CS 201: Compiler Construction

Project Report – Liveness Analysis

Lokesh Koppaka, Abhilash Sunkam
{lkopp001, asunk001}@ucr.edu

Introduction

The project is designed to implement Liveness Analysis across CFG. The report provides the implementation details of Liveness Analysis using Soot. Additionally, the report holds limitations of the implementation which are quite essential to point out for further scope of enhancements

Liveness Analysis

Liveness Analysis is data flow analysis procedure, essentially used to derive the liveness of the variables across various basic blocks in a CFG. Liveness Analysis has its applications across various analysis such as Register allocation, Construction of SSA, Useless – store elimination etc.. In liveness Analysis for each basic block we compute the LIVEOUT by using the following formulation in an iterative approach

$$LIVEOUT(b) = \cup x \in SUCC(b)[\, LIVEOUT(x) - VARKILL(x) + UEEXPR(x)]$$

Fig 1.0 Liveness Formula

The following table provides the details regarding the above formula parameters

$\boldsymbol{LIVEOUT(b)}$	LIVEOUT for current basic block b
$\cup \boldsymbol{x} \in \boldsymbol{SUCC(b)}$	Union Operation across all the successors of the current basic block b
$\boldsymbol{LIVEOUT(x)}$	LIVEOUT for successor basic block x
$\boldsymbol{VARKILL(x)}$	Variables killed in the basic block x – A variable "v" is killed in a basic block x if it is redefined in basic block x
$\boldsymbol{UEEXPR(x)}$	Upward exposed in the basic block x - A variable "v" is upward exposed if it is used in x before any redefinition in x

Table 1.0 Formula Parameters

The below table demonstrates the data flow analysis parameters for Liveness Analysis

Domain	Variables
Direction	Backward
Joining/Meet	Union
Initialization	Empty

Implementation

Liveness is implements using Soot in Java. The implementation holds the following essential classes.

- **Main.java**
  Main class holds the logic for user preference, performs soot configurations and initiates Liveness Analysis
- **Liveness.java**
  Liveness is the core class holding the logic of reading all the files, performing Liveness Analysis and generating LIVEOUT for each basic block across all the provided class methods.
  The Basic Iterative algorithm demonstrated in the class presentation of Liveness is used to implement Liveness Analysis.

```
for i = 1 to N     /* N: total number of blocks */
  LIVEOUT(i) ← ?
while (flag)
     flag ← false
    for i = 1 to N
        recompute LIVEOUT(i)
        if LIVEOUT(i) changed then
            flag ← true
```

Fig 2.0 Basic Iterative Approach

The following demonstrates various steps followed in Liveness to generate LIVEOUT for each basic block

1. Compute VARKILL and UEVAR for each basic block. The following snippet reads in all the units in the basic block, retrieves the left operand adds it to VARKILL, retrieves the right operands add them to UEVAR if the operand is not in VARKILL.

```
// Logic to compute UEVAR and VARKILL
while(blockIt.hasNext()) {
    // For each unit in the basic block
    Unit unit = blockIt.next();
    for (ValueBox useBox: unit.getUseBoxes()) {
        //Logic to add rightSide Operands to UEVAR if not present in VARKILL
        if (useBox.getValue() instanceof Local) {
            Local useLocal = (Local) useBox.getValue();
            if(!VARKILL.contains(useLocal.getName())) {
                UEEXP.add(useLocal.getName());
            }
        }
    }
    for (ValueBox defBox: unit.getDefBoxes()) {
        // Logic to add leftSide Operands to VARKILL
        if (defBox.getValue() instanceof Local) {
            Local defLocal = (Local) defBox.getValue();
            VARKILL.add(defLocal.getName());
        }
    }
}
```

Note: The code also takes care of branching statements as well.

2. UEVAR, VARKILL computed above are maintained in a HashMap where key being the basic block, value being a list holding the VARKILL at position 0 and UEVAR at position 1

3. Once UEVAR, VARKILL are computed basic iterative solution is used to implement LIVEOUT. The code iterates through each basic block, gets all the successors of the current basic block and compute the LIVEOUT of the current basic block as discussed in the above

