Generation of In-Bounds Inputs for Arrays in Memory-Unsafe Languages

- Marcus Rodrigues
- Breno Guimarães
- Fernando Quintão

The goal of this work is to develop techniques to support the dynamic analysis of program parts

The goal of this work is to develop techniques to support the dynamic analysis of program parts

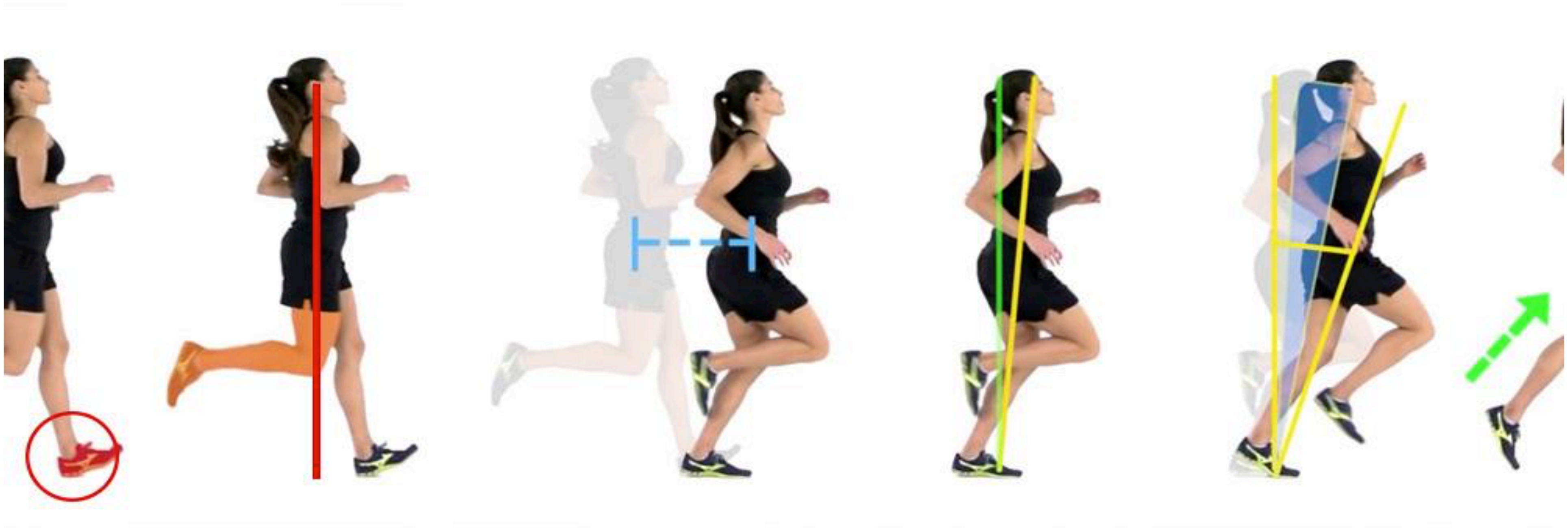
We want to produce inputs that will not cause out-of-bounds memory accesses during testing

The goal of this work is to develop techniques to support the dynamic analysis of program parts

Our insight is the observation that many array accesses in actual programs can be grouped into a category of expressions that are easy to bound

We want to produce inputs that will not cause out-of-bounds memory accesses during testing

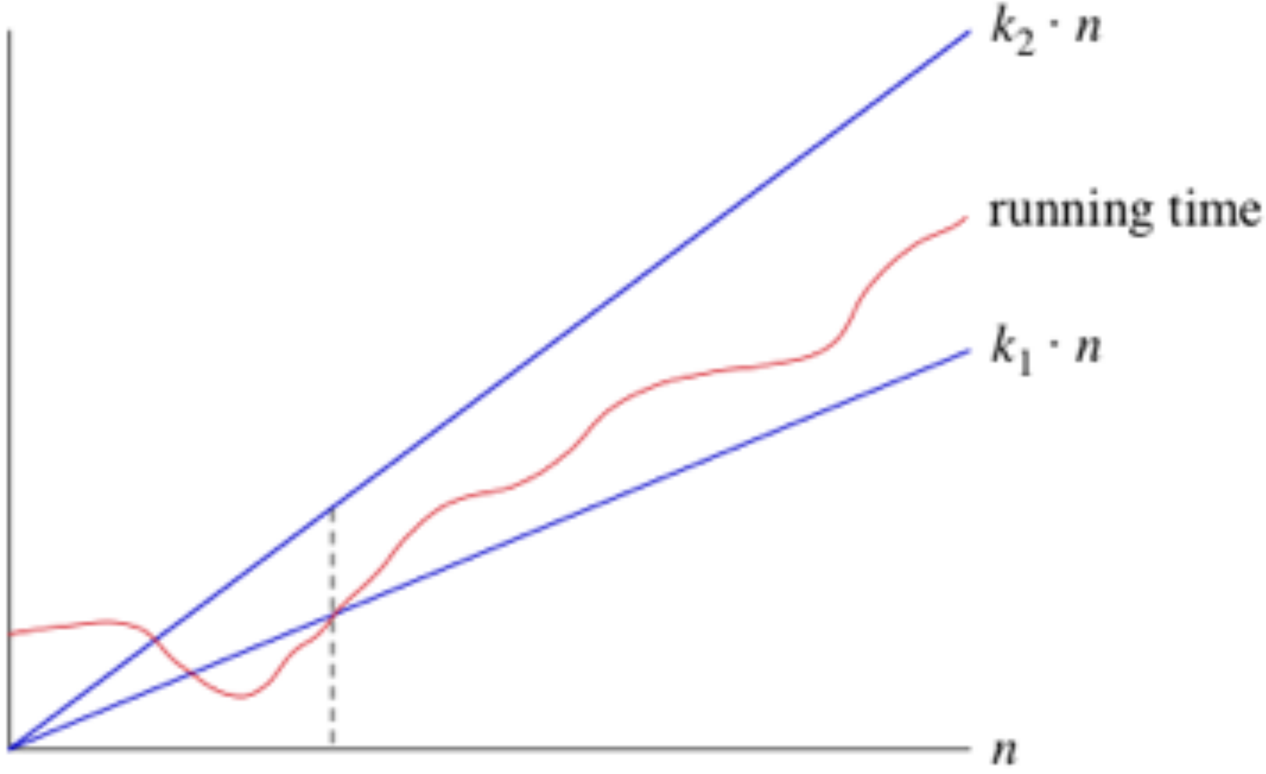
Dynamic Program Analysis



Valgrind

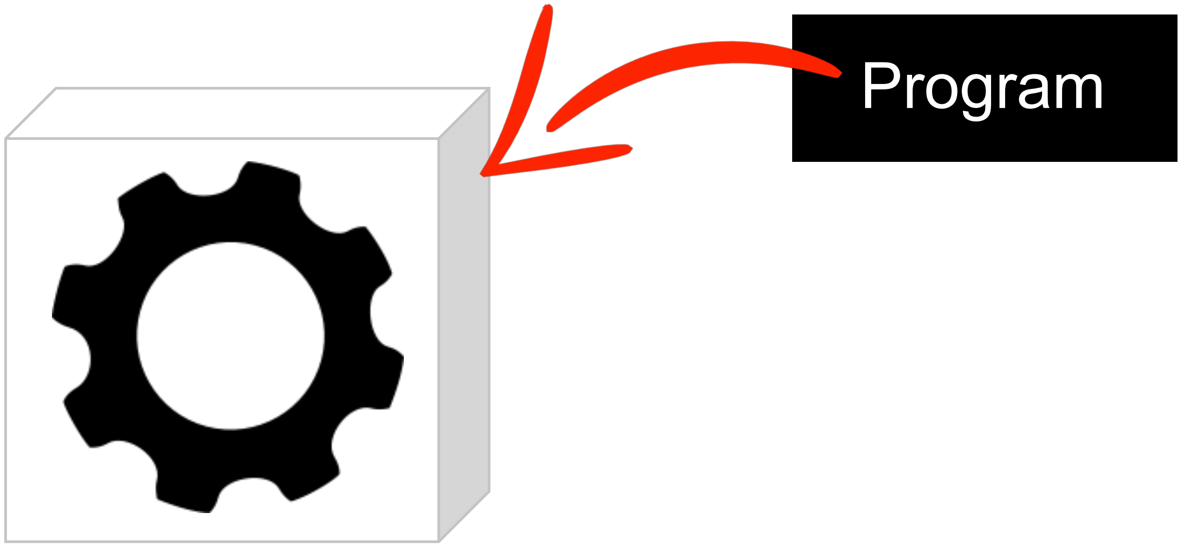
Valgrind: a framework for heavyweight dynamic binary instrumentation, PLDI'07

