Generation of Inductive Assertion Network using Interprocedural Data Flow Analysis

Anna Zaks

March 21, 2006

Joint work with: Amir Pnueli
Introduction

• This is a preliminary result.

• Interprocedural Translation Validation as motivation.

• Constant propagation is one of the central optimizations.

• Use source invariants in order to strengthen our proof rules.

• Introduce an existing method for obtaining the precise solution for the constant propagation problem.

• Obtain the source invariants using the solution.
An application \( \mathcal{A} \) consists of \( m + 1 \) modules, where \( P_0 \) represents the main procedure:

\[
\begin{align*}
P_0() & \quad P_1(in : \overrightarrow{x}_1; \overrightarrow{z}_1) & \quad P_m(in : \overrightarrow{x}_m; \overrightarrow{z}_m) \\
& \quad & \quad \ldots
\end{align*}
\]

The variables of each module \( P_i \) are partitioned into \( \overrightarrow{y} = (\overrightarrow{x}; \overrightarrow{z}; \overrightarrow{w}) \), where:

- \( \overrightarrow{x} \) are the input parameters passed by value;
- \( \overrightarrow{z} \) are the input parameters passed by reference;
- \( \overrightarrow{w} \) denotes local(working) variables.
Instructions

Nodes of the graph are connected by directed edges labeled by instructions:

- A **Guarded assignment** is an instruction of the form $c \rightarrow [\overline{u} := \text{exp}(\overline{y})]$, where $c$ is a boolean expression over $\overline{y}$, where $\overline{u} \subseteq \overline{y}$.

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

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

From Programs to Transition Graphs

- 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 is 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 $\pi$.

- The global variables are efficiently modeled using input parameters.

- The representation of functions is straightforward.
Example of Constant Propagation: Source

main()

0

ym := 5

1

foo(xm, 5, ym, zm)

5

write(zm)

6

foo((xm + 6), 5, ym, zm)

7

foo(x, c, y, z)

2

(x ≥ 0) → [(x, z) := (x − 2, y)]

3

(x < 0) → [z := c]

4

(x = 0) → [y := x]

foo(x, c, y, z)

Generation of Inductive Assertion Network using Interprocedural Data Flow Analysis
Example of Constant Propagation: Target

**main()**

0

$ym := 5$

1

$foo(xm, 5, ym, zm)$

5

$write(5)$

6

$foo((xm + 6), 5, ym, zm)$

7

$foo(x, c; y, z)$

2

$(x \geq 0) \rightarrow [(x, z) := (x - 2, y)]$

3

$(x < 0) \rightarrow [z := c]$

$x = 0 \rightarrow [y := x]$

4

$foo(x, c, y, z)$
Let fictitious variables $\overrightarrow{X}$ and $\overrightarrow{Z}$ represent the values of the input variables $\overrightarrow{x}$ and $\overrightarrow{z}$ at the procedure entry.

An assertion network $N = \{\varphi_0, \ldots, \varphi_n\}$ associates an assertion $\varphi_l$ with each program location $l$:

- For each module $P_k$, the input predicate associated with the procedure entry location is denoted by $p_k(\overrightarrow{X}, \overrightarrow{Z}; \overrightarrow{x}, \overrightarrow{z})$.

- The assertion associated with the exit location is denoted by $q_k(\overrightarrow{X}, \overrightarrow{Z}; \overrightarrow{z})$. The output predicate $q_k$ is the procedure summary: it specifies the relation between the input and output values.

- The assertions at all other locations of the procedure $\varphi_l(\overrightarrow{Y})$ may depend on any of the variables.
We associate the following two conditions with a procedure call $P_k(\exp(\overrightarrow{y}), \overrightarrow{u})$:

- $\mathcal{VC}_{call} : \varphi_i(\overrightarrow{Y}) \rightarrow p_k(\exp(\overrightarrow{y}), \overrightarrow{u}; \exp(\overrightarrow{y}), \overrightarrow{u})$
- $\mathcal{VC}_{return} : \varphi_i(\overrightarrow{Y}) \land q_k(\exp(\overrightarrow{y}), \overrightarrow{u}; \overrightarrow{z}_k) \rightarrow \varphi_j(\overrightarrow{Y})[\overrightarrow{u} \mapsto \overrightarrow{z}_k]$
Inductive Assertion Network

- An assertion network $\mathcal{N}$ for a program $\mathcal{A}$ is said to be inductive if all the verification conditions for all edges in $\mathcal{A}$ are valid.

- Every inductive network is invariant.

Our goal is to construct an inductive assertion network for the source program that would be strong enough to prove translation in presence of context-sensitive constant propagation.

Since the interprocedural constant propagation is not a trivial problem, we compute the inductive network utilizing the existing dataflow analysis.
First Attempt

On the first thought, any precise solution to the interprocedural constant propagation problem should suffice: $\varphi_l$ should be extended with conjunct $x = 5$ if $x$ always evaluates to constant 5 at location $l$.

However, the resulting assertion network $\mathcal{N}$ may not be inductive.
Back to our Example

```
main()
{
    ym := 5
}
foo(xm, 5, ym, zm)
{
    zm := 5
}
write(zm)
{
    zm := 5
}
foo((xm + 6), 5, ym, zm)

foo(x, c; y, z)
{
    c := 5
}
(x ≥ 0) → [(x, z) := (x - 2, y)]
(x < 0) → [z := c]
(x = 0) → [y := x]
foo(x, c, y, z)
{
    c := 5
}
```

Generation of Inductive Assertion Network using Interprocedural Data Flow Analysis
Solution

As our interprocedural dataflow analysis algorithm, we are going to use the one presented in:

- Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation by M. Sagiv, T. Reps, and S. Horwitz.

