Program analysis and verification - Final project

Oren Kishon, Asya Frumkin

November 2014

Introduction

This work is the final project of the course Program Analysis and Verification by Noam Rinetzky, 2014.

The project's purpose was to create a static analysis software tool to detect index-out-of-bound errors in C programs using methods of abstract interpretation.

The analysis tool takes as input a C function and anlayzes it using a fixed point iteration algorithm on the blocks of its control flow graph. The analysis is done by abstracting variables' values to the intervals domain. Propagation through the blocks is done by Join and Meet operations on the abstract values.

Software tools

This new software tool is based on two software libraries.

- Clang[1]: The LLVM compiler's C language frontend. Clang has the ability of parsing a C program and creating its control flow graph and its abstract syntax tree. It also provides API for iterating over these CFG and AST nodes, and each statement and expression inside these nodes (or blocks). It does not, however, has the feature of fixed-point iteration analysis based on abstract values, and this is a needed feature we have found requests for over the internet. Clang does have the feature of analysis based on symbolic execution (referred to as "Path sensitive analysis") which goes down path conditions of a program, but this algorithm was not what we desired.
- Apron[2]: A software library for numerical analysis in abstract interpretation. Used for transforming variable values to the intervals domain and applying linear expression constraints on them and the operations join and meet.

A short manual on using our tool is in the appendix.

Figure 1: Loop example input function

```
static void loop_example(void)
{
    int x;
    x = 7;

    while (x < 1000)
        ++x;

    if (x != 1000)
        printf("Unable to prove x == 1000!\n");
}
```

CFG creation and the fixed point algorithm

The first thing done by the program is to create the CFG data structure of the input function. This is done by Clang. Clang can also actually run a graphic program in the OS to plot the CFG. See figure 1 and figure 2 for an example of a program and its CFG.

The blocks of the CFG are iterated in a worklist algorithm (referred to also as chaotic iteration) implemented according to the one learned in class (see figure 3). The implementation can bee seen in figure 4. It is very simple: At the beginning all of the blocks are in the queue (with all abstract values initialized to bottom). Popped out blocks are processed by the transfer functions. If their values have changed, then their successors in the CFG are re-queued in the queue. This goes on until no more changes occure. If an error has been found, in our case - an put-of-bound index error, the iteration is aborted.

Processing done on a CFG block

The Clang block data structure holds the block's statements list and pointers to the neighboring blocks (predecessors and successors). To this existing data structure we have added an "Apron context" which carries the array of abstract values of a block. The list of tracked variables is common between all blocks, but each block assigns different abstract values to them. Beside this abstract value context, each block also holds a list of entry values. This list is actually updated by the predecessors of the block when they are finished being processed. At the beggining of a run on a block, its entry value is computed by applying a join on all of this list's members. In the same way, the block will update its successors' entry values when it finishes each run, according to the condition in its termination statement. To sum up the algorithm of a single run on a block:

1. Compute entry value by "join"ing all values in predecessors values list
2. Mutate this abstract value by the transfer functions of the block's statemets (the assignments)