Aprof

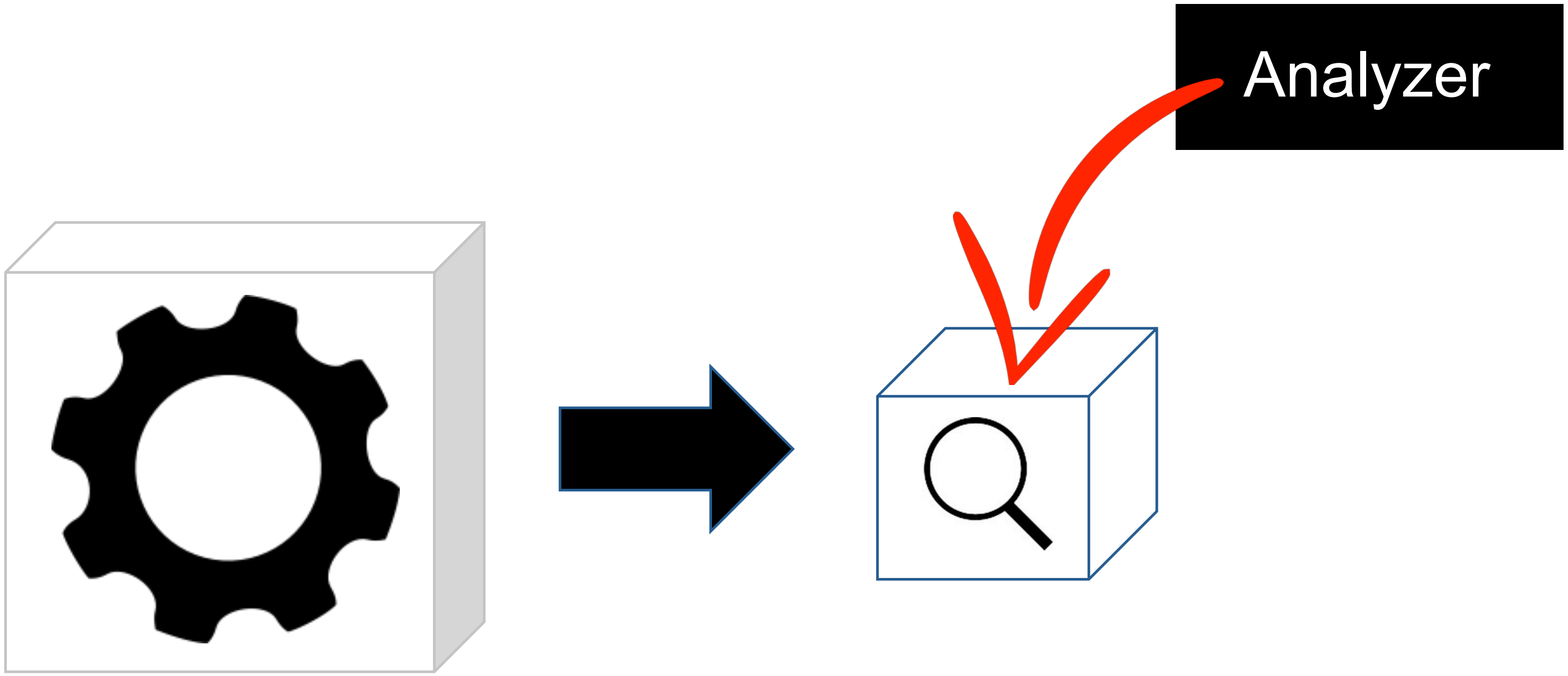


Input-sensitive profiling, PLDI'12

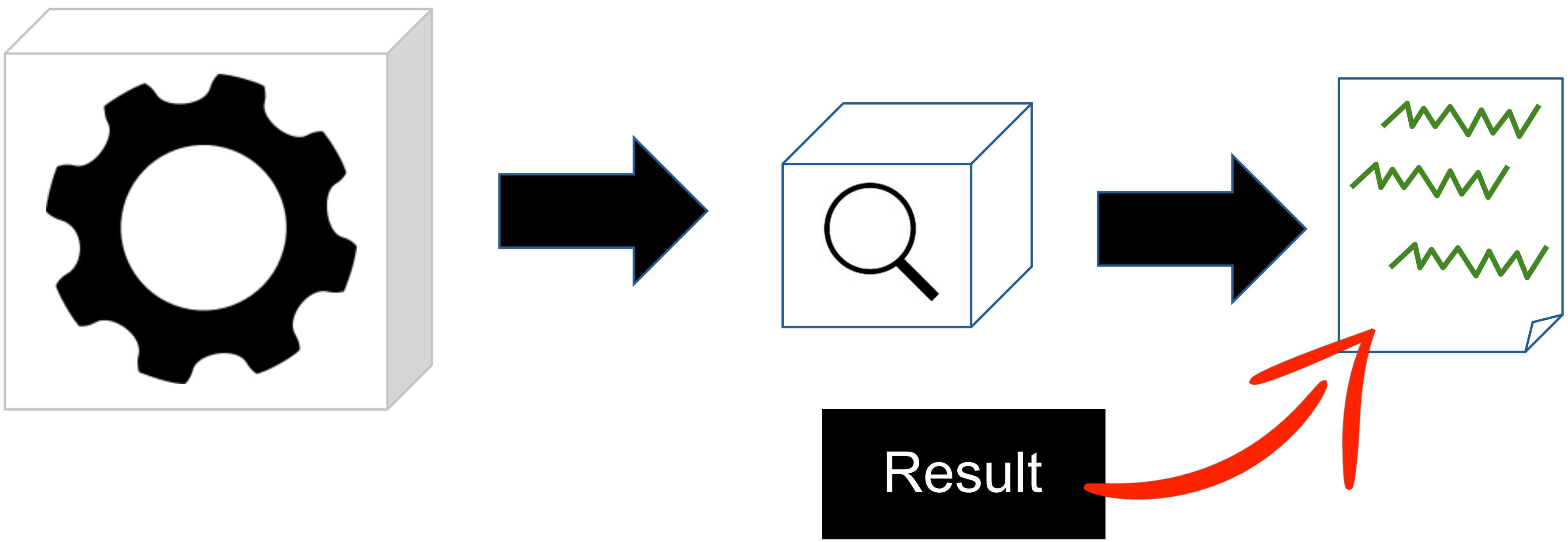
Modus Operandi: Compile

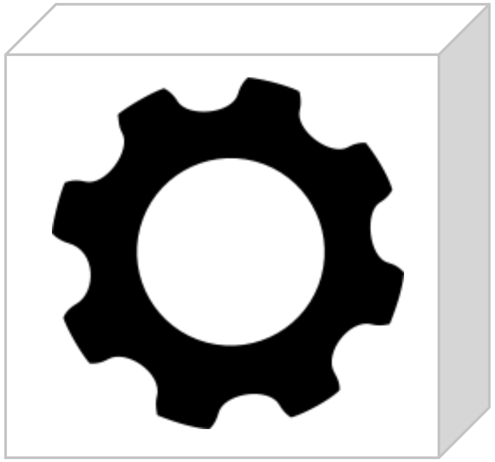


Modus Operandi: Run



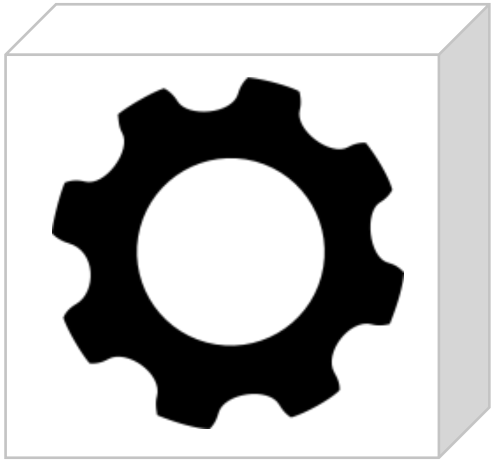
Modus Operandi: Analyze





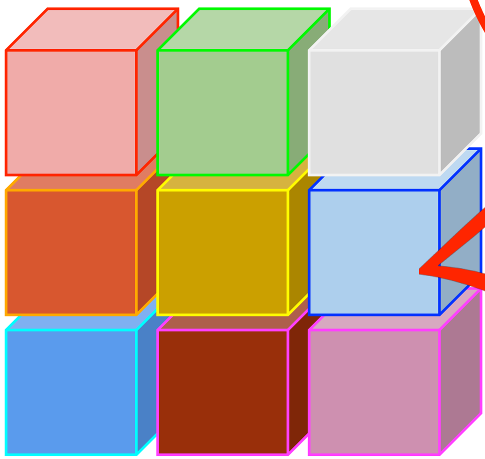
```
void kernel_2mm(int ni, int nj,
 int nk, int nl, int alpha,
int beta, float **tmp, float **A, float **B, float **C, float **D) {
int i, j, k;
  for (i = 0; i < ni; i++)
            for (j = 0; j < nj; j++) {
            tmp[i][j] = 0;
            for (k = 0; k < nk; ++k)
            tmp[i][j] += alpha * A[i][k] * B[k][j];
            }

  for (i = 0; i < ni; i++)
            for (j = 0; j < nl; j++) {
            D[i][j] *= beta;
            for (k = 0; k < nj; ++k)
            D[i][j] += tmp[i][k] * C[k][j];
            }
}

void initMatrix(int **v, int n) {
  int i, j;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      v[i][j] = 0;
    }
  }
}

void kernel_trmm(int m, int n, float alpha, float **A, float **B)
{
  int i, j, k;
  float temp;

  for (i = 0; i < m; i++)
            for (j = 0; j < n; j++) {
            for (k = i+1; k < m; k++)
            B[i][j] += A[k][i] * B[k][j];
            B[i][j] = alpha * B[i][j];
            }
}

void kernel_seidel_2d(int n, int tsteps, float **A)
{
  int t, i, j;
  for (t = 0; t <= tsteps - 1; t++)
            for (i = 1; i <= n - 2; i++)
            for (j = 1; j <= n - 2; j++)
            A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]
            + A[i][j-1] + A[i][j] + A[i][j+1]
            + A[i+1][j-1] + A[i+1][j] + A[i+1][j+1])/9.0;
}

void kernel_trisolv(int n, float **L, float *x, float *b)
{
  int i, j;
  for (i = 0; i < n; i++)
            {
            x[i] = b[i];
            for (j = 0; j <i; j++)
            x[i] -= L[i][j] * x[j];
            x[i] = x[i] / L[i][i];
            }

}
```

```
float sqrt(float);

void kernel_gramschmidt(int m, int n, float **A, float **R, float **Q)
{
  int i, j, k;

  float nrm;

  for (k = 0; k < n; k++) {
            nrm = 0.0;
            for (i = 0; i < m; i++)
            nrm += A[i][k] * A[i][k];
            R[k][k] = sqrt(nrm);
            for (i = 0; i < m; i++)
            Q[i][k] = A[i][k] / R[k][k];
            for (j = k + 1; j < n; j++) {
            R[k][j] = 0.0;
            for (i = 0; i < m; i++)
            R[k][j] += Q[i][k] * A[i][j];
            for (i = 0; i < m; i++)
            A[i][j] = A[i][j] - Q[i][k] * R[k][j];
            }
  }
}

void kernel_floyd_warshall(int **path, int n)
{
  int i, j, k;
  for (k = 0; k < n; k++) {
            for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
            path[i][j] = path[i][j] < path[i][k] + path[k][j] ?
            path[i][j] : path[i][k] + path[k][j];
  }
}

double sqrt(double x);
void kernel_cholesky(int n, float **A)
{
  int i, j, k;

  for (i = 0; i < n; i++) {
            //j<i
            for (j = 0; j < i; j++) {
            for (k = 0; k < j; k++) {
            A[i][j] -= A[i][k] * A[j][k];
            }
            A[i][j] /= A[j][j];
            }
            // i==j case
            for (k = 0; k < i; k++) {
            A[i][i] -= A[i][k] * A[i][k];
            }
            A[i][i] = sqrt(A[i][i]);
  }

}

int main () {

  return 0;
}
```



```
void kernel_2mm(int ni, int nj,
 int nk, int nl, int alpha,
int beta, float **tmp, float **A, float **B, float **C, float **D) {
int i, j, k;
  for (i = 0; i < ni; i++)
                                    for (j = 0; j < nj; j++) {
                                    tmp[i][j] = 0;
                                    for (k = 0; k < nk; ++k)
                                    tmp[i][j] += alpha * A[i][k] * B[k][j];
                                    }

  for (i = 0; i < ni; i++)
                                    for (j = 0; j < nl; j++) {
                                    D[i][j] *= beta;
                                    for (k = 0; k < nj; ++k)
                                    D[i][j] += tmp[i][k] * C[k][j];
                                    }
}

void initMatrix(int **v, int n) {
  int i, j;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      v[i][j] = 0;
    }
  }
}

void kernel_trmm(int m, int n, float alpha, float **A, float **B)
{
  int i, j, k;
  float temp;

  for (i = 0; i < m; i++)
                                    for (j = 0; j < n; j++) {
                                    for (k = i+1; k < m; k++)
                                    B[i][j] += A[k][i] * B[k][j];
                                    B[i][j] = alpha * B[i][j];
                                    }
}

void kernel_seidel_2d(int n, int tsteps, float **A)
{
  int t, i, j;
  for (t = 0; t <= tsteps - 1; t++)
                                    for (i = 1; i <= n - 2; i++)
                                    for (j = 1; j <= n - 2; j++)
                                    A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]
                                    + A[i][j-1] + A[i][j] + A[i][j+1]
                                    + A[i+1][j-1] + A[i+1][j] + A[i+1][j+1])/9.0;
}

void kernel_trisolv(int n, float **L, float *x, float *b)
{
  int i, j;
  for (i = 0; i < n; i++)
                                    {
                                    x[i] = b[i];
                                    for (j = 0; j <i; j++)
                                    x[i] -= L[i][j] * x[j];
                                    x[i] = x[i] / L[i][i];
                                    }

}
```

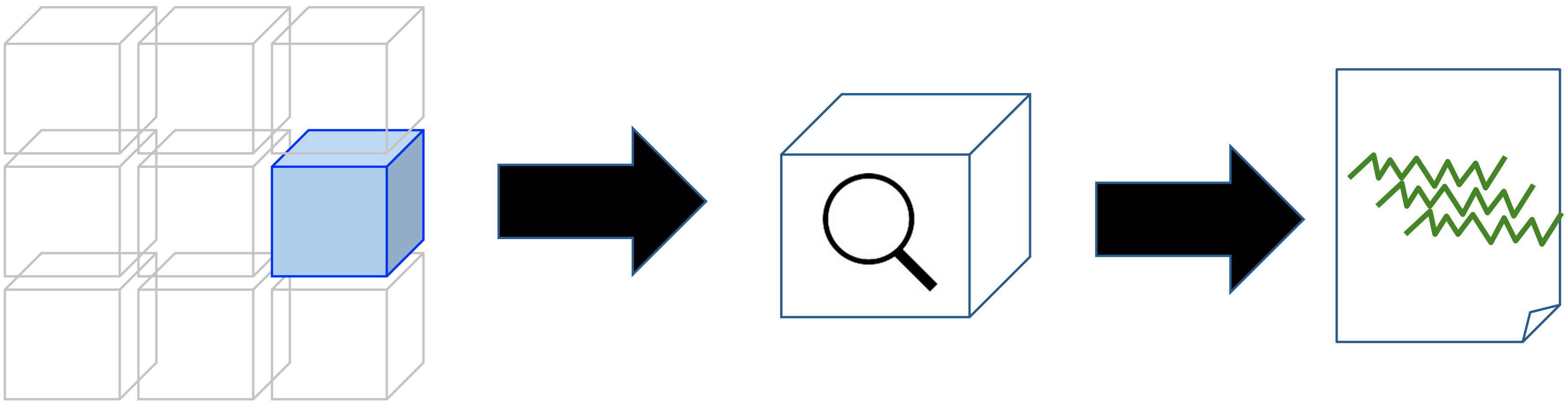
```
float sqrt(float);

void kernel_gramschmidt(int m, int n, float **A, float **R, float **Q)
{
  int i, j, k;

  float nrm;

  for (k = 0; k < n; k++) {
                                    nrm = 0.0;
                                    for (i = 0; i < m; i++)
                                    nrm += A[i][k] * A[i][k];
                                    R[k][k] = sqrt(nrm);
                                    for (i = 0; i < m; i++)
                                    Q[i][k] = A[i][k] / R[k][k];
                                    for (j = k + 1; j < n; j++) {
                                    R[k][j] = 0.0;
                                    for (i = 0; i < m; i++)
                                    R[k][j] += Q[i][k] * A[i][j];
                                    for (i = 0; i < m; i++)
                                    A[i][j] = A[i][j] - Q[i][k] * R[k][j];
                                    }
  }
}

void kernel_floyd_warshall(int **path, int n)
{
  int i, j, k;
  for (k = 0; k < n; k++) {
                                    for (i = 0; i < n; i++)
                                    for (j = 0; j < n; j++)
                                    path[i][j] = path[i][j] < path[i][k] + path[k][j] ?
                                    path[i][j] : path[i][k] + path[k][j];
  }
}

double sqrt(double x);
void kernel_cholesky(int n, float **A)
{
  int i, j, k;

  for (i = 0; i < n; i++) {
                                    //j<i
                                    for (j = 0; j < i; j++) {
                                    for (k = 0; k < j; k++) {
                                    A[i][j] -= A[i][k] * A[j][k];
                                    }
                                    A[i][j] /= A[j][j];
                                    }
                                    // i==j case
                                    for (k = 0; k < i; k++) {
                                    A[i][i] -= A[i][k] * A[i][k];
                                    }
                                    A[i][i] = sqrt(A[i][i]);
  }

}

int main () {

  return 0;
}
```



