Creation of a Static Analysis Algorithm Using Ad Hoc Programming Languages

D. Khalansky, A. Lazdin, I. Muromtsev

ITMO University
Outline

1. Languages in language research
2. Our method
3. Case: value range analysis
4. Language
   - Type theory
   - Arithmetics
     - Randomness
   - Imperative order
     - Naive implementation
     - More robust implementation
   - Conditionals
   - Loops
5. Further extensions
6. Conclusion
```c
#include <stdlib.h>
#include <string.h>

struct vec { size_t n; size_t el; void *els; };

void remove_eq(struct vec *v, const void *el) {
    for (size_t i = 0; i < v->n; ++i) {
        if (memcmp((const char *)v->els + v->el * i, el, v->el))
            continue;
        memmove((char *)v->els + v->el * i,
                 (const char *)v->els + v->el * (i + 1),
                 v->n--; i -- 1);
        --i;
    }
}
```
Decomposition of C code

**Structs**  Just data with automatic accessors; can be replaced with bitwise operations on in-memory representations;

**Flow control**  In case of `continue`, `break`, and `goto` which doesn’t go backwards can be replaced with more verbose conditional statements;

**Multiple types of numbers**  Specific cases of a more general notion of a number modulo some power of two;

**Pointer arithmetics**  Many mechanisms applicable to the normal arithmetics can be used for pointers as well.

And so on. Evidently, one could write the same programs with a really small set of constructs.
Example of Lisp code

```
(print
  ((lambda (n)
     (let ((cont #f))
       (let ((m (call/cc (lambda (k)
                          (set! cont k)
                          (cons 1 n)))))
         (if (> (cdr m) 0)
           (cont (cons (* (car m) (cdr m))
                    (− (cdr m) 1)))
           (car m)))))
  6))
```
Our method

1. Decide which class of data manipulation is of interest;
2. Create a type system which is capable solely of representing this precise data manipulation;
3. Determine the sufficient basis of operations which can be performed on the data and formalize the algorithm for them;
4. Extend the language with new constructs as needed, slightly adapting the algorithm.
Value range analysis

\[ x \leftarrow \text{rand()} \mod 16 \]

\[ x \in [0; 15] \]

\[ x < 8 \]

\[ x \in [0; 7] \]

\[ x \leftarrow x + 8 \]

\[ x \in [8; 15] \]
Type theory

Infinite numbers
- Have range $[0; +\infty)$;
- Support $[+], [-], [\times], [/], [<], [=]$;
- Type of integer is determined at time of analysis.

Modular numbers with parameter $n$
- Have range $[0; 2^n)$;
- Support $[+], [-], [\times], [/], [<], [=], [\land], [\lor], [\sim]$;
- At risk of overflow.

Exist conversions between the two kinds.
Arithmetics

- $a + b$, $a - b$, $a \times b$, $a / b$, $a < b$, $a == b$ — don’t really need an introduction;
- $a \& b$ — bitwise AND;
- $a \mid b$ — bitwise OR;
- $\sim a$ — bitwise negation;
- $\text{inf } a$ — conversion to a natural number;
- $a \text{ bits } N$ — $a$ modulo $2^N$. 
Expressions like \(3 + (0xF8 \land 0xA) \times 4\) are completely deterministic, don’t allow us to simulate user input.

**Format**

\[ \text{rand bits } N \rightarrow \text{an arbitrary value in } [0; 2^N). \]
Tree of arithmetic expressions

```
+  
/   
×   ×  
/     
1 rand mod 2^4 3
/     
rand mod 2^4 0b1110
```
Chaining of assignments

Chaining

Chaining is ordered execution of statements, with statements commonly separated by semicolons:

\[ s_1; s_2; \ldots; s_n \]

Assignment statement

Setting the value pointed to by the identifier \( v \) to the result of evaluation of expression \( e \):

\[ v \leftarrow e \]
Chaining of assignments: naive version

\[
\@x = \text{rand bits 4} \quad x \in [0; 15]
\]

\[
@y = @x + 2 \quad y \in [0 + 2; 15 + 2] = [2; 17]
\]

\[
@z = @x + 5 \quad z \in [0 + 5; 15 + 5] = [5; 20]
\]

\[
@m = @z - @y \quad m \in [5 - 17; 20 - 2] = [-12; 18]
\]
Chaining of assignments: more robust version

\[
\begin{align*}
\@x &= \text{rand bits 4} \\
\; \\
\@y &= \@x + 2 \\
\; \\
\@z &= \@x + 5 \\
\; \\
\@m &= \@z - \@y
\end{align*}
\]

\[
\begin{align*}
x &= \text{rand}_1 \text{ bits 4} \\
\; \\
y &= \text{rand}_1 \text{ bits } 4 + 2 \\
\; \\
z &= \text{rand}_1 \text{ bits } 4 + 5 \\
\; \\
m &= \text{rand}_1 \text{ bits } 4 + 5 - (\text{rand}_1 \text{ bits } 4 + 2) = 3
\end{align*}
\]
Conditional expressions

\[ e_1 \neq 0 \quad \text{if} \quad e_1 = 0 \]

\[ x \leftarrow e_2 \quad \text{if} \quad e_1 \neq 0 \]

\[ x \leftarrow e_3 \quad \text{if} \quad e_1 = 0 \]

\[ x = (e_1 \neq 0) \cdot e_2 + (e_1 = 0) \cdot e_3 \]
for $v$ to $e$ do $s$ done, $e \in [n;m]$

\[
\begin{aligned}
&\{ n \} \\
&\quad s; s; s; \ldots; s; \\
&\quad \text{if } n < e \text{ then } s \text{ fi} \\
&\quad \text{if } n + 1 < e \text{ then } s \text{ fi} \\
&\quad \vdots \\
&\quad \text{if } m - 1 < e \text{ then } s \text{ fi} \\
\end{aligned}
\]
Further extensions

- Pure functions: they are just operations on numbers, and their range analysis can be pre-compiled in a modular fashion;
- More types of numbers;
- Probabilistic model: determine not only the possibility of a certain execution path but its probability as well;
- Complex structures based on bitwise arithmetics;
- More complex loop handling with finding repeating states of interconnected variables in a loop.
Conclusion

- The algorithm we’ve developed can be easily checked due to modular approach taken during its development;
- The algorithm can easily be extended to account for more complex language features;
- Development has been a relatively simple task of creation, not implementation.