# SmaLLVM Analyzer

Kihong Heo

## 1 Objective

This article describes the design of a static analyzer to detect potential division-by-zero errors for the SmaLLVM language.

## 2 Syntax

A program is a control-flow graph $G = \langle \mathbb{C}, \rightarrow \rangle$ where $\mathbb{C}$ is a set of control-flow nodes and $\rightarrow \subseteq \mathbb{C} \times \mathbb{C}$ is the control-flow relation of the program. Each node is associated with a command $c \in C$. We use node $n \in \mathbb{C}$ interchangeably with the command $c$ of $n$. Commands and expressions are defined as follows:

$$
\begin{array}{rcll}
E & ::= & n & \text{integer} \\
& | & x & \text{variable} \\
& | & E \oplus E & \text{arithmetic operation} \\
& | & E \otimes E & \text{comparison operation} \\
\oplus & ::= & + \mid - \mid \times \mid / & \\
\otimes & ::= & < \mid \leq \mid > \mid \geq \mid == \mid != & \\
C & ::= & \mathit{Atomic} & \text{atomic statement} \\
& | & \mathit{Cond} & \text{conditional branch} \\
& | & \mathit{Phi} & \text{phi nodes} \\
\mathit{Atomic} & ::= & x := E & \text{assignment} \\
& | & x := \mathtt{source}() & \text{input} \\
& | & \mathtt{print}(x) & \text{print} \\
& | & \mathtt{goto} & \text{unconditional jump} \\
\mathit{Cond} & ::= & \mathtt{tbr}\ E & \text{true branch} \\
& | & \mathtt{fbr}\ E & \text{false branch} \\
\mathit{Phi} & ::= & \{x_i := \phi\ [E_{i1}, \ldots, E_{ik}]\}_{i=0}^{n} & \text{list of phi nodes}
\end{array}
$$

Notice that the syntax definition elides labels in LLVM commands such as `L` in `goto L`. In our setting, labels are translated to control-flow edges.

## 3 Abstract Domains

$$\begin{array}{rcccl} & & \mathbb{D}^\sharp & = & \mathbb{C} \to \mathbb{M}^\sharp \\ m^\sharp & \in & \mathbb{M}^\sharp & = & \mathbb{X} \to \mathbb{Z}^\sharp \\ n^\sharp & \in & \mathbb{Z}^\sharp & = & \{\bot, \mathsf{Neg}, \mathsf{Zero}, \mathsf{Pos}, \top\} \\ x & \in & \mathbb{X} & & \text{variable} \end{array}$$

## 4 Abstract Semantics

The abstract semantics of a program is characterized by the least fixed point of the following function $F \in (\mathbb{C} \to \mathbb{M}^\sharp) \to (\mathbb{C} \to \mathbb{M}^\sharp)$:

$$F(X) = \lambda c \in \mathbb{C}. f_c\big(\bigsqcup_{c' \to c} (X(c'))\big)$$

where $f_c \in \mathbb{M}^\sharp \to \mathbb{M}^\sharp$ is defined as follows:

$$f_c(m^\sharp) = \begin{cases} m^\sharp\{x \mapsto \mathit{eval}^\sharp_E(m^\sharp)\} & c = \text{“}x := E\text{”} \\ m^\sharp\{x \mapsto \top\} & c = \text{“}x := \texttt{input}()\text{”} \\ m^\sharp & c = \text{“}\texttt{print}(x)\text{”} \\ m^\sharp & c = \text{“}\texttt{goto}\text{”} \\ \mathit{filter}^\sharp_E(m^\sharp) & c = \text{“}\texttt{tbr}\ E\text{”} \\ \mathit{filter}^\sharp_{\neg E}(m^\sharp) & c = \text{“}\texttt{fbr}\ E\text{”} \\ \bigsqcup_i \{m^\sharp\{x_i \mapsto \bigsqcup_j \{\mathit{eval}^\sharp_{E_{ij}}(m^\sharp)\}\} & c = \text{“}\{x_i := \phi\ [E_{i1}, \ldots, E_{ik}]\}_{i=0}^n\text{”} \end{cases}$$

Notice that the semantics of phi nodes is defined with the input memory $m^\sharp$ and the updated memory by one phi node does not affect to the semantics of the other phi nodes. For more detailed explanation, see the phi node semantics of LLVM[1].

The abstract semantic function of expression $\mathit{eval}^\sharp_E : \mathbb{M}^\sharp \to \mathbb{Z}^\sharp$ is defined as follows:

$$\begin{array}{rcl} \mathit{eval}^\sharp_n(m^\sharp) & = & \text{sign of } n \\ \mathit{eval}^\sharp_x(m^\sharp) & = & m^\sharp(x) \\ \mathit{eval}^\sharp_{E_1 \oplus E_2}(m^\sharp) & = & \mathit{eval}^\sharp_{E_1}(m^\sharp) \oplus^\sharp \mathit{eval}^\sharp_{E_2}(m^\sharp) \\ \mathit{eval}^\sharp_{E_1 \otimes E_2}(m^\sharp) & = & \mathit{eval}^\sharp_{E_1}(m^\sharp) \otimes^\sharp \mathit{eval}^\sharp_{E_2}(m^\sharp) \end{array}$$

The abstract semantic functions for $\oplus^\sharp$ and $\otimes^\sharp$ are the best (sound yet most precise) abstractions of the concrete counterparts in the sign domains.

The abstract filter function $\mathit{filter}^\sharp_E(m^\sharp) : \mathbb{M}^\sharp \to \mathbb{M}^\sharp$ is also the best abstraction of the concrete counterpart.

[1] `https://llvm.org/docs/LangRef.html#id312`