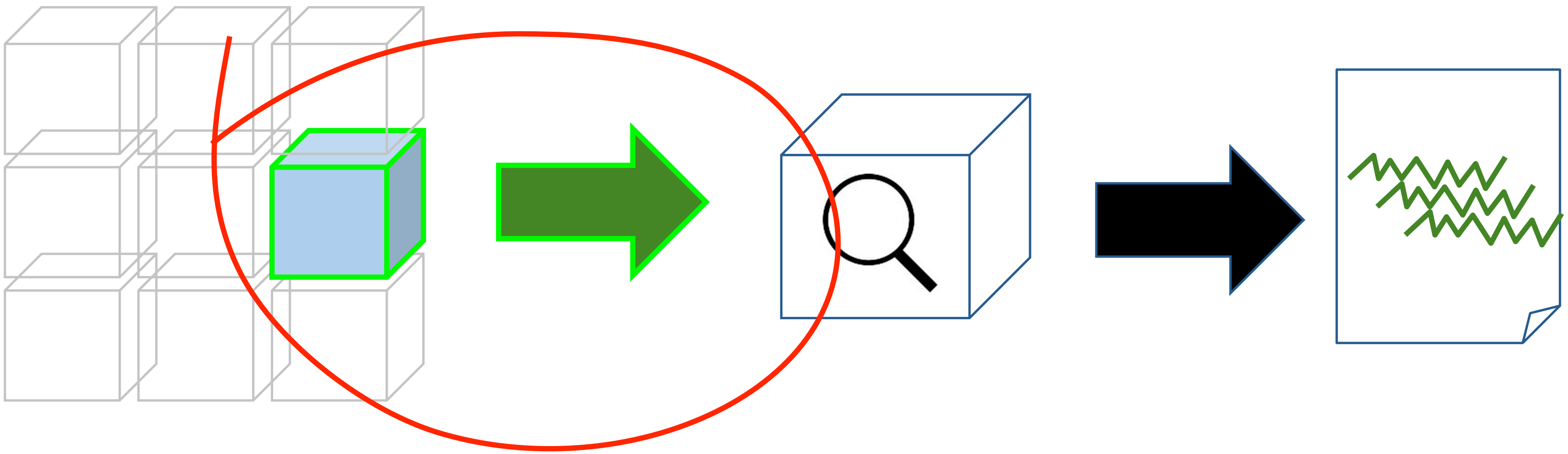
```
void kernel_2mm(int ni, int nj,
 int nk, int nl, int alpha,
int beta, float **tmp, float **A, float **B, float **C, float **D) {
int i, j, k;
  for (i = 0; i < ni; i++)
                    for (j = 0; j < nj; j++) {
                    tmp[i][j] = 0;
                    for (k = 0; k < nk; ++k)
                    tmp[i][j] += alpha * A[i][k] * B[k][j];
                    }

  for (i = 0; i < ni; i++)
                    for (j = 0; j < nl; j++) {
                    D[i][j] *= beta;
                    for (k = 0; k < nj; ++k)
                    D[i][j] += tmp[i][k] * C[k][j];
                    }
}

void initMatrix(int **v, int n) {
  int i, j;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      v[i][j] = 0;
    }
  }
}

void kernel_trmm(int m, int n, float alpha, float **A, float **B)
{
  int i, j, k;
  float te

  for (i = 0; i < m; i++)
                    for (j = 0; j < n; j++) {
                    for (k = i+1; k < m; k++)
                    B[i][j] += A[k][i] * B[k][j];
                    B[i][j] = alpha * B[i][j];
                    }
}

void kernel_seidel_2d(int n, int tsteps, float **A)
{
  int t, i, j;
  for (t = 0; t <= tsteps - 1; t++)
                    for (i = 1; i <= n - 2; i++)
                    for (j = 1; j <= n - 2; j++)
                    A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]
                    + A[i][j-1] + A[i][j] + A[i][j+1]
                    + A[i+1][j-1] + A[i+1][j] + A[i+1][j+1])/9.0;
}

void kernel_trisolv(int n, float **L, float *x, float *b)
{
  int i, j;
  for (i = 0; i < n; i++)
                    {
                    x[i] = b[i];
                    for (j = 0; j <i; j++)
                    x[i] -= L[i][j] * x[j];
                    x[i] = x[i] / L[i][i];
                    }

}
```

```
float sqrt(float);

void kernel_gramschmidt(int m, int n, float **A, float **R, float **Q)
{
  int i, j, k;

  float nrm;

  for (k = 0; k < n; k++) {
                    nrm = 0.0;
                    for (i = 0; i < m; i++)
                    nrm += A[i][k] * A[i][k];
                    R[k][k] = sqrt(nrm);
                    for (i = 0; i < m; i++)
                    Q[i][k] = A[i][k] / R[k][k];
                    for (j = k + 1; j < n; j++) {
                    R[k][j] = 0.0;
                    for (i = 0; i < m; i++)
                    R[k][j] += Q[i][k] * A[i][j];
                    for (i = 0; i < m; i++)
                    A[i][j] = A[i][j] - Q[i][k] * R[k][j];
                    }
  }
}

void kernel_floyd_warshall(int **path, int n)
{
  int i, j, k;
  for (k = 0; k < n; k++) {
                    for (i = 0; i < n; i++)
                    for (j = 0; j < n; j++)
                    path[i][j] = path[i][j] < path[i][k] + path[k][j] ?
                    path[i][j] : path[i][k] + path[k][j];
  }
}

double sqrt(double x);
void kernel_cholesky(int n, float **A)
{
  int i, j, k;

  for (i = 0; i < n; i++) {
                    //j<i
                    for (j = 0; j < i; j++) {
                    for (k = 0; k < j; k++) {
                    A[i][j] -= A[i][k] * A[j][k];
                    }
                    A[i][j] /= A[j][j];
                    }
                    // i==j case
                    for (k = 0; k < i; k++) {
                    A[i][i] -= A[i][k] * A[i][k];
                    }
                    A[i][i] = sqrt(A[i][i]);
  }

}

int main () {

  return 0;
}
```

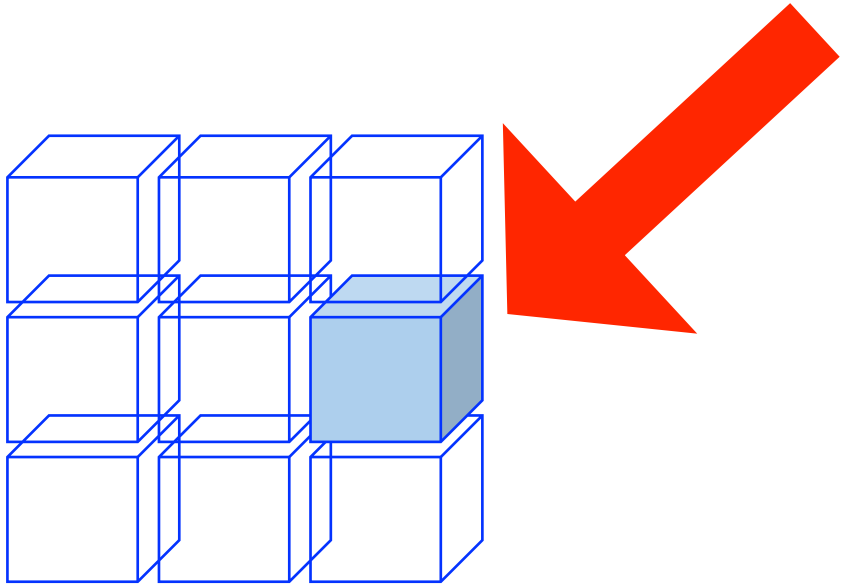
It's hard to analyze individual program parts



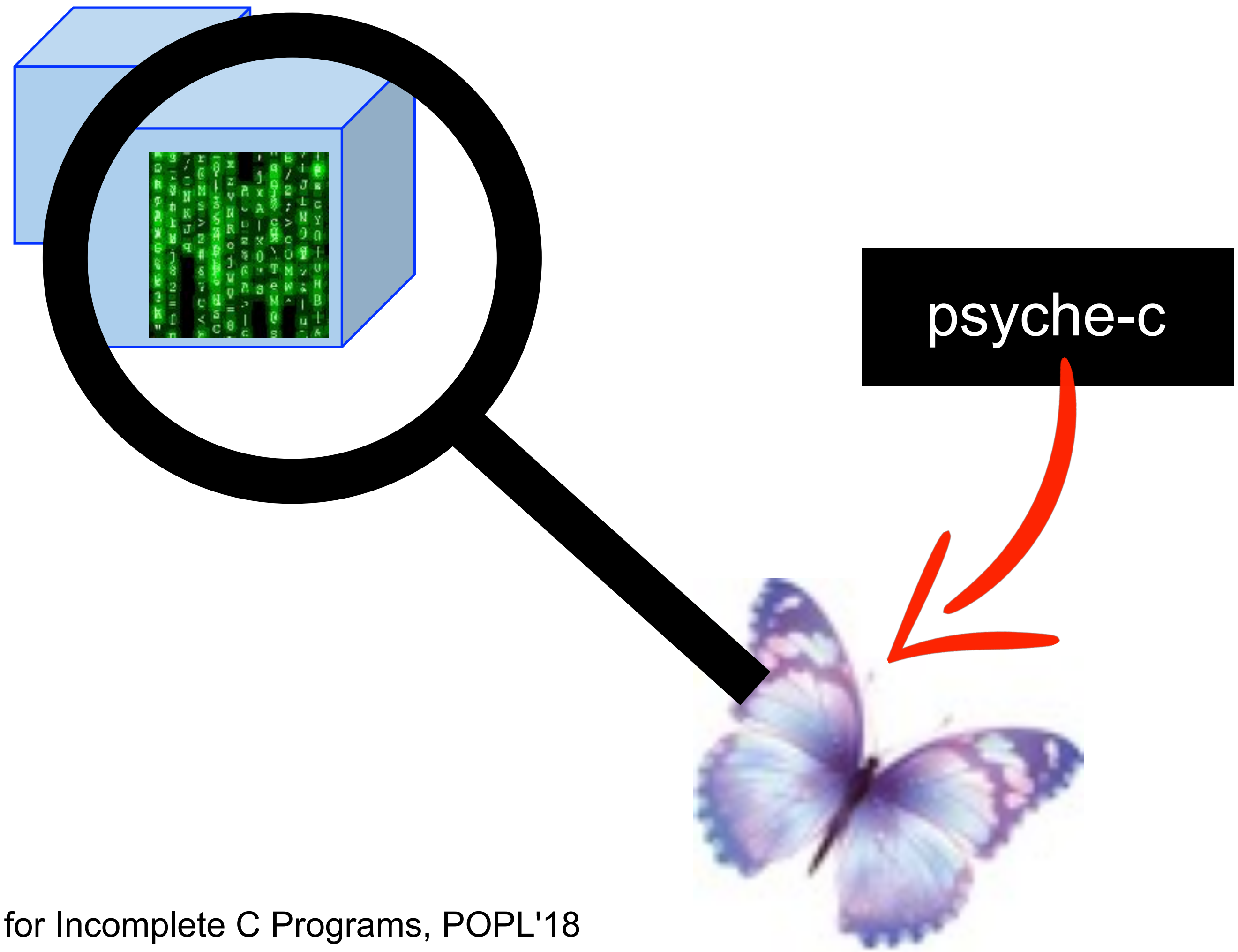
Goal



Pick the part of the program that you want

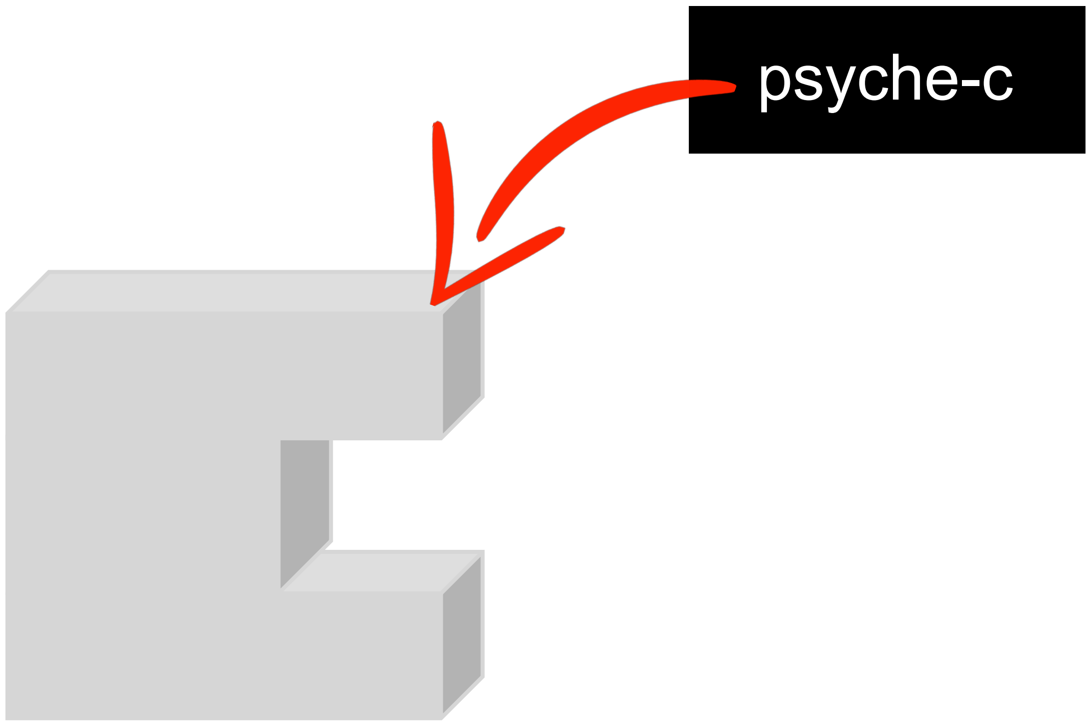


Fill in the holes, so that it compiles

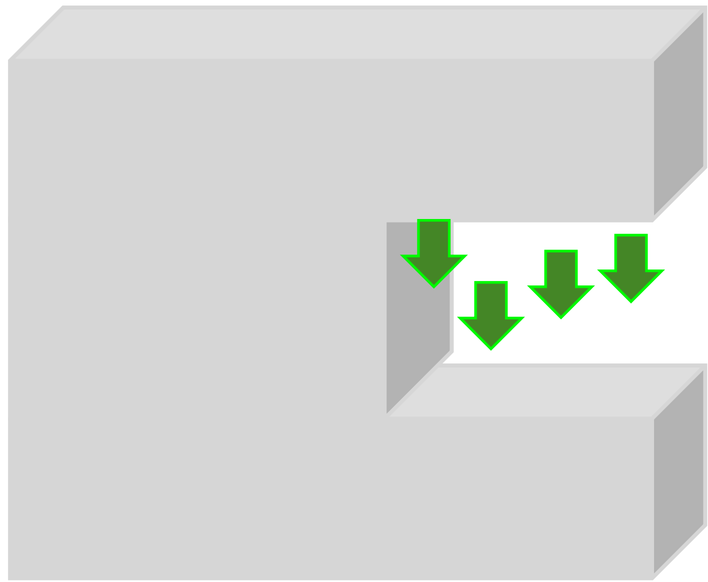


Inference of Static Semantics for Incomplete C Programs, POPL'18

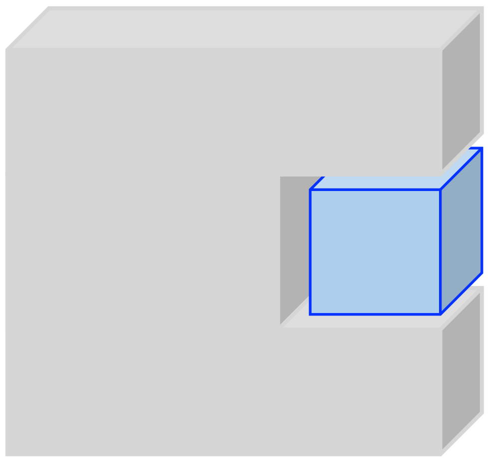
Build a driver to run the individual part



