Advanced T pics in

C compilers

DFA

Simone Campanoni
simone.campanoni@northwestern.edu
Outline

• DFA (summary from 323)

• Data Flow Engine in NOELLE

• Data Flow Analyses available in NOELLE
The need for DFAs

• We constantly need to improve programs (e.g., speed, energy efficiency, memory requirements)
• We constantly need to identify opportunities
• After having found an opportunity (e.g., propagating constants), you need to ask yourself:
  • What do I need to know to take advantage of this opportunity? (e.g., I need to know the possible values a given variable might have at a given point in the program)
  • How can I automatically compute this information? Often the solution relies on understanding how data flows through the code. This is often done by designing ad-hoc DFAs
New transformations and analyses

• New transformations (often) need to understand specific and new code properties related to how data might change through the code
  • So we need to know how to design a new data flow analysis that identifies these new code properties

• Generic recipe
  Data flow analysis (DFA): traverse the CFGs collecting information about what may happen at run time (Conservative approximation)
  Transformation: Modify the code based on the result of data flow analysis (Correctness guaranteed by the conservative approximation of DFA)
New transformations and analyses

- Generic recipe
  - **Data flow analysis (DFA):**
    traverse the CFGs collecting information about what may happen at run time (Conservative approximation)
  - **Transformation:**
    Modify the code based on the result of data flow analysis (Correctness guaranteed by the conservative approximation of DFA)

What are the possible values \( b \) can have at run time?
Static program vs. dynamic execution

- **Static:**
  Finite program

- **Dynamic:**
  Can have infinitely many possible control flows

- **Data flow analysis abstraction:**
  For each point in a program:
  combine information about all possible run-time instances of the same program point.

**What are the possible values of b?**

Data flow analysis (DFA):
traverse the CFGs collecting information about what may happen at run time
(Conservative approximation)
Data-flow expressed in CFG

**Data-flow value:**
set of all possible program states that can be observed at a given program point

- e.g., all definitions in the program that might have been executed before that point

**Data-flow analysis**
computes IN and OUT sets by computing the DFA-specific transfer functions
Transfer functions

• Let $i$ be an instruction: $\text{IN}[i]$ and $\text{OUT}[i]$ are the set of data-flow values before and after the instruction $i$ of a program

• A transfer function $f_s$ relates the data-flow values before and after an instruction $i$

• In a forward data-flow problem
  \[
  \text{OUT}[i] = f_s(\text{IN}[i])
  \]

• In a backward data-flow problem
  \[
  \text{IN}[i] = f_s(\text{OUT}[i])
  \]

$f_s$ is DFA-specific
Transfer function internals: \( Y[ i ] = fs ( X[ i ] ) \)

- It relies on information that reaches \( i \)

- It transforms such information to propagate the result to the rest of the CFG

  \[ \text{\( GEN[i] = \text{data flow value added by } i \)} \]
  \[ \text{\( KILL[i] = \text{data flow value removed because of } i \)} \]

- To do so, it relies on information specific to \( i \)
  - Encoded in \( GEN[i], KILL[i] \)
    - \( fs \) uses \( GEN[i] \) and \( KILL[i] \) to compute its output

- \( GEN[i] \) and \( KILL[i] \) are DFA-specific and (typically) data/control flow independent!
DFA steps

1) Define the DFA-specific sets GEN[i] and KILL[i], for all i

2) Implement the DFA-specific transfer function $f_s$

3) Compute all $IN[i]$ and $OUT[i]$ following a DFA-generic algorithm
   
   $OUT[i] = f_s(IN[i])$
   $IN[i] = f_s(OUT[i])$
Forward DFA

for (each instruction \( i \)) \( \text{IN}[i] = \text{OUT}[i] = \{ \} \);

do {
    for (each instruction \( i \)) {
        \( \text{IN}[i] = fs_p \) a predecessor of \( i \) (\( \text{OUT}[p] \))
        \( \text{OUT}[i] = fs(\text{IN}[i]) \)
    }
} while (changes to any \( \text{OUT} \) occur)
for (each instruction \(i\)) \(\text{IN}[i] = \text{OUT}[i] = \{ \}\);

do {
  for (each instruction \(i\)) {
    \text{OUT}[i] = fs_{s\text{ a successor of } i} (\text{IN}[s])
    \text{IN}[i] = fs(\text{OUT}[i])
  }
} while (changes to any IN occur)
Optimization example: work list

