CS42/413
Introduction to Compilers
Radu Rugina
Lecture 37: Advanced Analyses
05 May 04

Dataflow Analysis
• Builds the CFG, iterate over basic blocks
• Compute information at each program point
  – E.g. constants, live variables, etc.
• Discussed: intra-procedural analysis
  – considers only the computation in the current procedure
• At function calls, assume worst case
  – Live variables: all globals/fields live before the call
  – Constant folding: globals/fields not constant after call

Inter-Procedural Analysis
• Precisely analyze interactions between functions/methods
• Same as dataflow analysis, but at each call analyze take into
  account the computation in the invoked function
• Examples: inter-procedural constant folding, inter-procedural
  register allocation, etc.

Issues
• Obtain a stack of analyses which corresponds to the
  execution stack of the program
• Analysis must bind actual parameters to formals
  before analyzing the callee
  – n = 2; m = 3;
• Another issue: different functions/methods have
  different analysis domains
  – E.g. for live variables, analysis domain includes set of
    variables local to the current function
  – Must change the analysis domain when analysis moves
    from caller to callee

Multiple Call Sites
• Another aspect: a function may be invoked from
  multiple call sites
• At different call sites, the analysis is different
  – Input context = analysis information at call site
• Hence, must re-analyze function in each context

Analysis Contexts
• The analysis of a function yields an analysis context
  which is a pair of:
  – an input context: the dataflow information at the entry (or
    exit) of the function
  – and a corresponding analysis result: the information at
    the exit (or entry) of the function, plus the return value
• Useful for memoization: whenever the information
  at a call site matches some input context, can reuse
  analysis result
Example
- Consider inter-procedural constant folding for the following program:

```c
int a;
void h() {
    int t;
    int b;
    scanf("%d", &b);
    a = 1;
    b = f(2,f(h,3));
}
```

- What are the contexts for function f?
- What is the value of b at the end of function h?

Recursion
- So far, analysis of recursive procedures doesn’t terminate
- Analysis creates an unbounded number of analysis contexts
- Need a fixed point algorithm
  - Similar to analysis of loops in dataflow analysis
- Approach: for each analysis context, keep a current best analysis result
  - Initialize current best to top
  - At recursive call sites use current result
  - At return: if result has changed, re-analyze function

Indirect Calls
- Problem: calls for which the invoked function cannot be precisely determined at compile time
  - Function pointers in C/C++
  - Dynamically dispatched functions in Java/C++
- Approach:
  - Analyze all possibly invoked functions
  - Then merge all of the results together
- To be precise, must accurately compute the possible targets of each indirect call
  - Function pointers: need points-to information
  - Virtual functions: need class hierarchy information

Exponential Blow-up
- Problem: the number of procedure calls in a program may be exponential in the program size:

```c
int f() { g(); g(); }
int g() { h(); h(); }
int h() { k(); k(); }
```
- Call graph = graph describing the call structure
  - Nodes are functions, edges are call sites
  - Functions close to the leaves get executed many times
- Similarly, inter-procedural analysis may re-analyze functions many times; hence the analysis becomes expensive

Context-Insensitive Analysis
- So far: different analyses of a function for different input context (i.e., context-sensitive analysis)
- Alternative: context-insensitive analysis
  - Merge together all of the input contexts
  - Get a conservative input context
  - Analyze function for that input
  - Use analysis result for all of the call sites
- Less precise because it doesn’t distinguish between different input contexts at different call sites
  - But more efficient: analyzes functions fewer times

Unrealizable Paths
- Source of imprecision: information may flow from one call site to another
- The results models execution paths that don’t follow the stack discipline, i.e., unrealizable paths

```
g(m) -- f(n,p) -- h()
       |         |
       v         v
       f(2,3)    f(x,5)
```

CS 412/413 Spring 2004 Introduction to Compilers
Flow-Sensitivity

- Dataflow analysis follows the control flow in the program to compute the result; hence, it is flow-sensitive.
- Alternative: flow-insensitive analysis
  - Ignores the control flow!
  - Regards a program as a collection of statements
  - Assumes that statements can be executed multiple times, in any order
  - More efficient, less precise than flow-sensitive
- Similarity: type information is essentially flow insensitive
  - To check types of variables, just check assignments
  - Okay if assignments executed in a different order

Flow-Insensitive Analysis

- Since the control flow is ignored, it is meaningless to compute a result per program point.
- Instead, compute a single result valid for the whole program!
- General approach:
  - Derive constraints for each statement
  - Solve the system of constraints
- Example: points-to analysis — for each pointer variable v, want to compute the set \( \text{Ptr}(v) \) of possible targets of v.

Algorithm 1

- Steensgaard algorithm:
  - for each variable v, compute a "pointer type" \( \tau_v = \text{Type}(v) \)
  - Then \( \text{Ptr}(v) = \{ u \mid \tau_v = \tau_u \text{ and } \tau_u = \text{Type}(u) \} \)
- To compute types, use a standard type inference algorithm based on unification
- Generated constraints:
  - \( x = \&y \quad : \quad \tau_x = \tau_y \)
  - \( x = y \quad : \quad \tau_x = \tau_y \)
  - \( x = \#y \quad : \quad \#\tau_x = \#\tau_y \)
  - \( \#x = y \quad : \quad \#\tau_x = \#\tau_y \)

Algorithm 2

- Andersen's algorithm: generate set inclusion constraints for each statement
  - \( x = \&y \quad : \quad \{ y \} \subseteq \text{Ptr}(x) \)
  - \( x = y \quad : \quad \text{Ptr}(y) \subseteq \text{Ptr}(x) \)
  - \( x = \#y \quad : \quad \text{Ptr}(z) \subseteq \text{Ptr}(x), \text{for all } z \in \text{Ptr}(y) \)
  - \( \#x = y \quad : \quad \text{Ptr}(y) \subseteq \text{Ptr}(z), \text{for all } z \in \text{Ptr}(x) \)
- Subset relation similar to subtyping
- More precise than Steensgaard, less precise than dataflow
- Both algorithms create a spurious edge \( b \rightarrow d \), as a result of statements \( a = \&b, \#a = \#d \)
  - However, this sequence never happens during execution

Example

- Consider the following program:

```plaintext
if (cond) {
  a = \&b;
  b = \&d;
} else {
  a = \&c;
  \#a = \#d;
}
```

- Result (valid at all program points):

```
\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3
```

Summary

- Inter-procedural analysis:
  - Context sensitive
  - Context insensitive
- Intra-procedural analysis:
  - Flow-sensitive (dataflow analysis)
  - Flow-insensitive
- Flow, context-sensitive: more precise, expensive
- Flow, context-insensitive: less precise, efficient