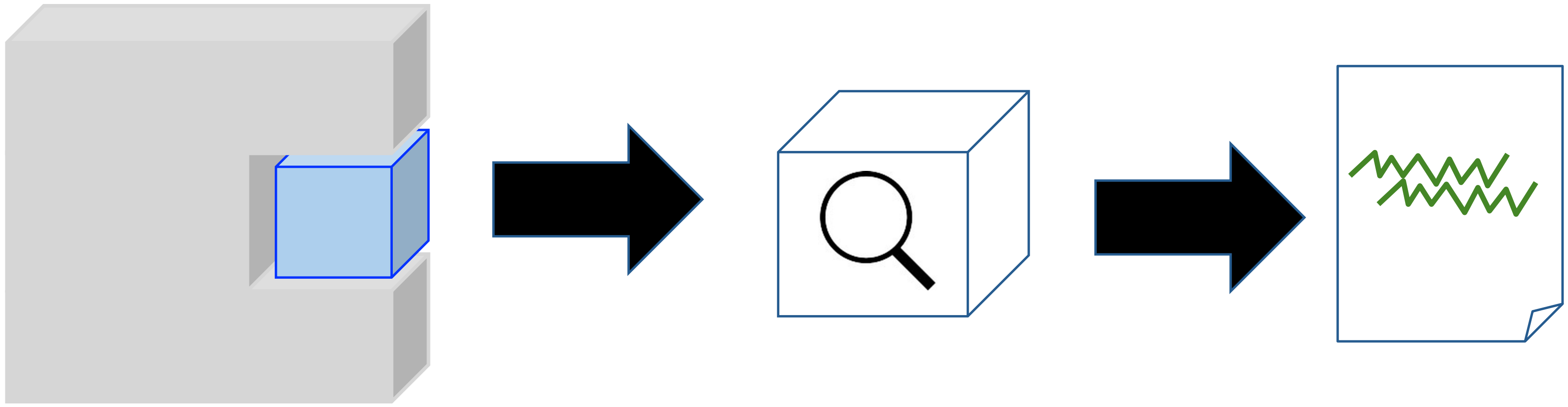
The driver must generate valid inputs



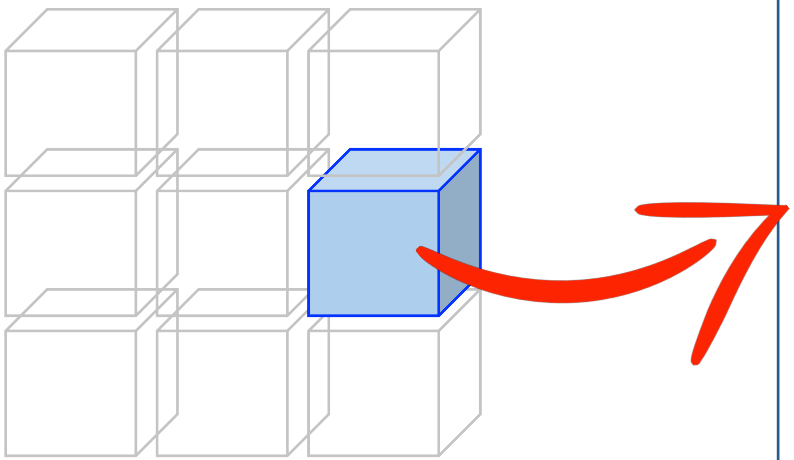
Driver + Program part = Testable program



Perform the dynamic analysis on this ensemble

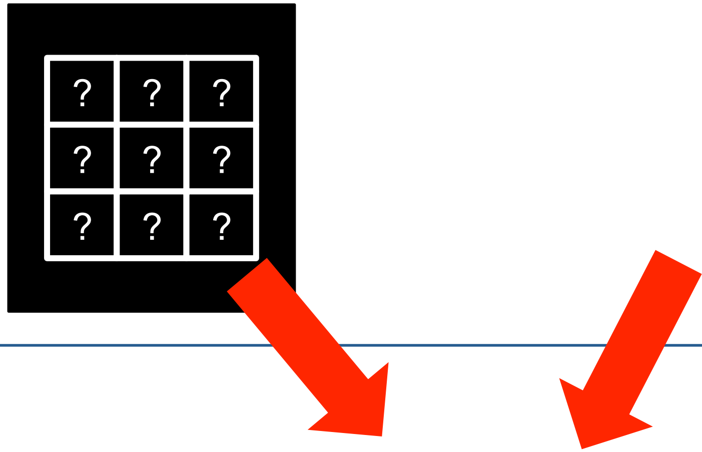


Example: Initializing a Matrix

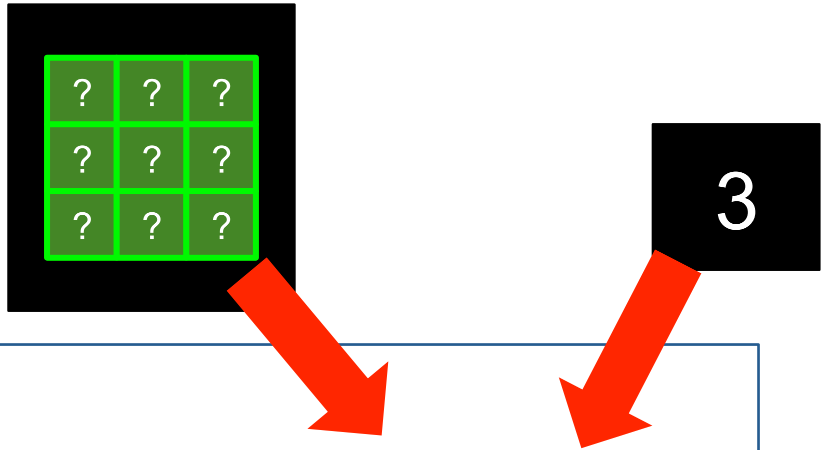


```
void initMatrix(int **v, int n) {
  int i, j;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      v[i][j] = 0;
    }
  }
}
```

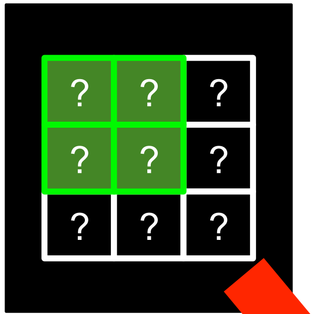
```
void initMatrix(int **v, int n) {
  int i, j;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      v[i][j] = 0;
    }
  }
}
```




```c
void initMatrix(int **v, int n) {
  int i, j;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      v[i][j] = 0;
    }
  }
}
```

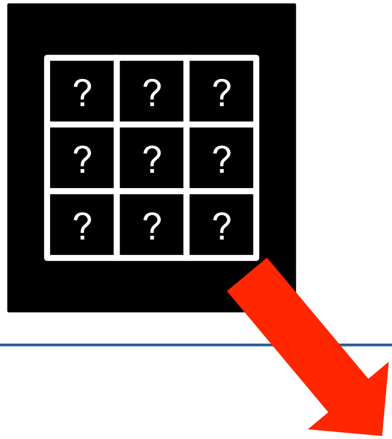


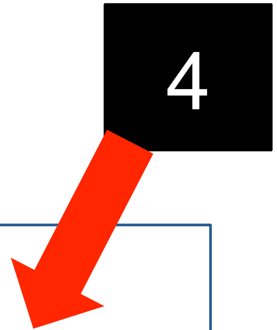

```
void initMatrix(int **v, int n) {
  int i, j;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      v[i][j] = 0;
    }
  }
}
```



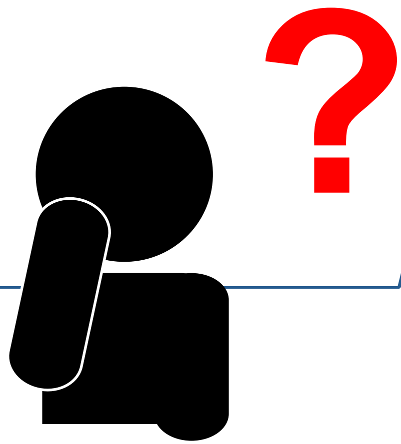
2

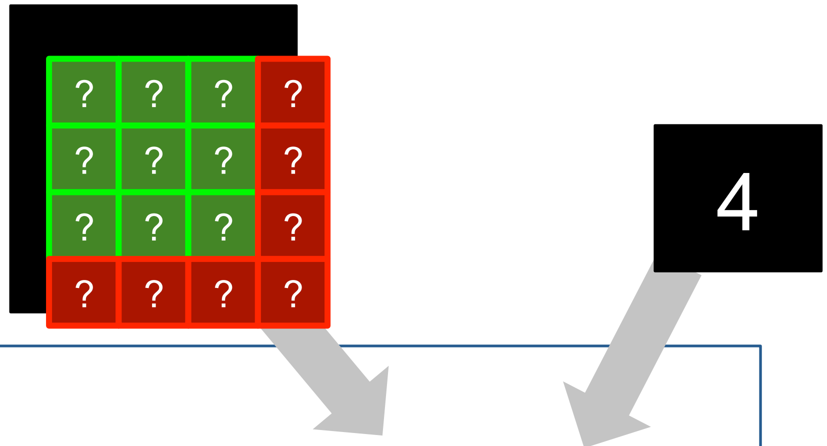
```
void initMatrix(int **v, int n) {
  int i, j;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      v[i][j] = 0;
    }
  }
}
```



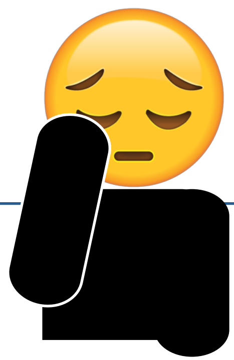


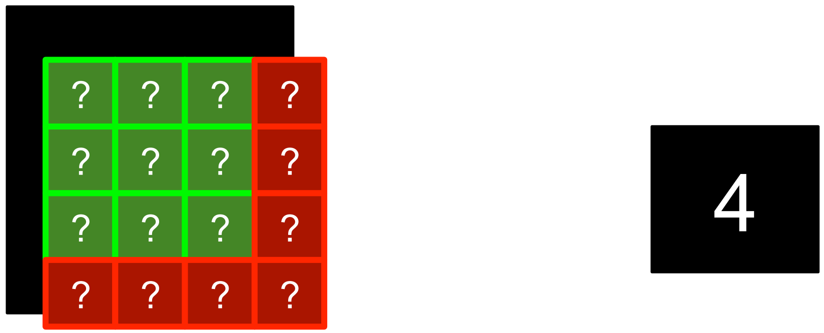

```
void initMatrix(int **v, int n) {
  int i, j;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      v[i][j] = 0;
    }
  }
}
```






```
void initMatrix(int **v, int n) {
  int i, j;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      v[i][j] = 0;
    }
  }
}
```





Out of bounds accesses cause undefined behavior

There is no contract between **v** and **n**

```
void initMatrix(int **v, int n) {
  int i, j;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      v[i][j] = 0;
    }
  }
}
```

We will show how to create such contracts

```
void initMatrix(int **v, int n) {
  int i, j;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      v[i][j] = 0;
    }
  }
}
```

Compilers Laboratory

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

SYMBOLIC INTERVAL ANALYSIS

$$R(i) = [0, N - 1]$$

A technique that finds conservative approximations for the upper and lower values of each variable

