Dependence-Based Program Analysis

Richard Johnson
Keshav Pingali

Overview

- Problem Description
- Solution to the Problem
- Definitions and Terms
- Outline of Algorithm
- Examples
- Conclusion

Problem Description

- Constant Propagation
- Partial Redundancies

The classic constant propagation and the elimination of partial redundancies algorithms are based on data flow analysis. The values are computed iteratively by propagating information from inputs of statements to their outputs in the case $forward$ analysis and from outputs to inputs in the case of $backward$ analysis.

This approach has several disadvantages.

Problem Description

The disadvantages are:

- Information is propagated throughout the control flow graph.
- When the vector at some point in the program is updated, the entire control flow graph below that point (or above it in backward analysis) may be re-analysized.
- Many optimizations benefit from analysis performed in stages, but this is difficult to do in the standard approach.

Two solutions:

- Def-use chains
- Static Single Assignment (SSA)

Def-Use Chains and SSA Edge Description

- A def-use chain for a variable x is an edge pair (e_1, e_2) such that:
  - the source of e_1 defines x.
  - the destination of e_2 uses x and
  - there is a control flow path from e_1 to e_2 with no assignment to x.

- An SSA edge for variable x corresponds to an edge pair (e_1, e_2) such that:
  - there exists a definition of x that reaches e_1.
  - there exists a use of x reachable from e_2
  - there is no assignment to x on any control flow path from e_1 to e_2 and
  - e_1 dominates e_2

Problem Description cont.

Def-use chains provide a partial solution to these problems, but there are three problems with using def-use chains:

- Def-use chains cannot be used for backward dataflow problems because they do not incorporate sufficient information about the control structure of the program.
- The lack of information in def-use chains affects the precision of the analysis.
- The worst case size of the def-use chains is $O(E^2V)$ where E is the number of edges of the control flow graph and V is the number of variables.

Problem Description cont.

The size problem can be overcome by using a "factored" representation of def-use chains called $static\ single\ assignment$ (SSA) form, which has the worst-case asymptotic size $O(EV)$. SSA uses a $\phi-$function to combine def-use edges having the same destination to reduce the size issue. The problem with the SSA form is it cannot be used for backward analysis.

Solution to the Problem

The Dependence Flow Graph (DFG) is a representation of the program dependences which generalizes def-use chains and static single assignment (SSA) form. The DFG for a program can be constructed in $O(EV)$ time. The algorithm itself can be used to construct a program's control dependence graph in $O(E)$ time and its SSA representation in $O(EV)$ time, which are improvements over existing algorithms.

Unlike def-use chains, which go directly from definitions to uses, a DFG edge for a variable x can bypass a region of the control flow graph only if this region is a single-entry single-exit region that does not contain an assignment to x, since such a region has neither data nor control information that is of interest to the program analysis.

Definitions and Terms

- A node or edge x is said to dominate node or edge y in a directed graph if every path from $start$ to y includes x.
- A node or edge x is said to postdominate node or edge y in a directed graph if every path from y to end includes x.
- A node or edge x is said to be control dependent on node n if x postdominates all edges on some path from n to x, but x does not postdominate n. (Intuitively, n is a conditional branch that determines if control will pass through x.)

Definitions and Terms cont.

- A DFG edge for variable x corresponds to an edge pair (e_1, e_2) such that:
  1. there exists a definition of x that reaches e_1
  2. there exists a use of x reachable from e_2
  3. there is no assignment to x on any control flow path from e_1 to e_2
  4. e_1 dominates e_2
  5. e_2 postdominates e_1 and
  6. every cycle containing e_1 also contains e_2 and vise versa

Conditions 1 through 4 are the same as in the SSA representation. In the DFG, the merge operator plays the same role as $\phi-$functions do in SSA form. Conditions 4 through 6 formally specify that the region of the control flow graph between e_1 and e_2 must be a single-entry single-exit region.

A Comparison of Program Representations

control flow · · · · · · · · ➤ dependence for x ——➤ dependence for y - - - - ➤

start
x := 1
y := 2
if x=1
y := y+1
.. y ..
.. x ..

(a) CFG with def-use chains

start
x := 1
y := 2
if x=1
y := y+1
φ
.. y ..
.. x ..

