].

Each procedure used in the program, whose implementation is given, is represented by a transition graph. We choose the set of cut-points of a procedure $P_k$ to be the set of nodes for the corresponding transition graph. If there exists a path $\pi$ from cut-point $i$ to cut-point $j$, which does not pass through any other cut-point, we add edge $(i, j)$ 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, $\text{pow}$ function in C), the call is modeled by uninterpreted functions.

B.2 Global Variables

The global variables are modeled using input parameters. Global variable $v$ is represented by variable $v_k$ in procedure $P_k$. Later in the presentation, we may drop index $k$ when it’s clear from the context. First, for each procedure $P_k$ except for the main module $P_0$, we compute the set of global variables $M$ that may be modified by the procedure or any of its children. We also compute the set of global variables $U$ whose value may be used by $P_k$ or its children such that $U \cap M = \emptyset$.

Second, we add variables of $M$ to the list of input parameters $\vec{z}_k$, which are passed by reference. We add variables of $U$ to the list of input parameters $\vec{x}_k$, which are passed by value. Third, we
modify each call to the procedure adding $M$ and $U$ to the argument list. Finally, we add all the global variables to the set of local variables of $P_0$ and move their initialization inside the scope of $P_0$.

B.3 Functions

A function call is modeled by a procedure call followed by an assignment:

$$n := f(\bar{y}, \bar{u}) \implies f(\bar{y}, (\bar{u}, n')); \quad n := n'$$

where $n'$ is a fresh variable. The following modifications should be applied to the module $f$. The list of input variables $\bar{z}_f$ should be extended with variable $\text{result}_f$, representing the return value of the function. Each edge $(i, j)$ labeled $\text{return } E(\bar{y}_f)$ should be replaced by an assignment edge from $i$ to $t_f$ (the exit location of the module $f$) labeled by $\text{result}_f := E(\bar{y}_f)$. 
