Transformer

Yu-Fang Chen[1], Chiao Hsieh[1,2], Ming-Hsien Tsai[1], and Bow-Yaw Wang[1]

[1] Institute of Information Science, Academia Sinica, Taiwan
[2] Graduate Institute of Electrical Engineering, National Taiwan University, Taiwan

Abstract.

1 Introduction

2 Preliminaries

Let $\texttt{Vars}$ denote the set of *program variables* and $\texttt{Vars}' = \{\texttt{x}' : \texttt{x} \in \texttt{Vars}\}$. We consider a variant of the WHILE language []:

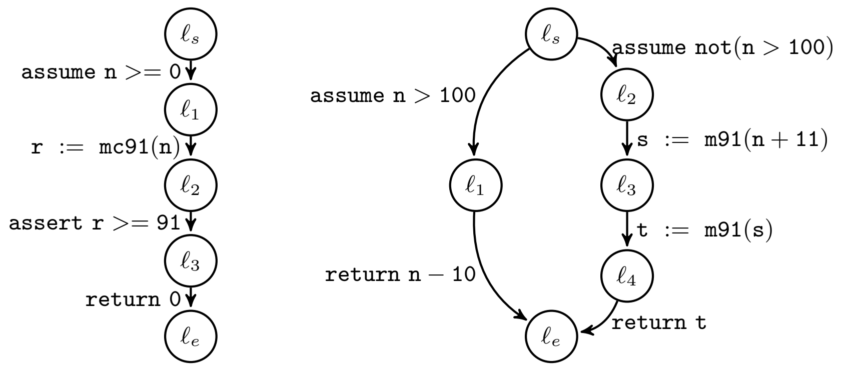
$$
\begin{array}{rlll}
\texttt{Expression} \ni E & ::= & \texttt{x} & \texttt{x} \in \texttt{Vars} \\
& \mid & \texttt{false} \mid \texttt{true} \mid \texttt{0} \mid \texttt{1} \mid \ldots & \text{constant} \\
& \mid & E \odot E & \odot \in \{+, -, =, >, \texttt{and}\} \\
& \mid & \texttt{not}\ E & \\
& \mid & \texttt{f}(E, E, \ldots, E) & \text{function invocation} \\
\texttt{Command} \ni C & ::= & \texttt{x}, \texttt{x}, \ldots, \texttt{x} := E, E, \ldots, E & \text{assignment} \\
& \mid & \texttt{assume}\ E & \text{assumption} \\
& \mid & \texttt{assert}\ E & \text{assertion} \\
& \mid & \texttt{return}\ E, E, \ldots, E & \text{function return}
\end{array}
$$

Note that simultaneous assignments are allowed in our language. To execute a simultaneous assignment, all expressions on the right hand side are first evaluated and then assign to respective variables. The assumption command excludes all computation not satisfying the given expression. The assert command terminates computation abnormally if the given expression is not satisfied. For instance, the following command always terminates normally:

$$\texttt{assume false};\quad \texttt{assert false}$$

no computation means terminate normally? Our simple language also allows functions to return multiple values. Together with simultaneous assignments, functions can update several variables at once.

A function is represented as a *control flow graph* $G = \langle V, E \rangle$ where the nodes in V are *program locations*, and each edge (ℓ, ℓ') in $E \subseteq V \times V$ is labeled by a command denoted by $\text{cmd}(\ell, \ell')$. We assume that the program locations ℓ_s and ℓ_e denote the entry and exit of each function. Moreover, the special $\texttt{main}()$ function specifies where a program starts. Figure 1 shows control flow graphs for the McCarthy 91 program. The $\texttt{main}()$ function assumes the variable $\texttt{n}$ is non-negative. It then checks `mc91(n)` is no less than 90 (Figure 1a). The `mc91(n)` function branches on whether the variable `n` is greater than 100. If so, `mc91(n)` returns $\mathtt{n}-10$. Otherwise, `mc91(n)` returns `mc91(mc91(n + 11))` (Figure 1b). Observe that the `assume` command models a conditional branch in the figure. Loops can be modeled similarly.



(a) `main()` (b) `mc91(n)`

Fig. 1: McCarthy 91

definitions of accessible variables, program states Let $G = \langle V, E\rangle$ be a control flow graph. An *inductive invariant* $\Pi(G, I_0) = \{I_\ell : \ell \in V\}$ for G from I_0 is a set of first-order logic formulae such that $I_{\ell_s} = I_0$, and for every $(\ell, \ell') \in E$

$$I_\ell \wedge \tau_{\mathrm{cmd}(\ell,\ell')} \implies I'_{\ell'}$$

where I' is the formula obtained by replacing every $\mathtt{x} \in \mathtt{Vars}$ in I with $\mathtt{x}' \in \mathtt{Vars}'$, and $\tau_{\mathrm{cmd}(e)}$ specifies the semantics of the command cmd(e). An inductive invariant $\Pi(G, I_0)$ is an over-approximation to the computation of G from I_0. More precisely, assume that the function G starts from a state satisfying I_0. For every program location ℓ, G must arrive in a state satisfying I_ℓ when the computation reaches ℓ. Observe that an inductive invariant $\Pi(G, I_0)$ establishes the weak correctness for the Hoare triple $\{I_0\}G\{I_{\ell_e}\}$.

