Introduction to Dataflow Analysis

Kenneth Zadeck
NaturalBridge, Inc.
zadeck@naturalbridge.com

GCC & GNU Toolchain Developers' Summit 2006

Roadmap

- Introduction
- Live Variables – Working Example
- Solving Dataflow Equations Efficiently
- Common Dataflow Problems
- Incremental Analysis
- Other Dataflow Problems

This is a Tutorial

- Feel free to ask questions at any time.
- Nothing new is going to be presented.
  - 80% of this talk was old news in 1980.
  - 95% of this talk was old news in 1985.
- I only had a little bit to do with developing dataflow analysis.
- All of the material presented except the incremental dataflow information should be any good compiler text book.
- There is no paper in the proceedings for this talk.

Scope

- Most of the talk is concerned with bit vector dataflow analysis. In the literature these problems are called *rapid problems*.

Roadmap

- Introduction
- Live Variables – Working Example
- Solving Dataflow Equations Efficiently
- Common Dataflow Problems
- Incremental Analysis
- Other Dataflow Problems

Live Variables

At the end of each statement, s, in the function f, what is the set of variables, v, that have a use that may be reached without going through a definition of v.

Live Variables

At the end of each statement, *s*, in the function *f*, what is the set of variables, *v*, that have a use that may be reached without going through a definition of *v*.

Dataflow information can be represented at the start of a statement or the end of a statement.

Live Variables

At the end of each statement, *s*, in the function *f*, what is the set of variables, *v*, that have a use that may be reached without going through a definition of *v*.

Global dataflow analyzes whole functions.

Live Variables

At the end of each statement, *s*, in the function *f*, what is the set of variables, *v*, that have a use that may be reached without going through a definition of *v*.

The domain of the problem. Common values include the set of variables, the set of definition sites, or the set of use sites.

Live Variables

At the end of each statement, *s*, in the function *f*, what is the set of variables, *v*, that have a use that may be reached without going through a definition of *v*.

The *gen set.* The points in the program that cause items to be added to the set.

Live Variables

At the end of each statement, *s*, in the function *f*, what is the set of variables, *v*, that have a use that may be reached without going through a definition of *v*.

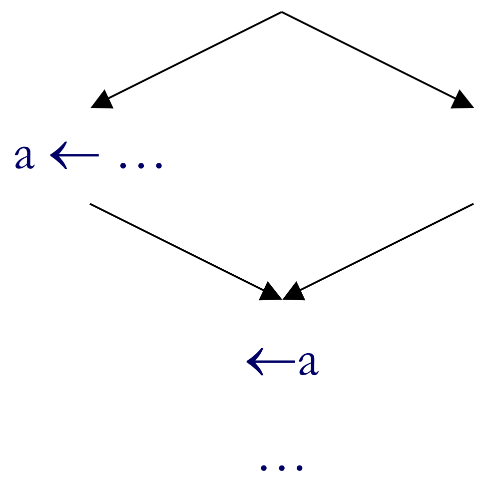
The *kill set*. The points in the program that cause items to be deleted from sets.

Direction

There are two live variables problems:

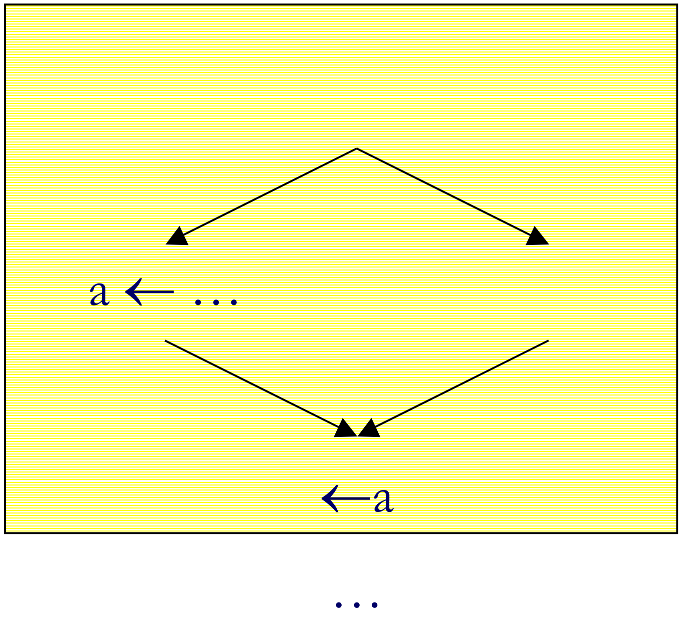
- *backwards* – propagation proceeds against the edges in the CFG.
  - gen is the set of uses.
  - kill is the set of defs and clobbers.
  - this is what **flow** does.
- *forwards* - propagation proceeds with the edges in the CFG.
  - gen is the set of definitions.
  - kill is the set of clobbers.
  - this is what global.c:make_accurate_live_analysis does.
  - in **df** we call this *uninitialized uses.*