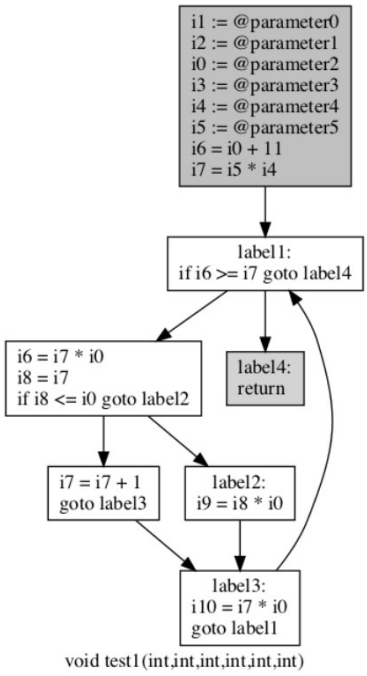
"Liveness Analysis" section. The iteration is continued till there exists no difference in the LIVEOUT among all the basic blocks from the previous iteration to the current iteration

```
// Iterative Logic to compute LIVEOUT
boolean flag = true;
while(flag) {
    flag = false;
    // for each block get the current LIVEOUT
    for(Block currentBlock : livenessMap.keySet()) {
        FlowSet<String> LIVEOUT = new ArraySparseSet<String>();
        List<Block> successors  = currentBlock.getSuccs();
        // Logic to update LIVEOUT = For all x in successor Union(LIVEOUT(x) - VARKILL(x) + UPEXP(x))
        for(Block block : successors) {
            FlowSet<String> succBlock2 = new ArraySparseSet<String>();
            FlowSet<String> succBlock = livenessMap.get(block);
            FlowSet<String> innerList = new ArraySparseSet<String>();
            succBlock.difference(blockMap.get(block).get(0), succBlock2);
            succBlock2.union(blockMap.get(block).get(1),innerList);
            LIVEOUT.union(innerList, LIVEOUT);
        }
        FlowSet<String> unionResult = new ArraySparseSet<String>();
        LIVEOUT.union(livenessMap.get(currentBlock),unionResult);
        // Logic to update back the latest LIVEOUT if it is changed from the previous iteration
        if(unionResult.size() != livenessMap.get(currentBlock).size()){
            livenessMap.put(currentBlock, LIVEOUT);
            flag = true;
        }
    }
}
```

How to Run?

The details about running the program is given in the README.txt file

Sample Input



Sample Output

```
Performing necessary configuration...
Completed configurations...
Peforming liveness analysis across all the methods....
********** test1 ***********
-------------------------------
|No of Basic Blocks = 7|
-------------------------------
Block 6:
[preds: 1 ] [succs: ]
return;

VARKILL = {} UEVAR = {}
-------------LIVE OUT ------------------
LIVE OUT = {}
****************************************************************
Block 5:
[preds: 3 4 ] [succs: 1 ]
i10 = i7 * i0;
goto [?= (branch)];

VARKILL = {i10} UEVAR = {i7, i0}
-------------LIVE OUT ------------------
LIVE OUT = {i0, i7, i6}
****************************************************************
Block 4:
[preds: 2 ] [succs: 5 ]
i9 = i8 * i0;

VARKILL = {i9} UEVAR = {i8, i0}
-------------LIVE OUT ------------------
LIVE OUT = {i6, i7, i0}
****************************************************************
Block 3:
[preds: 2 ] [succs: 5 ]
i7 = i7 + 1;
goto [?= i10 = i7 * i0];

VARKILL = {i7} UEVAR = {i7}
-------------LIVE OUT ------------------
LIVE OUT = {i6, i7, i0}
****************************************************************
Block 2:
[preds: 1 ] [succs: 3 4 ]
i6 = i7 * i0;
i8 = i7;
if i8 <= i0 goto i9 = i8 * i0;

VARKILL = {i6, i8} UEVAR = {i7, i0}
-------------LIVE OUT ------------------
LIVE OUT = {i6, i0, i7, i8}
****************************************************************
Block 1:
[preds: 0 5 ] [succs: 2 6 ]
if i6 >= i7 goto return;

VARKILL = {} UEVAR = {i6, i7}
-------------LIVE OUT ------------------
LIVE OUT = {i0, i7}
****************************************************************
Block 0:
[preds: ] [succs: 1 ]
i1 := @parameter0: int;
i2 := @parameter1: int;
i0 := @parameter2: int;
i3 := @parameter3: int;
i4 := @parameter4: int;
i5 := @parameter5: int;
i6 = i0 + 11;
i7 = i5 * i4;

VARKILL = {i1, i2, i0, i3, i4, i5, i6, i7} UEVAR = {}
-------------LIVE OUT ------------------
LIVE OUT = {i0, i7, i6}
****************************************************************
```

Limitations

1. Handles only simple +, -, *, / arithmetic expressions
2. Handles only simple branching statements
3. Limited to CFG Size

Future Scope

The implementation can further be extended to implement the above limitations

References

[1] Git hub link provided
[2] https://github.com/Sable/soot/wiki/Tutorials
[3] Liveness Analysis class presentation