Figure 2: Loop example CFG

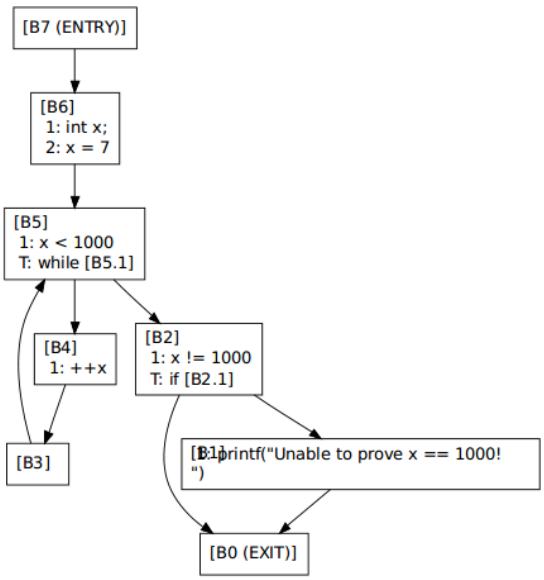


Figure 3: Worklist algorithm from class

```
for i:=1 to n do
   X[i] := ⊥
WL = {1,...,n}
while WL ≠ ∅ do
   j := pop WL // choose index non-deterministically
   N := F[i](X)
   if N ≠ X[i] then
      X[i] := N
      add all the indexes that directly depend on i to WL
      (X[j] depends on X[i] if F[j] contains X[i])
return X
```

Figure 4: Worklist algorithm imlpementation

```
while (const clang::CFGBlock *block = worklist.dequeue()) {
    // Did the block change?
    bool changed = blockAnalysis.runOnBlock(block);

    if (blockAnalysis.foundError()) {
        goto Error;
    }

    if (changed) {
        worklist.enqueueSuccessors(block);
    }
}
printf("Fixed point reached\n");
```

Figure 5: Run on a single block

```
bool runOnBlock(const clang::CFGBlock *block) {
    BlockApronContext *blkApronCtx = block2Ctx[block];

    if (!blkApronCtx->updateEntryValue())
        return false;
    printf("Some abstract values changed\n");

    // Init and apply the transfer function.
    TransferFunctions tf(blkApronCtx);
    for (clang::CFGBlock::const_iterator I = block->begin(),
            E = block->end(); I != E; ++I) {
        if (clang::Optional<clang::CFGStmt> cs =
                I->getAs<clang::CFGStmt>()) {

            tf.Visit(const_cast<clang::Stmt*>(cs->getStmt()));

            if (tf.foundError()) {
                error = true;
                break;
            }
        }
    }

    blkApronCtx->updateSuccessors();

    return true;
}
```

3. Pass updated values to successor blocks' entry lists.

The code for running on a single block is in figure 5.

Transfer functions

The transfer functions are implemented using a visitor design pattern. In every assignment statement, the abstract value of the assignee variable is updated using Apron's API. Assignment in Apron is done by creating a linear expression combined of other variables and constant values. A code snippet of the assignment function is in figure 6.

Figure 6: Assignment using Apron

```
/* x := a*y + c */
void assignLinearExpression(char *x, int a, char *y, int c) {
  // ap_linexpr1_make() destroys contents of *var
    ap_linexpr1_t expr = ap_linexpr1_make(env, AP_LINEXPR_SPARSE, 1);
    ap_linexpr1_set_list(&expr,
                         AP_COEFF_S_INT, a, y,
                         AP_CST_S_INT, c,
                         AP_END);

    abst = ap_abstract1_assign_linexpr(man, true, &abst, x, &expr, NULL);
    ap_linexpr1_clear(&expr);
}
```

Figure 7: Assume transformers

- $[\![\textbf{assume}\ x{=}c]\!]^\# S = S \sqcap \{x{\geq}c, x{\leq}c\}$
- $[\![\textbf{assume}\ x{<}c]\!]^\# S = S \sqcap \{x{\leq}c\text{-}1\}$
- $[\![\textbf{assume}\ x{=}y]\!]^\# S = S \sqcap \{x{\geq}\mathsf{lb}(S,y), x{\leq}\mathsf{ub}(S,y)\}$
- $[\![\textbf{assume}\ x{\neq}c]\!]^\# S = (S \sqcap \{x{\leq}c\text{-}1\}) \sqcup (S \sqcap \{x{\geq}c{+}1\})$

Branching

when ever a block is terminated in a branch statement (If, while, etc) it means each of its successors should receive the block exit value after being transformed by an "assume" transformation (see figure 7). A new abstract value array is created for each branch (one for the "then" and one for the "else", see figure 8) and each of these values is updated in its matching successor block's entry values list. The "assume" transformation is actually a meet operation with the corresponding constraint of the branch. The implementation of these transformers in Apron is by creating a constraint and applying a meet operation between the abstract value and the constraint. See figure 9 for the implementation, including the special case of the "!=" condition.