We build contracts between variables using a Symbolic Interval Analysis

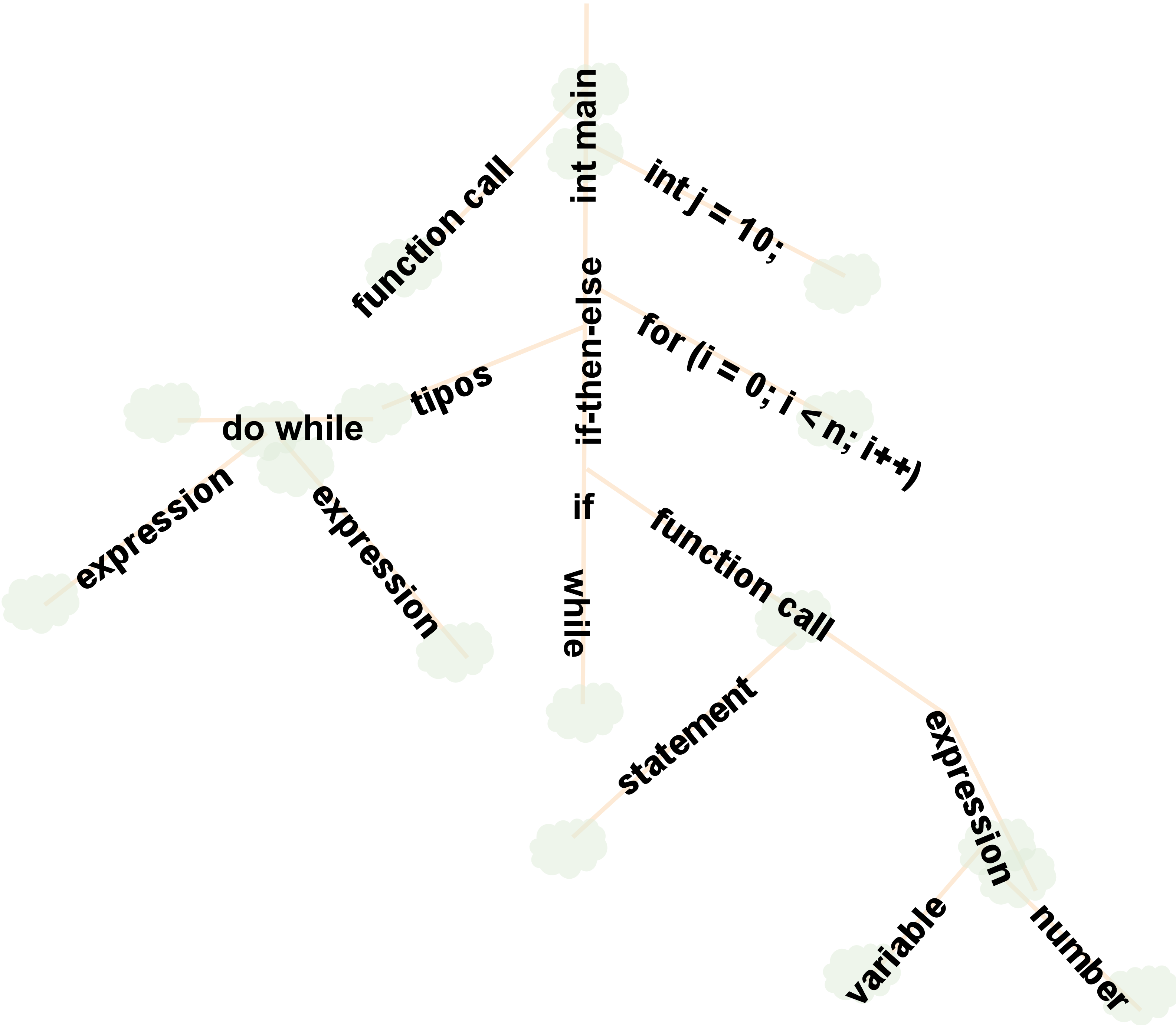
A technique that finds conservative approximations for the upper and lower values of each variable

We build contracts between variables using a Symbolic Interval Analysis

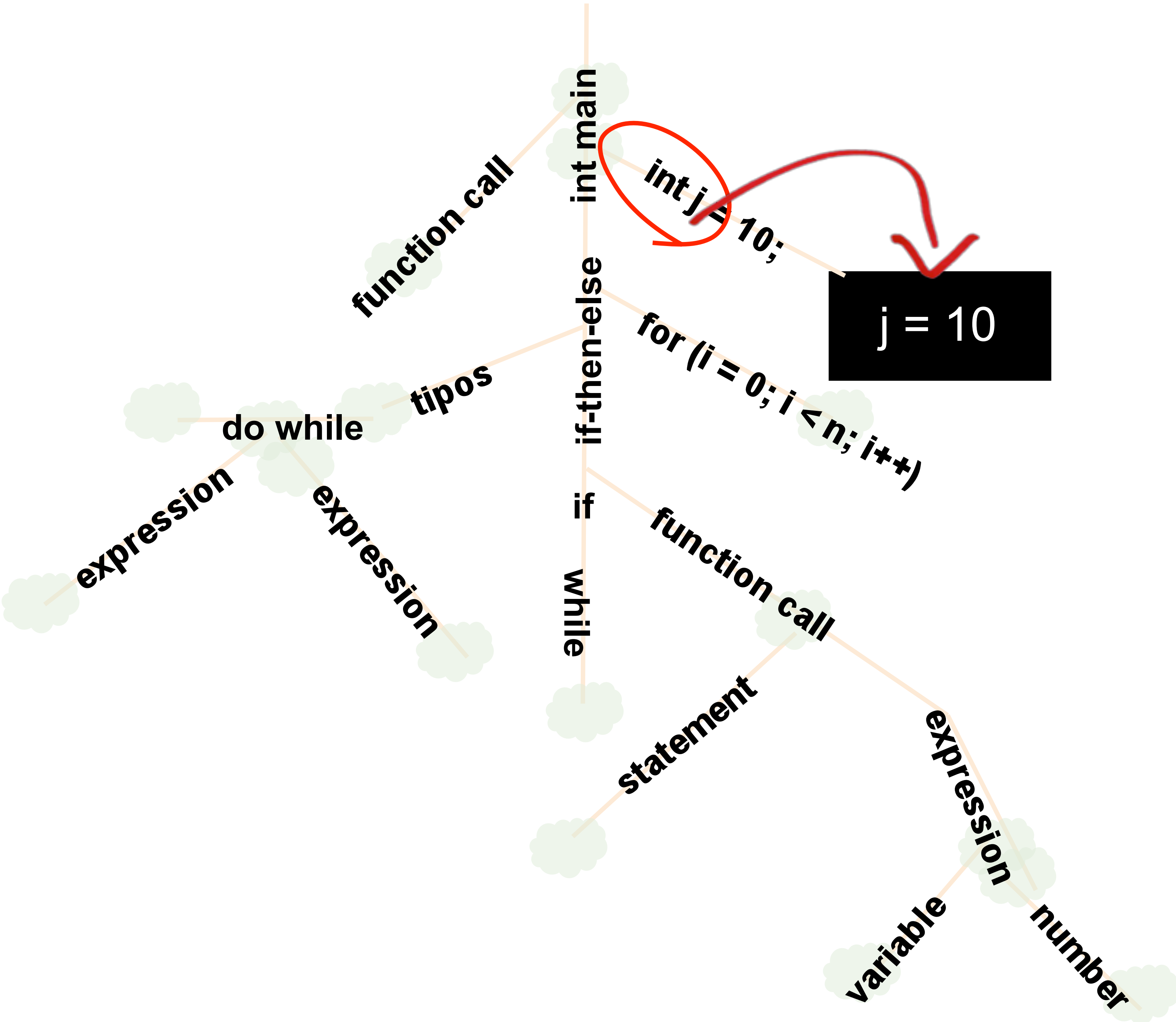
```
void foo(int *v) {
  int i = 0;

  while (i < N) {

    v[i] = 0;
    i++;

  }

}
```

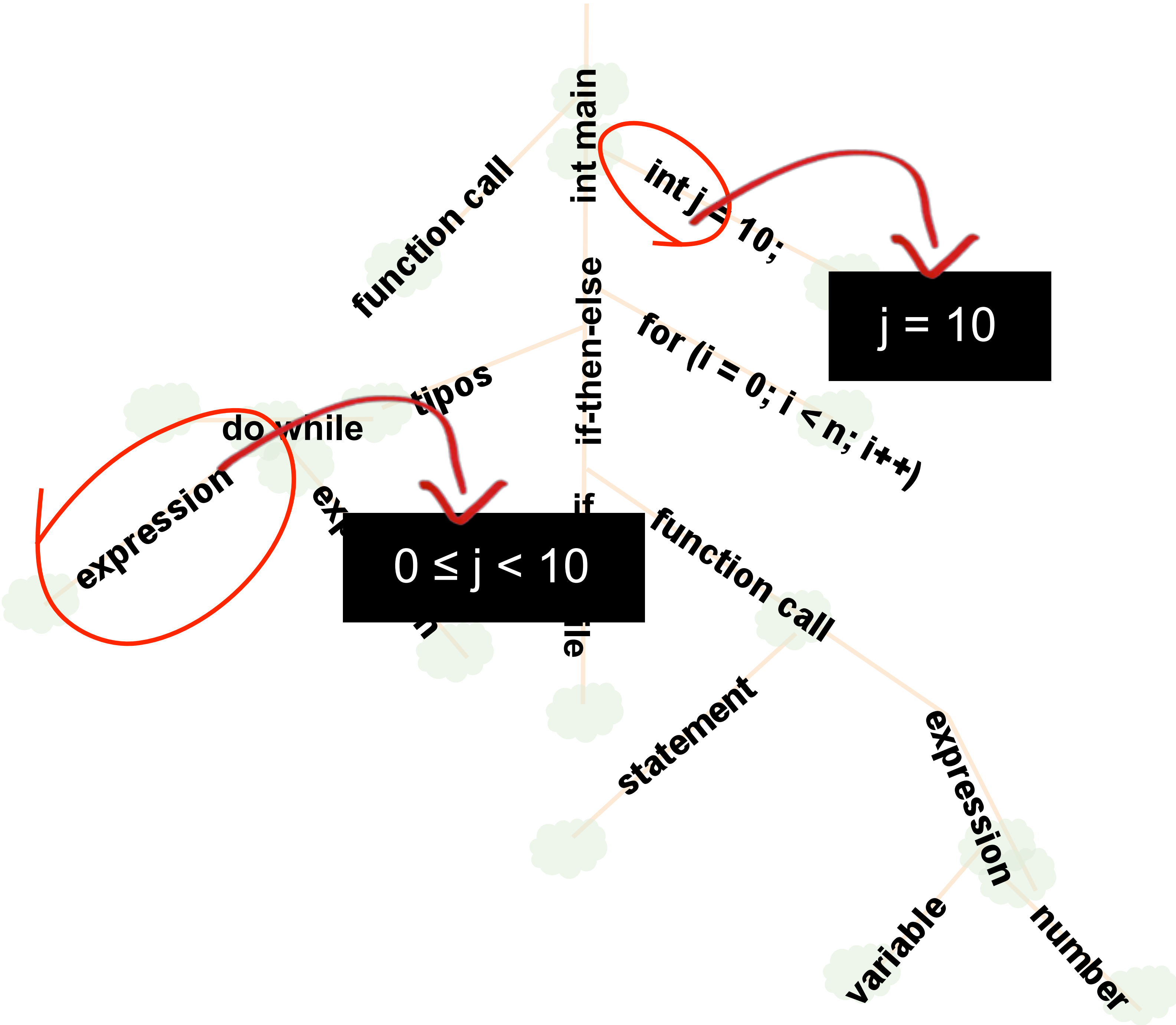
```
void foo(int *v) {
  int i = 0;
  // R(i) = [0, 0]
  while (i < N) {
    // R(i) = [0, N-1]
    v[i] = 0;
    i++;
    // R(i) = [1, N]
  }
  // R(i) = [0, N]
}
```

A technique that finds conservative approximations for the upper and lower values of each variable

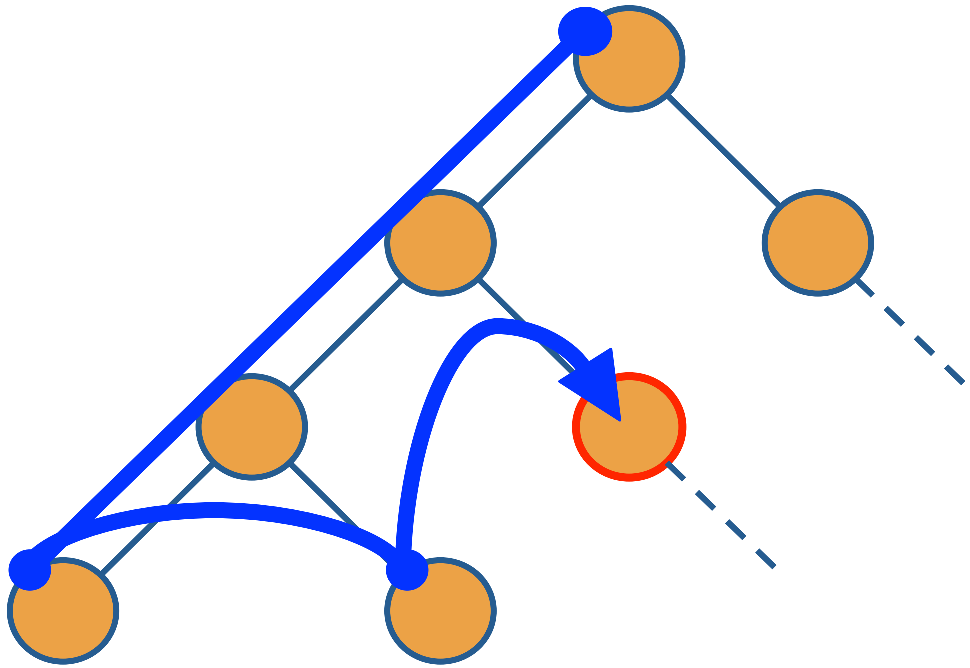


int main
function call
int j = 10;
if-then-else
for (i = 0; i < n; i++)
tipos
do while
expression
expression
if
while
function call
statement
expression
variable
number


int main
function call
int j = 10;
j = 10
if-then-else
for (i = 0; i < n; i++)
tipos
do while
expression
expression
if
while
function call
statement
expression
variable
number


int main
function call
int j = 10;
j = 10
if-then-else
for (i = 0; i < n; i++)
tipos
do while
expression
0 ≤ j < 10
function call
statement
expression
variable
number

Symbolic Interval Analysis works by applying rules on the nodes of the program's AST



Abstract Interpretation

$$\mathbf{rg}(R, \text{skip}) = R \qquad \frac{R' = R \setminus v \mapsto [n, n]}{\mathbf{rg}(R, v = n) = R'} \qquad \frac{R' = R \setminus v \mapsto [s, s]}{\mathbf{rg}(R, v = s) = R'} \qquad \frac{R(v_1) = [l, u] \qquad R' = R \setminus v \mapsto [l, u]}{\mathbf{rg}(R, v = v_1) = R'}$$

