Basic Static Analysis with ℤahlen: a Simple Imperative Programming Language

Shinyoung Kim

1 Introduction

TBC

2 Syntax

For the full specification of the syntax, please refer to `grammar/zahlen.ebnf`.

3 Semantics

In this section, we discuss the denotational and transitional semantics of ℤahlen, which is required to write a sound static analyzer.

3.1 Denotational Semantics

We define $[\![\textbf{stmt}]\!]\sigma$ as the denotational semantics of statement **stmt** under state $\sigma \in \Sigma$. Note that $\Sigma_\perp = \Sigma \cup \perp$ indicates the set of states augmented with $\perp$ representating non-termination.

The denotational semantics for ℤahlen statements are defined as follows:

$$[\![v = e]\!]\sigma = [\sigma \mid v : [\![e]\!]\sigma]$$

$$[\![\textbf{skip}]\!]\sigma = \sigma$$

$$[\![\textbf{ifelse}(e, s_1, s_2)]\!]\sigma = \begin{cases} [\![s_1]\!]\sigma, & \text{if } [\![e]\!]\sigma \\ [\![s_2]\!]\sigma, & \text{otherwise} \end{cases}$$

$$[\![s_1; s_2]\!]\sigma = \begin{cases} ([\![s_2]\!])([\![s_1]\!]\sigma), & \text{if } [\![s_1]\!]\sigma \neq \bot \\ \bot, & \text{otherwise} \end{cases}$$

$$[\![\textbf{while}(e)\{s\}]\!]\sigma = \text{lfp}_\sigma(\text{if } [\![e]\!]\sigma \text{ then } [\![s]\!]\sigma \text{ else } \sigma) \text{ where } \text{lfp}_\sigma : (\Sigma_\bot \to \Sigma_\bot) \to \Sigma_\bot$$

3.1.1 Why goto statements are problematic

My initial decision was to evade defining loops(primarily **while** loops) because the denotational semantics for them require a fixpoint-semantics declaration, which would be non-trivial to mechanically analyze. Instead, loops would be purely a syntactic construct and were expected to be rewritten to a combination of **ifelse** and **goto** statements. Much like what you would see in assembly, the **goto** l statement unconditionally jumps to a statement with label l.

The label-statement associative map $\mathcal{S}$ is a set of 2-tuples $\{(l_1, s_1), (l_2, s_2), ..., (l_n, s_n)\}, l_N \in L, s_n \in S$ where l_n, s_n are the statement labels and the associated statements respectively. Then we can define the map lookup function $\eta : L \to S^*$ where $S^* = S \cup \{\textbf{term}\}$:

$$\eta(l) = \begin{cases} s, & \text{if}(l, s) \in \mathcal{S} \\ \textbf{term}, & \text{otherwise} \end{cases}$$

which then seemingly allows us to define the semantics for defining statement labels and the **goto** statement:

$$[\![l : s]\!]\sigma = \mathcal{S} \leftarrow \mathcal{S} \cup \{(l, s)\}; [\![s]\!]\sigma$$

$$[\![\textbf{goto } l]\!]\sigma = [\![\eta(l)]\!]\sigma$$

The semantics at a glance appear to work out, but there is one critical problem: forward jumps. Suppose we have the following code:

```
n = 5;
sum = 0;
loop: ifelse(n == 0, goto end, skip);  # sum from 1 to 5
sum = sum + n;
goto loop
end: skip
```

Here the **goto** end statement is referencing the statement label end which hasn't yet been collected per the semantics of label designation, leading to $\mathcal{S} = \varnothing$ when evaluating the **ifelse** statement. This will lead to the program terminating which can't be seen as wrong per se, but poses grave restrictions on what procedures can be expressed.

The treatment of jump statements within denotational semantics are normally handled with *continuations*, which pass along not only the program state σ, but also a sequence of statements to be executed after the execution of the current statement. The executing statement is free to modify the continuation which in turn modifies the control flow.

I chose not to install the **goto** statement within Zahlen.

3.1.2 Transitional Semantics

3.1.3 ordered rule tex

$$\text{Modus Ponens} \quad \frac{A \quad A \implies B}{B} \qquad \text{Modus Tollens} \quad \frac{\neg B \quad A \implies B}{\neg A}$$

$$Out_C \quad \frac{v = [\![e]\!]}{\langle c!e.P, \rho\rangle \xrightarrow{e!v} \langle P, \rho\rangle} \qquad Inp_Q \quad \frac{r \notin qv(x?q.P)}{\langle c?q.P, \rho\rangle \xrightarrow{c?r} \langle P\{r/q\}, \rho\rangle}$$

3.2 typesetting commands

Command

Command

Command

Command

An associative array $A[K, V]$ is a 3-tuple $(K, V, \mathcal{D})$ where K is a set of keys, V a set of values, and D a set of ordered pairs (k, v) where v is the value associated to key k. Then the functions $\mathsf{lookup}(A, k)$ and $\mathsf{insert}(A, k, v)$ can be defined as the following:

$$[\![\mathsf{lookup}(A, k)]\!]\sigma = a$$

4 Abstract Interpretation

The concrete semantics declared through denotational semantics will be used for value abstraction. In ℤahlen static analysis, array indexing errors are detected through the denotational-interval semantics pair, whereas reachability analysis would use the transitional and program label-wise abstraction pair.

5 Analysis Coverage

In this chapter we describe the specific analysis goals the static analyzer achieves.

5.1 Out-of-bounds Array Indexing Error

Zahlen's static analyzer will detect out-of-bounds array indexing errors. The analyzer guarantees soundness, meaning the analysis result being positive for index errors will always indicate the error exists within the code.

Zahlen uses the interval abstract domain defined on top of the concrete semantics, which approximates the valid indexing range for defined multidimensional arrays and any array access operations.

5.1.1 Abstract Domain

An integer interval is defined as a pair of integers which denote all integers that satisfy the following:

$$[a, b] = \{a \leq n \leq b, n \in \mathbb{Z}\}$$

Interval arithmetic is defined as the following:

$$[a, b] + [c, d] = [a + c, b + d]$$
$$[a, b] - [c, d] = [a - c, b - d]$$
$$[a, b] \times [c, d] = [\min(ab, ad, bc, bd), \max(ab, ad, bc, bd)]$$

We now define the abstraction function $\alpha : C \to A$ along the concrete semantics of Zahlen:

content...