\(\text{OUT}[\text{ENTRY}] = \{ \};\)
for (each basic block \(B\) other than \(\text{ENTRY}\)) \(\text{OUT}[B] = \{ \};\)
workList = all basic blocks
while (workList isn’t empty)
  \(B = \text{pick and remove a block from workList}\)
  \(\text{oldOUT} = \text{OUT}[B]\)
  \(\text{IN}[B] = \bigcup_{p \text{ a predecessor of } B} \text{OUT}[p];\)
  \(\text{OUT}[B] = \text{GEN}[B] \cup (\text{IN}[B] - \text{KILL}[B]);\)
  if (oldOut != OUT[B]) workList = workList U {all successors of \(B\)}
}
Outline

• DFA (summary from 323)

• Data Flow Engine in NOELLE

• Data Flow Analyses available in NOELLE
The need for a data flow engine

• Implementing a data flow analysis that scales well with the number of instructions takes time and efforts
• The typical required optimizations (see 323) are DFA-agnostic
• A data-flow engine, therefore, can be built once and used by many data-flow analyses
• LLVM does not provide a data-flow engine
• NOELLE provides a data-flow engine to accelerate the development of data-flow analyses accelerating therefore research
Let’s build our first DFA with NOELLE
Normalize the code

Code must be normalized before you use NOELLE

• noelle-norm MYIR.bc –o IR.bc
  or
• noelle-simplification MYIR.bc –o IR.bc
Fetching the data flow engine

```cpp
/*
 * Fetch NOELLE
 */
auto& noelle = getAnalysis<Noelle>();
```

```cpp
/*
 * Fetch the data flow engine.
 */
auto dfe = noelle.getDataFlowEngine();
```
Using the data-flow engine

/*
 * Fetch the entry point.
 */
auto fm = noelle.getFunctionsManager();
auto mainF = fm->getEntryFunction();

auto customDfr = dfe.applyBackward(
    mainF,
    computeGEN,
    computeKILL,
    computeIN,
    computeOUT
);

It includes
the final IN and OUT for all instructions

void (Instruction *, DataFlowResult *)

void (std::set<Value *> & IN,
     Instruction * inst,
     DataFlowResult * df)

New DFA example

**Goal:** identify the load instructions that could use the value loaded from memory by a given load instruction for all load instructions

Correct (and conservative) solution:
- Backward DFA
- **GEN**\[i\] = \{i\} if i is a load instruction, {} otherwise
- **KILL**\[i\] = {}
- **OUT**\[i\] = \(\bigcup\)\(_{s = \text{successors}(i)}\) **IN**\[s\]
- **IN**\[i\] = **GEN**\[i\] \(\cup\) **OUT**\[i\]
New DFA example

- GEN[i] = {i} if i is a load instruction, {} otherwise

```cpp
auto computeGEN = [](Instruction *i, DataFlowResult *df) {
    if (!isa<LoadInst>(i)){
        return ;
    }
    auto& gen = df->GEN(i);
    gen.insert(i);
    return ;
};
```
New DFA example

- $KILL[i] = \{}$

```cpp
auto computeKILL = [] (Instruction *, DataFlowResult *) {
    return ;
};
```
New DFA example

\[ \text{OUT}[i] = \bigcup_{s = \text{successors}(i)} \text{IN}[s] \]
New DFA example

- \( \text{IN}[i] = \text{GEN}[i] \cup \text{OUT}[i] \)

```cpp
auto computeIN = [] (std::set<Value *> & IN, Instruction *inst, DataFlowResult *df) {
    auto& genI = df->GEN(inst);
    auto& outI = df->OUT(inst);
    IN.insert(outI.begin(), outI.end());
    IN.insert(genI.begin(), genI.end());
    return ;
};
```
Computing DFA result

```cpp
auto customDfa = dfe.applyBackward(
  mainF,
  computeGEN,
  computeKILL,
  computeIN,
  computeOUT
);
```
Using DFA result

```c++
for (auto inst : instructions(mainF)){
    if (!isa<LoadInst>(inst)){
        continue ;
    }
    auto insts = customDfr->OUT(inst);
    errs() << " Next are the " << insts.size() << " instructions ";
    errs() << "that could read the value loaded by " << *inst << "\n";
    for (auto possibleInst : insts){
        errs() << " " << *possibleInst << "\n";
    }
}
```
Normalize the code

Code must be normalized before you use NOELLE

- noelle-norm MYIR.bc –o IR.bc
  or
- noelle-simplification MYIR.bc –o IR.bc
Outline

• DFA (summary from 323)

• Data Flow Engine in NOELLE

• Data Flow Analyses available in NOELLE
Running available data flow analyses

```cpp
/*
 * Fetch NOELLE
 */
auto& noelle = getAnalysis<Noelle>();

auto dfa = noelle.getDataFlowAnalyses();

/*
 * Fetch the entry point.
 */
auto fm = noelle.getFunctionsManager();
auto mainF = fm->getEntryFunction();

auto dfr = dfa.runReachableAnalysis(mainF);```