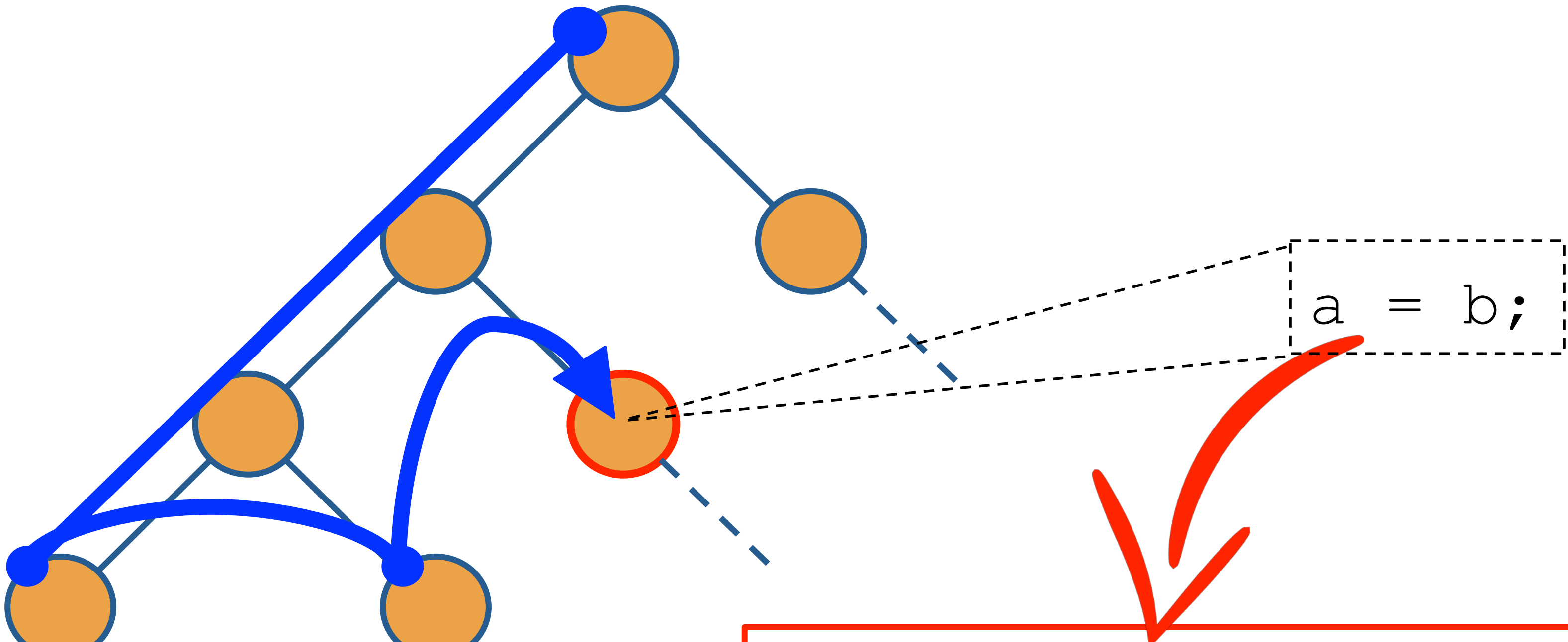
$$\frac{R(v_1) = [l_1, u_1] \qquad R(v_2) = [l_2, u_2] \qquad R' = R \setminus v \mapsto ([l_1 + l_2, u_1 + u_2])}{\mathbf{rg}(R, v = v_1 + v_2) = R'} \qquad \frac{\mathbf{rg}(R, S_1) = R_1 \qquad \mathbf{rg}(R_1, S_2) = R_2}{\mathbf{rg}(R, S_1; S_2) = R_2}$$

$$\frac{\begin{array}{c} R(v_a) = [l_a, u_a] \qquad R(v_b) = [l_b, u_b] \qquad R_t = (R \setminus v_a \to [l_a, \min(u_b - 1, u_a)]) \setminus v_b \to [\max(l_a + 1, l_b), u_b] \\ \mathbf{rg}(R_t, S_t) = R'_t \qquad R_f = (R \setminus v_a \to [\max(l_a, l_b), u_a]) \setminus v_b \to [l_b, \min(u_a, u_b)] \qquad \mathbf{rg}(R_f, S_f) = R'_f \qquad R' = R'_t \sqcup R'_f \end{array}}{\mathbf{rg}(R, \text{if}(v_a < v_b)\ S_t \text{ else } S_f) = R'}$$

$$\frac{\mathbf{fp}(R, \text{if}(v_a < v_b)\ S \text{ else skip}) = R'}{\mathbf{rg}(R, \text{while}(v_a < v_b)\ S) = R'} \qquad \frac{\mathbf{rg}(R, S) = R}{\mathbf{fp}(R, S) = R} \qquad \frac{\mathbf{rg}(R, S) = R_1 \qquad R_1 \neq R \qquad \mathbf{fp}(R_1, S) = R'}{\mathbf{fp}(R, S) = R'}$$

Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, POPL'77

Visitors and Pattern Matching



$$\frac{R(v_1) = [l, u] \qquad R' = R \setminus v \mapsto [l, u]}{\mathbf{rg}(R, v = v_1) = R'}$$

SYMBOLIC SUMMATIONS

Most memory accesses in actual programs have a structure that is simple to analyze and bound

```
void initMatrix(int **v, int n) {
  int i, j;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      v[i][j] = 0;
    }
  }
}
```

$$e = x_1 + x_2 + \ldots + x_n + c$$

Most memory accesses in actual programs have a structure that is simple to bound

$$e = x_1 + x_2 + \ldots + x_n + c$$

$e ::=$	c	constants
	x	traceable variables
	$e + e$	summations
	$e \times e$	multiplications

Most memory accesses in actual programs have a structure that is simple to bound

The values assigned to symbolic summations grow monotonically

$$
\begin{array}{lll}
e ::= & c & \text{constants} \\
& x & \text{traceable variables} \\
& e + e & \text{summations} \\
& e \times e & \text{multiplications}
\end{array}
$$

Most memory accesses in actual programs have a structure that is simple to bound

- Formal arguments
- Unambiguous global variables
  - No aliasing
- Return value of functions

traceable variables

Most memory accesses in actual programs have a structure that is simple to bound

```
float* get_vector(int Width);

float* convol(float* mm, int row, int col, int N) {
  int i, j;
  float* v = get_vector(N);
  for (i = 0; i < N; i++) {
    v[i] = mm[(1+row) * (N+1) + i + 1] * mm[1 + col + (i+1) * (N+1)];
  }
  return v;
}
```

First order traceable variables

- Formal arguments
- Unambiguous global variables
  - No aliasing
- Return value of functions

Most memory accesses in actual programs have a structure that is simple to bound

```
float* get_vector(int Width);

float* convol(float* mm, int row, int col, int N) {
  int i, j;
  float* v = get_vector(N);
  for (i = 0; i < N; i++) {
    v[i] = mm[(1+row) * (N+1) + i + 1] * mm[1 + col + (i+1) * (N+1)];
  }
  return v;
}
```

Second order traceable variables

- Formal arguments
- Unambiguous global variables
  - No aliasing
- Return value of functions

Most memory accesses in actual programs have a structure that is simple to bound

```
float* get_vector(int Width);

float* convol(float* mm, int row, int col, int N) {
  int i, j;
  float* v = get_vector(N);
  for (i = 0; i < N; i++) {
    v[i] = mm[(1+row) * (N+1) + i + 1] * mm[1 + col + (i+1) * (N+1)];
  }
  return v;
}
```

Symbolic summations

Most memory accesses in actual programs have a structure that is simple to bound

```
float* get_vector(int Width);

float* convol(float* mm, int row, int col, int N) {
  int i, j;
  float* v = get_vector(N);
  for (i = 0; i < N; i++) {
    v[i] = mm[(1+row) * (N+1) + i + 1] * mm[1 + col + (i+1) * (N+1)];
  }
  return v;
}
```

"He who controls the traceable variables controls the function to be tested"

Occurences of symbolic summations in GNU BinUtils

Binutils	Instrs.	Memory	Traceable	Monotonic
base64	412	15	6	6
basename	197	9	6	6
chgrp	278	32	31	31
chroot	533	24	22	22
chmod	551	64	62	62
cksum	235	7	4	2
cat	678	44	42	41
chown	301	45	44	44
comm	751	246	212	212
chcon	507	30	29	29
chowncore	647	118	118	118

If we replace program variables by their upper limits (as found by the interval analysis)[§], then the resulting expression is still a symbolic summation

Theorem 3.6 (Preservation)

§: Unless the upper range is infinite

If we replace program variables by their upper limits (as found by the interval analysis), then the resulting expression is still a symbolic summation

Theorem 3.6 (Preservation)

$1 + col + (i+1) * (N+1) \quad \longrightarrow \quad 1 + col + N*N + N$

- R(col) = [col, col]
- R(N) = [N, N]
- R(i) = [0, N-1]

If we replace program variables by their upper limits (as found by the interval analysis), then the resulting expression is still a symbolic summation

Theorem 3.6 (Preservation)

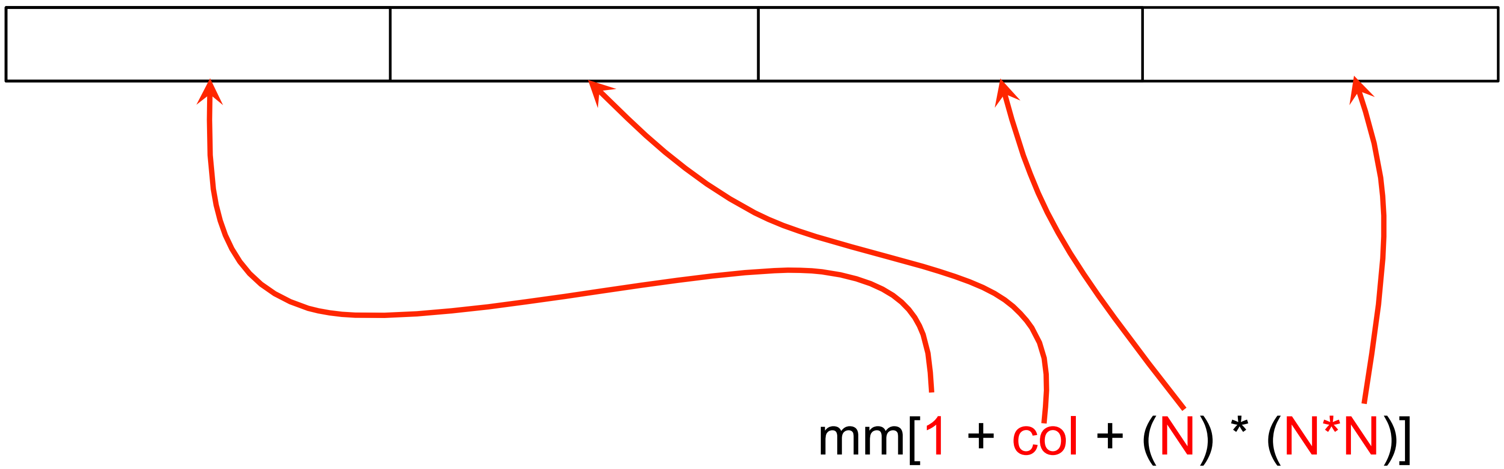
1 + col + (i+1) * (N+1) 1 + col + N*N + N

After this substitution, we get a symbolic summation that is a function only of first-order traceable variables (which we control)

If an array is only indexed by symbolic summations, then we know how to replace traceable variables by concrete values, so that the array is only accessed by indices within its allocated bounds

```
float* get_vector(int Width);

float* convol(float* mm, int row, int col, int N) {
  int i, j;
  float* v = get_vector(N);
  for (i = 0; i < N; i++) {
    v[i] = mm[(1+row) * (N+1) + i + 1] * mm[1 + col + (i+1) * (N+1)];
  }
  return v;
}
```

If an array is only indexed by symbolic summations, then we know how to replace traceable variables by concrete values, so that the array is only accessed by indices within its allocated bounds



Details are in the paper, but the key idea is that we can recursively divide the task of creating valid indices among each sum in the symbolic summation

EXPERIMENTS

Correctly analyzed kernels (checked with Valgrind)