A *program analyzer* accepts programs as inputs and checks if all assertions (specified by the `assert` command) are satisfied. One way to implement program analyzers is to compute inductive invariants.

Proposition 1. *Let $G = \langle V, E\rangle$ be a control flow graph and $\Pi(G, \mathtt{true})$ an inductive invariant for G from* `true`*. If $\models I_\ell \implies B_\ell$ for every edge $(\ell, \ell') \in E$ with cmd$(\ell, \ell') =$ `assert`(B_ℓ), then all assertions in G are satisfied.*

A program analyzer checks assertions by computing inductive invariants is called an *inductive* program analyzer. Note that an inductive program analyzer need not give any information when an assertion fails. Indeed, most inductive program analyzers simply report false positives when inductive invariants are too coarse. A *program checker* is a program analyzer that returns an error trace when an assertion fails; an *error trace* is a sequence of variable valuations from the program entry to the failed assertion. Rather than reporting false positives, program checkers have to return error traces to witness failed assertions. Producing error traces (especially for recursive programs) complicates analysis algorithms. We hence consider a subclass of program checker. A *recursion-free inductive program checker* is a program checker that checks recursion-free programs by computing inductive invariants, and reports an error trace when an assertion fails. Several recursion-free inductive program checkers are available such as CPACHECKER, BLAST, SLAM more?. Our goal is to check recursive programs by using a recursion-free inductive program checker as a black box.

3 Overview

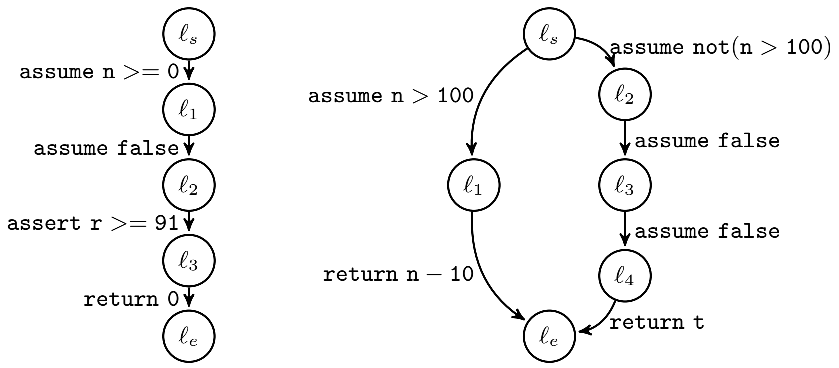
Let BASICCHECKER denote a recursion-free inductive program checker and $G = \langle V, E \rangle$ a control flow graph. Since non-recursive functions can be replaced by their control flow graphs after proper variable renaming, we assume that G only contains the `main`() and recursive functions. If G does not contain recursive functions, BASICCHECKER is able to check G by computing inductive invariants.

When G contains recursive functions, we transform G into two recursion-free programs $\underline{G}$ and $\overline{G}$. The program $\underline{G}$ under-approximates the computation of G. That is, every computation of $\underline{G}$ is also a computation of G. If BASICCHECKER finds an error trace in $\underline{G}$, our algorithm terminates and reports the error trace. The program $\overline{G}$ on the other hand over-approximates the computation of G; every computation of G is also a computation of $\overline{G}$. If BASICCHECKER proves that all assertions in $\overline{G}$ are satisfied, our algorithm terminates and reports that all assertions in G are satisfied. If BASICCHECKER fails to prove assertions in $\overline{G}$, our algorithm refines the approximations by unwinding recursive functions and reiterates (Algorithm 1).

```
**Input**: $G = \langle V, E \rangle$ : a control flow graph
$k \leftarrow 0$;
$G_0 \leftarrow G$;
**repeat**
    **if** BASICCHECKER $(\underline{G}_k) = ErrorTrace(\tau)$ **then**
        // find an error trace in the under-approximation $\underline{G}_k$
        **return** $ErrorTrace(\tau)$;
    **else if** BASICCHECKER $(\overline{G}_k) = Pass$ **then**
        // prove all assertions in the over-approximation $\overline{G}_k$
        **return** $Pass$;
    **else**
        $G_{k+1} \leftarrow$ unwind $G_k$;
        $k \leftarrow k + 1$;
**until** *forever* ;
```

Algorithm 1: Overview

To see how to under approximate computation, consider a control flow graph G_k with recursive functions $f_0(\mathbf{x}_0), f_1(\mathbf{x}_1), \ldots, f_m(\mathbf{x}_m)$. The under-approximation $\underline{G}_k$ is obtained by substituting the command `assume false` for every command with recursive function calls. The substitution effectively blocks all recursive invocations. Note that $\underline{G}_k$ is recursion-free. BASICCHECKER is able to check the under-approximation $\underline{G}_k$.



(a) $\underline{\texttt{main}}$() (b) $\underline{\texttt{mc91}}$(n)

Fig. 2: Under Approximation of McCarthy 91

4 Unwinding Recursion

4.1 Unwinding

4.2 Bug Catching

4.3 Property Proving

5 Experiments

6 Conclusion

References