Array index out-of-bound checking

When applying the transfer functions on assignments, the analysis tool also checks whether an array subscript expression (For example: A[i]) could be using an index with a value out of bounds of the array allowed indexes (below 0 or more than the size). In this case, the potential error is reported to output and the tool aborts. As far as our implementation goes, this check can be performed on arrays with constant size (for example, declared as "int A[32]") but with the

Figure 8: Creating entry values for next blocks using constraints

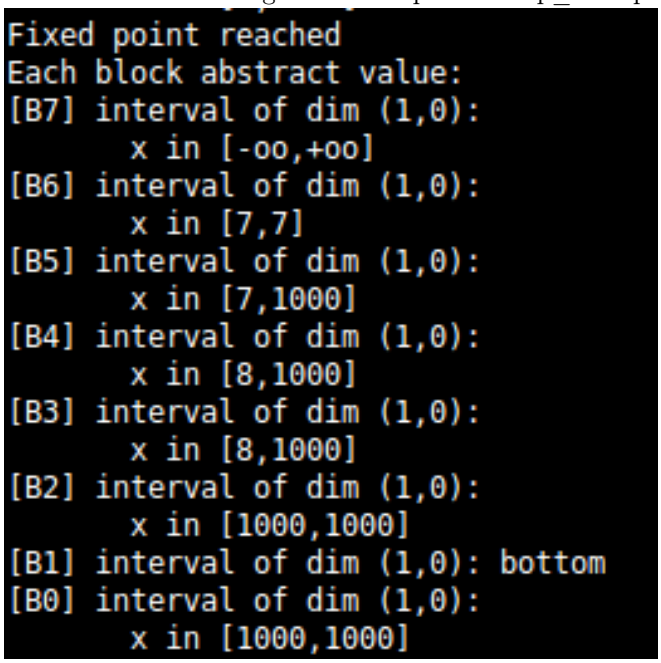
```
switch (BO->getOpcode()) {
case clang::BO_LT:
  absThen = meet_constraint(x, LESS, y, c);
  absElse = meet_constraint(x, GREATER_EQUAL, y, c);
  break;
case clang::BO_NE:
  absThen = meet_constraint(x, NOT_EQUAL, y, c);
  absElse = meet_constraint(x, EQUAL, y, c);
  break;
default:
  printf("Can only handle ops: <, !=\n");
  exit(1);
}
```

Figure 9: Implementation of assume transformers using Apron

```
ap_abstract1_t meet_constraint(char *var, compare_t op, char *y, int c) {
    ap_lincons1_t cons;
    ap_abstract1_t a, b;

    switch(op) {
    case EQUAL:
    case GREATER_EQUAL:
    case LESS:
        cons = create_linexp_constraint(var, op, y, c);
        break;
    case NOT_EQUAL:
        // Instead of using != we do: join(meet(ABS, x<c), meet(ABS, x>c)).
        // If (x=[c, c]) then the result will be bottom.
        a = meet_constraint(var, LESS, y, c);
        b = meet_constraint(var, GREATER_EQUAL, y, c + 1);
        return ap_abstract1_join(man, false, &a, &b);
    default:
        exit(1);
    }
```

Figure 10: Output for loop_example()