30

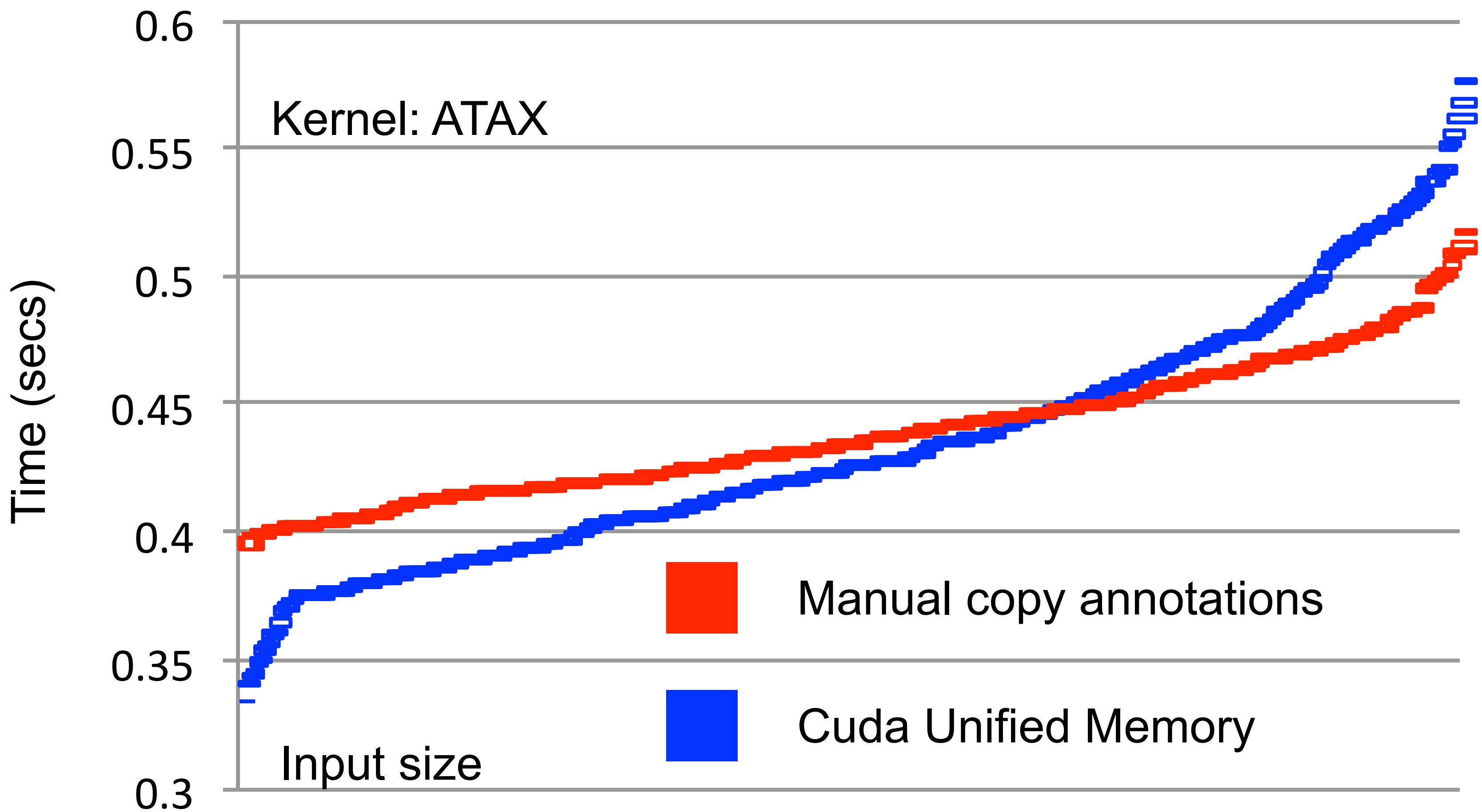
Performance test on Polybench: Cuda Unified Memory vs Manual Transfer Annotations

Changes

÷

Performance test on Polybench: Cuda Unified Memory vs Manual Transfer Annotations

The Result Changes Depending on the Input Size



Performance test on Polybench: Cuda Unified Memory vs Manual Transfer Annotations

Number of distinctic array accesses analyzed

99

Performance test on Polybench: Cuda Unified Memory vs Manual Transfer Annotations

Kernels analyzed with Aprof

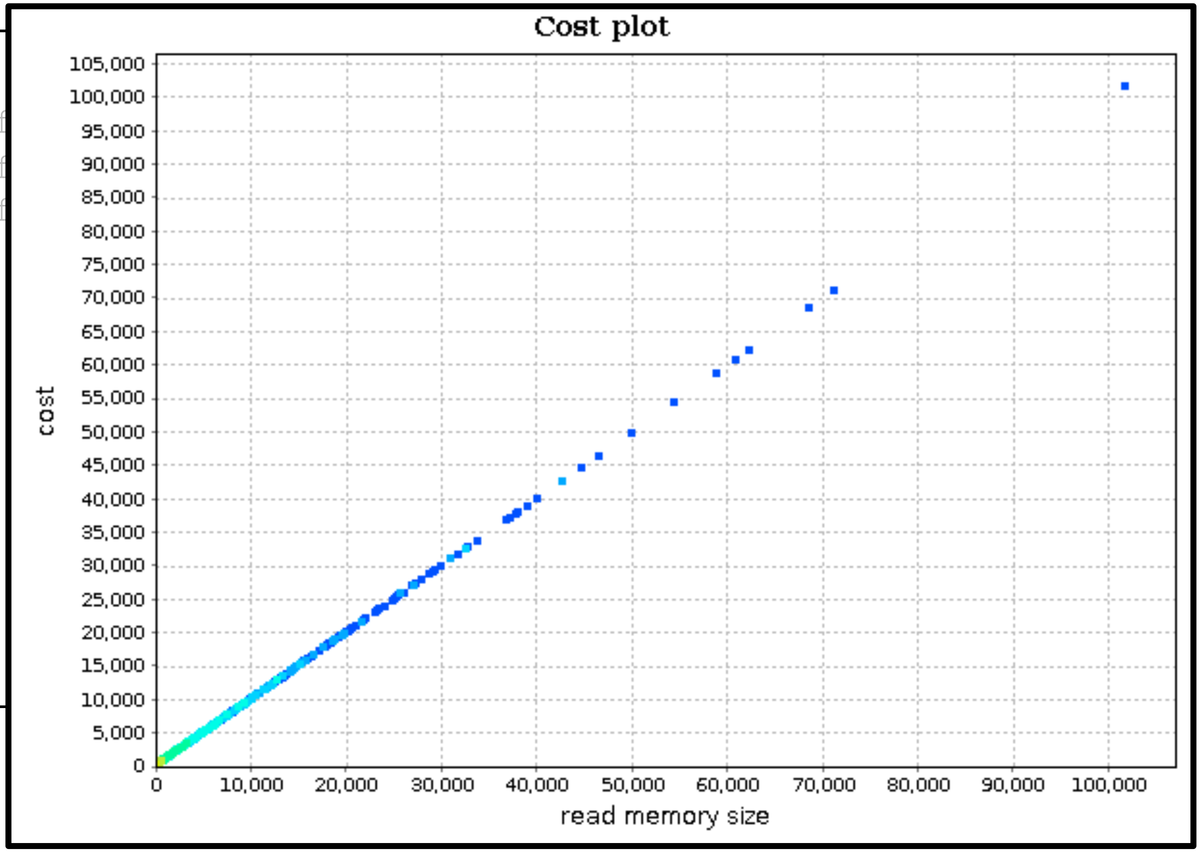
30

kernel_trisolv

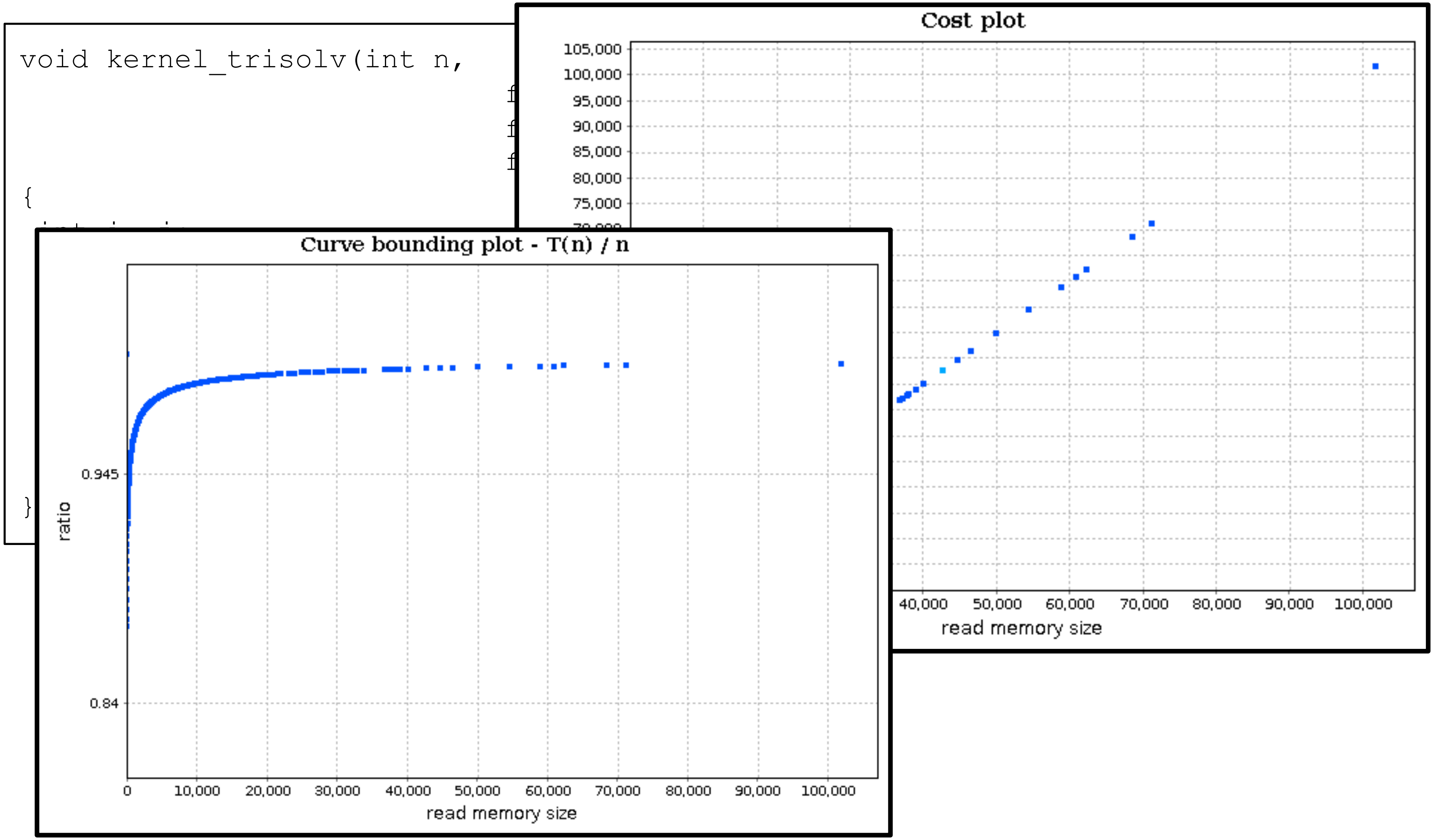
```
void kernel_trisolv(int n,
                     float **L,
                     float *x,
                     float *b)
{
 int i, j;
 for (i = 0; i < n; i++)
 {
  x[i] = b[i];
  for (j = 0; j <i; j++)
   x[i] -= L[i][j] * x[j];
  x[i] = x[i] / L[i][i];
 }
}
```

kernel_trisolv

```
void kernel_trisolv(int n,
                         f
                         f
                         f
{
 int i, j;
 for (i = 0; i < n; i++)
 {
  x[i] = b[i];
  for (j = 0; j <i; j++)
   x[i] -= L[i][j] * x[j];
  x[i] = x[i] / L[i][i];
 }
}
```