(b) SSA form

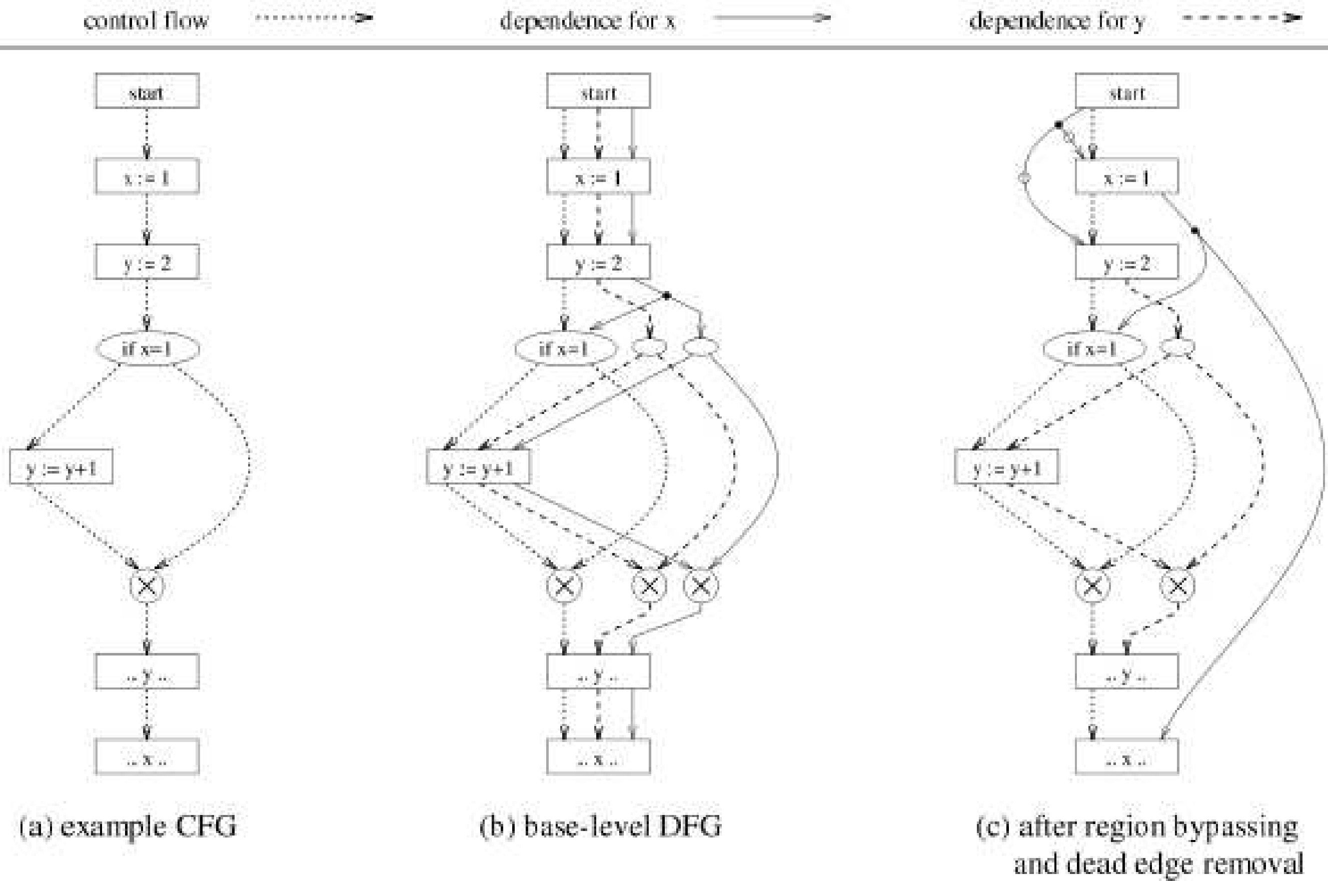
start
x := 1
y := 2
if x=1
y := y+1
.. y ..
.. x ..

(c) DFG form

Outline of Algorithm

1. Determine the variables defined within each single-entry single-exit region.
2. Create a base-level DFG with no region bypassing.
3. Perform region bypassing using the information found in Step 1.
4. Remove dead flow edges generated by bypassing.

An Illustration of DFG Construction



Def-Use Constant Propagation Example

```
if (p) then
{
        z := 1;
        x := z+2;
}
else
{
        z := 2;
        x := z+1;
}
y := x;
```

(a) all-paths
The first use of z can be replaced by 1, and the second use of z by 2. The right hand sides of the two definitions of x can now be simplified to the constant 3, and the final use of x can be replaced by 3.

DFG Constant Propagation Example

```
p := true;
if (p) then
{
        x := 1;
}
else
{
        x := 2;
}
y := x;
```

(b) possible-paths
By ignoring the definition on the unexecuted branch, the use of x in the last statement can be determined to have a value 1. Such $possible - path\ constants$ are common in code generated from inline expansion of procedures or macros, but algorithms that use def-use chains alone do not find these constants.

Conclusion

This approach achieved:

- Speeded up the program analysis and optimization that is based on the use of the dependence flow graph
- Derived a fast algorithm for constructing a DFG which can also be used to construct a factored control dependence graph.
- Avoids inefficiencies by:
  1. Propagating information for a variable x only where needed, bypassing single-entry single-exit regions of the graph that contain on definition of x
  2. Performing work proportional to the number of variable references at each assignment statement