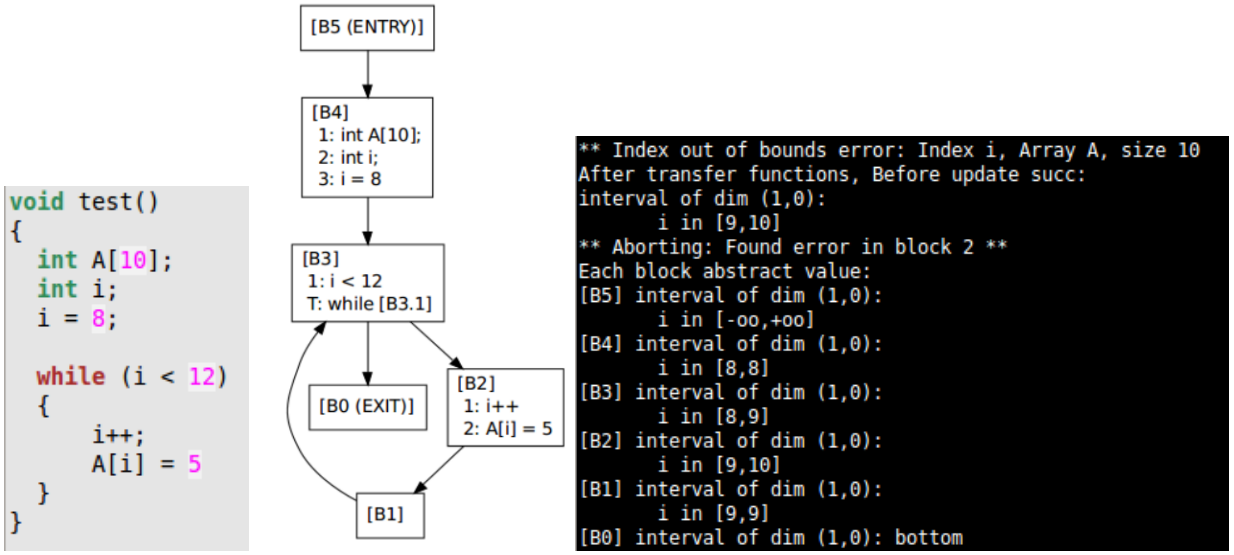
checked index being an abstract value. Extending it to dynamic array sizes is a matter of further development over the Clang AST code. The implementation in Apron of the check of the abstract index is similiar to the other linear-expression-constraints operations described in other sections. The one unique thing here is the use of the Apron function "ap_abstract1_sat_lincons(...)" which takes an abstract value and a constraint, and checks if the value satisfies the constraint.

Results

Figure 10 shows the result output of the loop_example() function described in the previous sections. This function has no errors, so the iteration reaches a fixed point. This output should be viewed along with the CFG of the function. It can be observed that the exit block, [B0], has the correct value of 1000, and the error block, [B1], has the value of bottom because its not reachable.

Next is an example of a program with an array index error. In figure 11, in the analysis output , it can be seen that the analysis has discovered the error when the index 'i' has reached the possible value of 10, the smallest un-allowed value, and aborted the analysis with a proper report to output.

Figure 11: Array test function: Code, CFG and analysis output



Summary

In this project we have implemented a chaotic iteration algorithm and static analysis using abstract interpretation as we learned them in the course. We have invested much in implementing these algorithms for analyzing C programs by the use of "melting" together the Clang and Apron libraries. On top of this implementation we have showed an implementation of index-out-of-bound error checking. We have intended to create a much more sophisticated array index analyzer but we have discovered that the implementation of the infrastructure for the analysis itself is a very hard task on its own (The tool has almost 1000 lines of code, completely new). But it is also apperent that once this infrastructure works, creating anlyzers on top of it is much more easy.

Appendix - Running the tool

- The compilation instructions are in "README.md"
- The compilation is by running the script ./build.sh
- The execution is by running: "./worklist <C file> <CFG creation param> <function name>
  - C file: the name of the input C file
  - CFG creation param: 0 for no CFG output, 1 for textual CFG putput to standart output and 2 for graphical CFG dot file. For this, the program graphviz must be installed in the system.

  - functin name: The function inside the file being analyzed.

- Execution example:
  - ./worklist test_array.c 0 test

- Notice that some input example programs are included with the source code.

References

[1] clang: a C language family frontend for LLVM http://clang.llvm.org/

[2] APRON numerical abstract domain library http://apron.cri.ensmp.fr/library/