kernel_trisolv

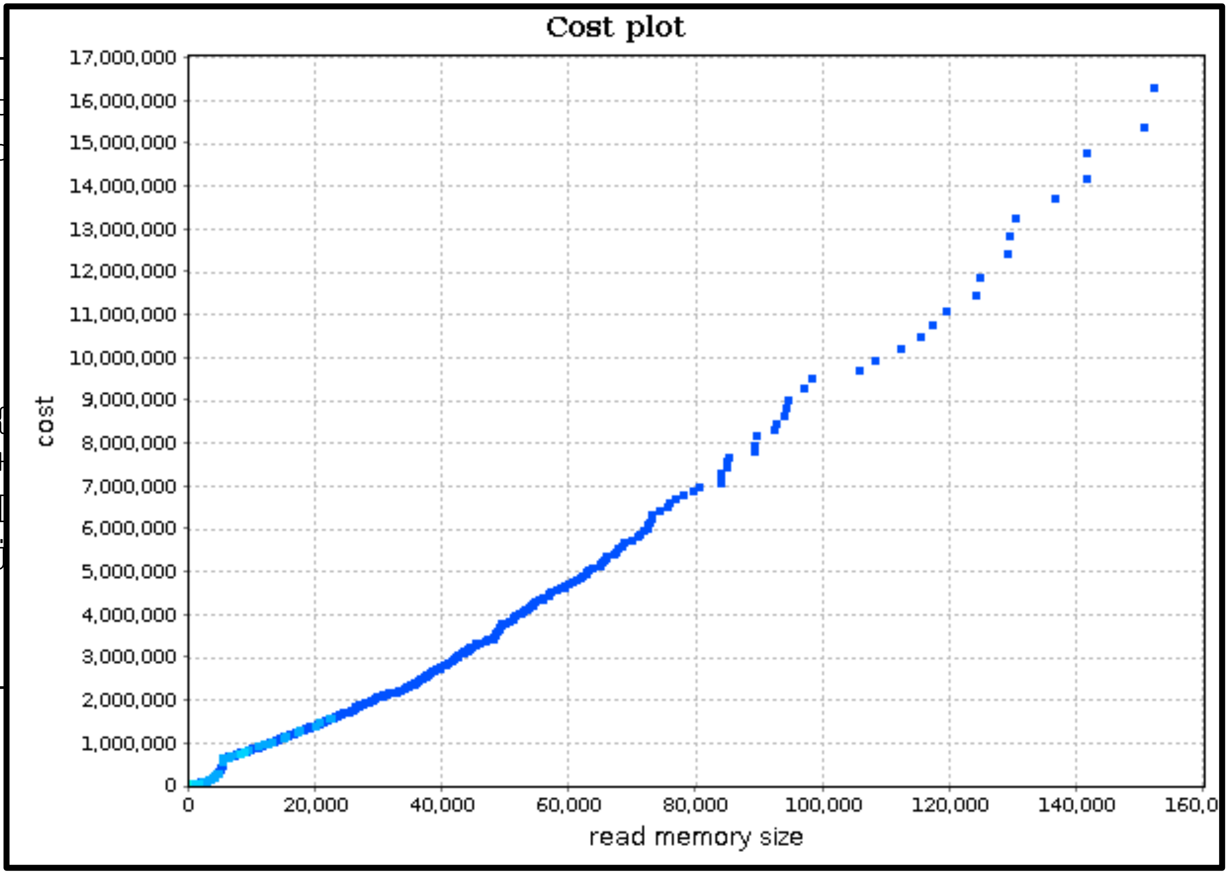


kernel_trmm

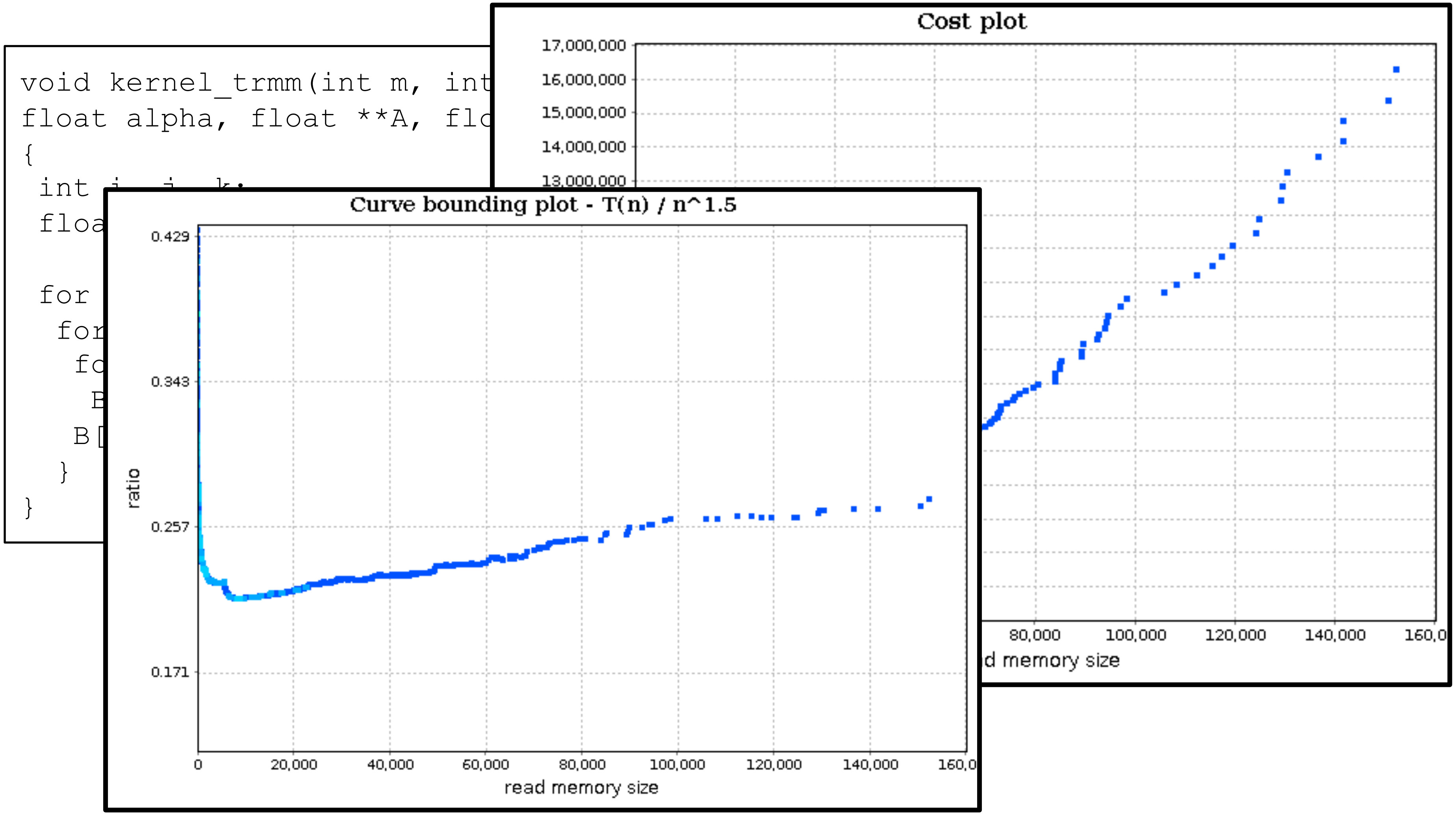
```
void kernel_trmm(int m, int n,
float alpha, float **A, float **B)
{
 int i, j, k;
 float temp;

 for (i = 0; i < m; i++)
  for (j = 0; j < n; j++) {
   for (k = i+1; k < m; k++)
    B[i][j] += A[k][i] * B[k][j];
   B[i][j] = alpha * B[i][j];
  }
}
```

kernel_trmm

```
void kernel_trmm(int m, int
float alpha, float **A, flo
{
 int i, j, k;
 float temp;

 for (i = 0; i < m; i++)
  for (j = 0; j < n; j++) {
   for (k = i+1; k < m; k++
    B[i][j] += A[k][i] * B[
   B[i][j] = alpha * B[i][j
  }
}
```



kernel_trmm



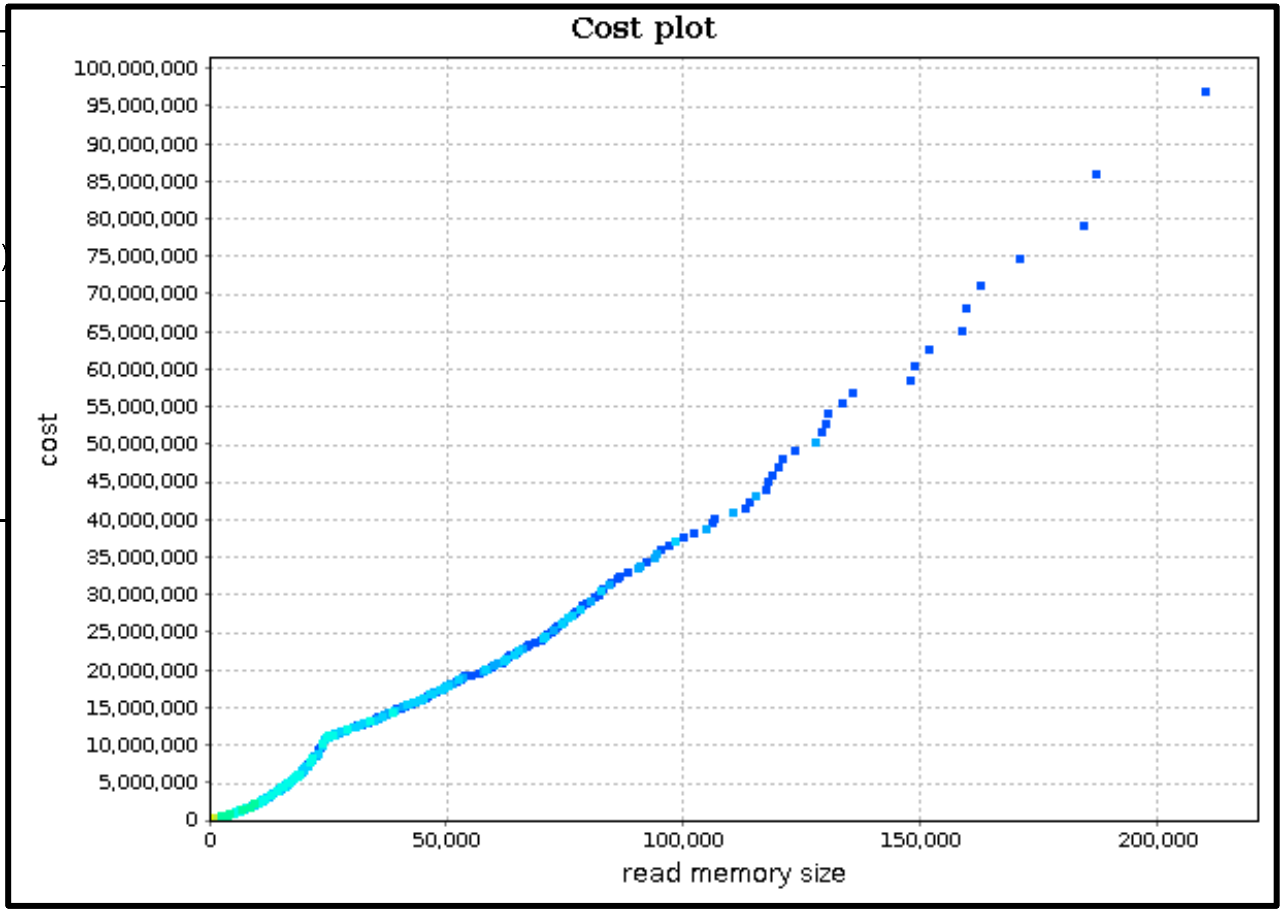
void kernel_trmm(int m, int
float alpha, float **A, flo
{
floa
for
for
fo
B
B[
}
}
Cost plot
17,000,000
16,000,000
15,000,000
14,000,000
13,000,000
80,000
100,000
120,000
140,000
160,0
d memory size
Curve bounding plot - T(n) / n^1.5
0.429
0.343
0.257
0.171
ratio
0
20,000
40,000
60,000
80,000
100,000
120,000
140,000
160,0
read memory size

Floyd-Warshall

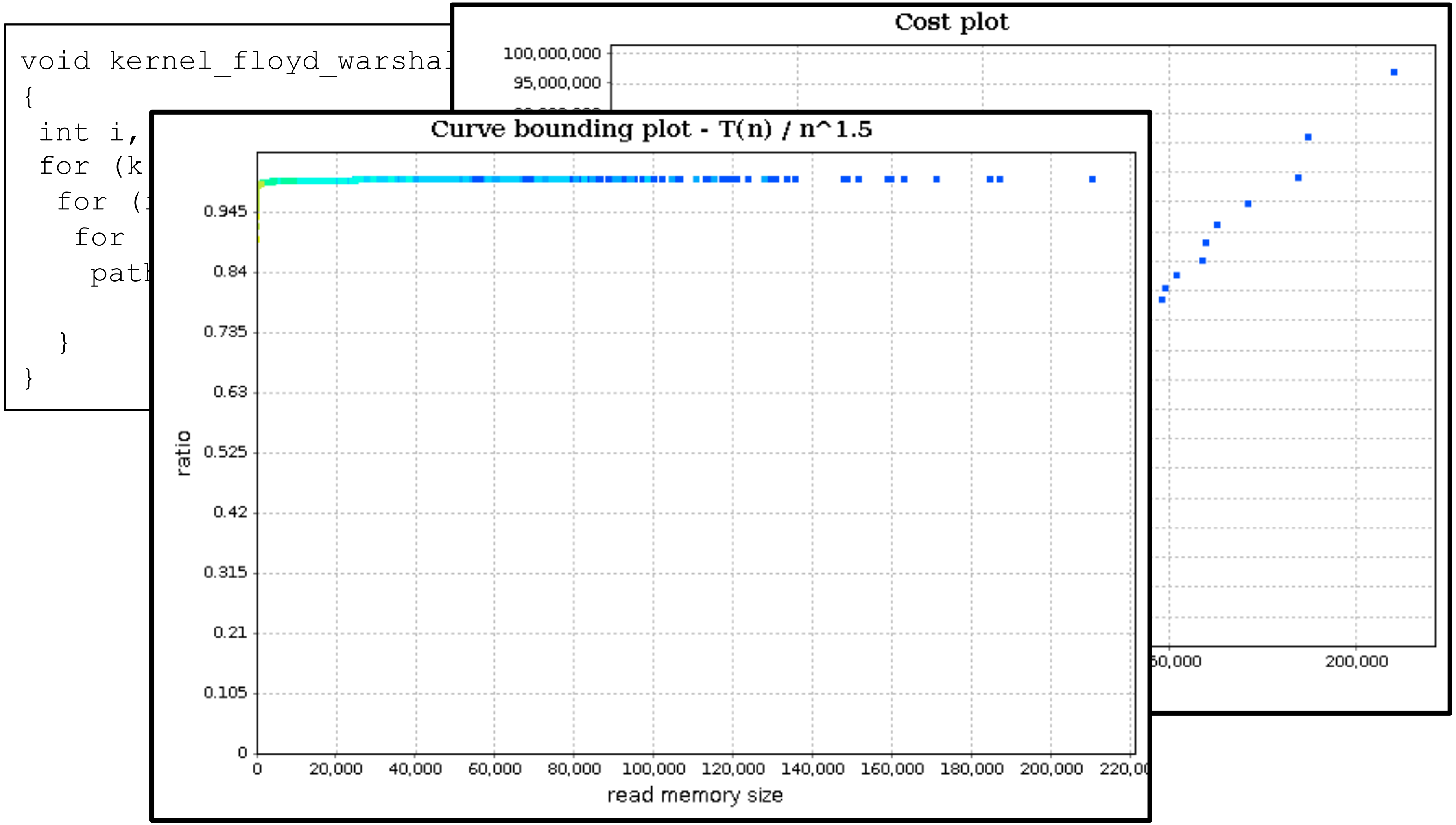
```
void kernel_floyd_warshall(int **path, int n)
{
 int i, j, k;
 for (k = 0; k < n; k++) {
  for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
    path[i][j] = path[i][j] < path[i][k] + path[k][j] ?
                 path[i][j] : path[i][k] + path[k][j];
  }
}
```

Floyd-Warshall

```
void kernel_floyd_warshal
{
 int i, j, k;
 for (k = 0; k < n; k++)
  for (i = 0; i < n; i++)
   for (j = 0; j < n; j+
    path[i][j] = path[i]
                 path[i]
  }
}
```



Floyd-Warshall



Generation of In-Bounds Inputs for Arrays in Memory-Unsafe Languages

A technique to test parts of a program without generating out-of-bounds accesses in arrays

https://github.com/maroar/griffin-TG

http://lac.dcc.ufmg.br/

fernando@dcc.ufmg.br


Generation of In-Bounds Inputs for Arrays in Memory-Unsafe Languages