- Precise Interprocedural Dataflow Analysis via Graph Reachability by M. Sagiv, T. Reps, and S. Horwitz.

- Two Approaches to Interprocedural Dataflow Analysis (Functional Approach) by M. Sharir and A. Pnueli.

The interprocedural dataflow analysis algorithm not only provides a solution to the problem, 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.
Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation

- Solves copy constant propagation and linear constant propagation (both distributive problems);

- Produces precise (”meet-over-all-valid-paths” solution) results for both recursive and non-recursive programs;

- The dynamic-programming algorithm solves these constant propagation problems in polynomial time $O(E \times MaxVisible^3)$;

- Uses a compact representation of distributive dataflow functions (”environment transformers”).
Distributive Constant Propagation Problems

- In copy constant propagation, a variable \( x \) is discovered to be \( 10 \) only if
  - \( x := 10 \)
  - \( y := 10; \ x := y \)

- In linear constant propagation, a variable \( x \) is discovered to be \( 10 \) only if
  - \( x := 10 \)
  - \( y := 3; \ x := 2 \times y + 4 \)
Environment Transformer

- Let $V$ be a finite set of program variables. Let $L = \mathbb{Z}^\top$.
- 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.

For an environment $env \in Env(V, L)$, if $env(v) \in \mathbb{Z}$ then the variable $v$ has a known constant value in the environment $env$; the value $\bot$ denotes non constant, and $\top$ denotes an unknown value.

A dataflow problem is specified by annotating each edge of the supergraph of $\mathcal{A}$ with an environment transformer. We are interested in meet over all valid paths solution.
Environment Transformers for Constant Propagation Problems

- **Linear constant propagation:**

  \[
  T : \text{Env}(V, L) \mapsto \text{Env}(V, L)
  \]

  \[
  x := c \quad \lambda_{\text{env.env}}[x \to c]
  \]

  \[
  x := c_1 \times y + c_2 \quad \lambda_{\text{env.env}}[x \to c_1 \times \text{env}(y) + c_2]
  \]

  \[
  x := y + z \quad \lambda_{\text{env.env}}[x \to \bot]
  \]

- The last transformer is a safe approximation; the exact transformer \(\lambda_{\text{env.env}}[x \to \text{env}(y) + \text{env}(z)]\) cannot be used in this framework because it is not distributive.

- Consider two environments: \(\text{env}_1 : [y \to 3; z \to 2]\) and \(\text{env}_2 : [y \to 4; z \to 1]\)

- **Distributivity:**

  \(T(\text{env}_1 \cap \text{env}_2) = T(\text{env}_1) \cap T(\text{env}_2)\)

- \(T_{x+y}(\text{env}_1 \cap \text{env}_2) = T_{x+y}([y \to \bot; z \to \bot]) = [x \to \bot + \bot] = [x \to \bot]\)

- \(T_{x+y}(\text{env}_1) \cap T_{x+y}(\text{env}_2) = [x \to 3 + 2] \cap [x \to 4 + 1] = [x \to 5]\)
Pointwise Representation of Environment Transformers

Every distribution transformer $T : Env(V, L) \mapsto Env(V, L)$ can be represented using a set of functions:

$$F^T = \{ f_{v,v'} \mid v, v' \in V \cup \{ \Lambda \} \}, \text{ each of type } L \mapsto L.$$

- 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 $f_{v,v'} = \lambda.l.\top$.

- Function $f_{\Lambda,v'}$ 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:

$$T(env)(v') = f_{\Lambda,v'} \sqcap (\sqcap_{v \in V} f_{v,v'}((env)(v))).$$
A Dynamic Programming Algorithm

Two phase algorithm:

- Compute the dataflow solution at the nodes of a procedure as a function of the initial values at the procedure entry - path functions.
- Compute the dataflow values at every point using the path functions.
Phase One: environment transformers

Below is the list of environment transformers computed for procedure $foo$. We omit all the functions that evaluate to top $f_{(v,v')} = \lambda l. \top$.

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

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

$$F_{(2,4)} = \{ f_{c,c} = \lambda l.l, f_{c,z} = \lambda l.l, f_{y,z} = \lambda l.l \}$$
Computation of Invariants using the Env Transformers

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

- We ignore all functions of the form $f(v,v') = \lambda l. \top$.

- For each variable $v'$ that is not set to $\bot$ by $f(\Lambda, v') \in F_{(r_k,l)}$ we add the following conjunct to $\varphi_l$:

$$\bigvee_{f_{v,v'} \in F_{(r_k,l)}} v' = f_{v,v'}(V)$$

Note that we use disjunction to model the effect of the meet operator.

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

The list of environment transformers computed for procedure \textit{foo}:

\[
\begin{align*}
F_{(2,2)} &= \{ f_{\chi,\chi} = \lambda l.\, l, f_{c,c} = \lambda l.\, l, f_{y,y} = \lambda l.\, l, f_{z,z} = \lambda l.\, l \} \\
F_{(2,3)} &= \{ f_{c,c} = \lambda l.\, l, f_{y,y} = \lambda l.\, l, f_{y,z} = \lambda l.\, l \} \\
F_{(2,4)} &= \{ f_{c,c} = \lambda l.\, l, f_{c,z} = \lambda l.\, l, f_{y,z} = \lambda l.\, l \}
\end{align*}
\]
For example, let’s show that the return verification condition for call edge \((1, 5)\) of our example holds.

\[
\mathcal{VC}_{ret}: \quad \varphi_1 \land \varphi_4[(C, Y) \mapsto (5, y_m)] \quad \land \quad y_m = 5 \land 5 \land (z = 5 \lor z = y_m) \land c = 5 \quad \rightarrow \quad \varphi_5[z_m \mapsto z] \quad \iff \quad z = 5
\]
Questions?