Direction II



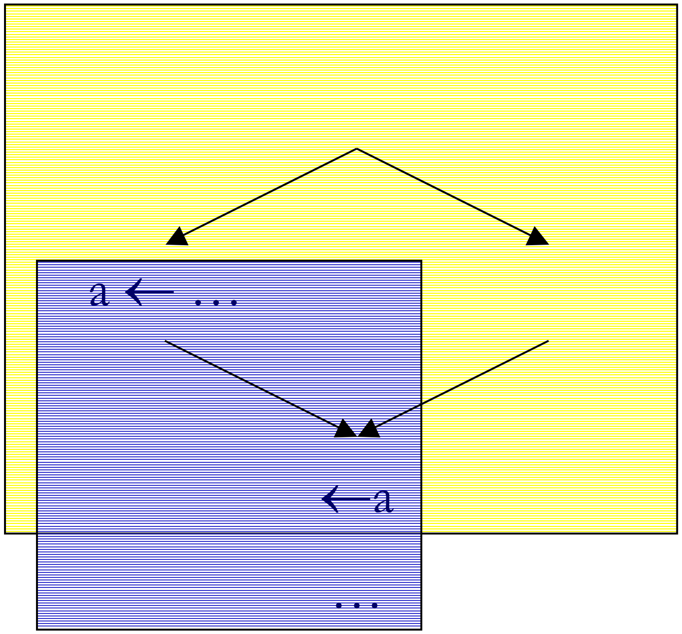
Direction II

- backward (flow)



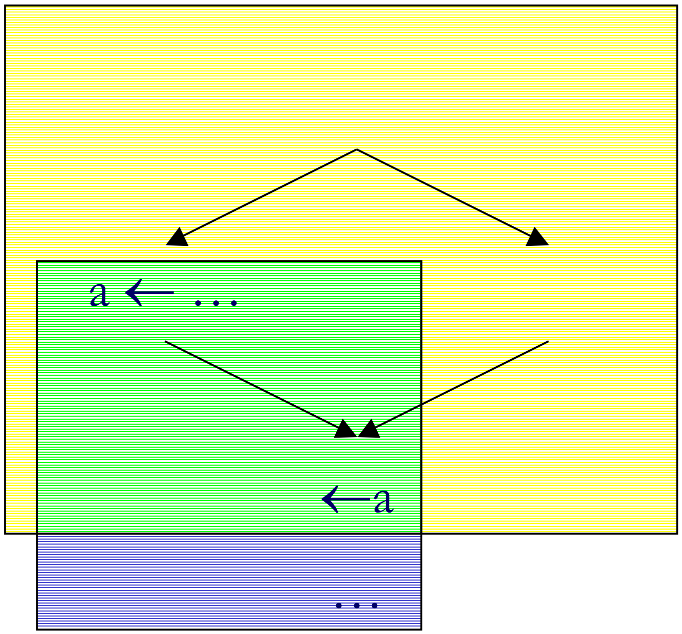
Direction II

- backward (flow)
- forward (uninitialized uses)



Direction II

- backward (flow)
- forward (uninitialized uses)
- forward & backward



Dataflow Equations

- Backwards:

$$out_h = \bigcup_{s = succ(h)} in_s$$

$$in_h = out_h - kill_h + gen_h$$

- Forwards:

$$in_h = \bigcup_{p = pred(h)} out_p$$

$$out_h = in_h - kill_h + gen_h$$

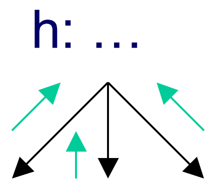
for every h in the program.

Dataflow Equations

- Backwards:

$$out_h = \bigcup_{s = succ(h)} in_s$$

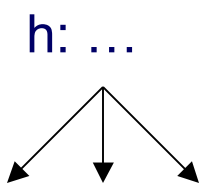
$$in_h = out_h - kill_h + gen_h$$



Dataflow Equations

- Backwards:

$$out_h = \bigcup_{s = succ(h)} in_s$$



$$in_h = out_h - kill_h + gen_h$$

Apply the effects of h.

Correctness and Quality

- A solution is *correct* if it satisfies the system of equations.
- For cyclic control flow graphs there are many correct solutions.
- Want to find the *minimal* solution.
  - For live variables this is the correct solution with the with the *fewest* 1 bits.

Roadmap

- Introduction
- Live Variables – Working Example
- Solving Dataflow Equations Efficiently
- Common Dataflow Problems
- Incremental Analysis
- Other Dataflow Problems

Plan I – Worklist Iteration

worklist $\leftarrow$ all statements
while (worklist $\neq$ empty)
{
    h $\leftarrow$ take from worklist
    old $\leftarrow$ in_h
    $out_h \leftarrow$ empty
    for each s in $succ_h$
        $out_h \leftarrow out_h \,||\, in_s$
    $in_h \leftarrow (out_h \,\&\&\, \sim kill_h) \,||\, gen_h$
    if ($in_h \neq$ old)
        for each p in $pred_h$
            add p to worklist
}

Plan 1.1 – Flow.c

```
worklist ← all statements
while (worklist ≠ empty)
{
    h ← take from worklist
    old ← in_h
    out_h ← empty
    for each s in succ_h
        out_h ← out_h || in_s
    in_h ← (out_h && ~kill_h) || gen_h
    if (in_h ≠ old)
        for each p in pred_h
            add p to worklist
}
```

Evaluate all of the statements in a basic block at one time and in reverse order.

Plan 2 – Transfer Functions

- It is not necessary to apply gen and kill separately for each statement.
- Gen and kill can be computed for an entire basic block.
- For live variables:

  $gen_{s1+s2} \leftarrow (gen_{s2} - kill_{s1}) \| gen_{s1}$

  $kill_{s1+s2} \leftarrow (kill_{s2} - gen_{s2}) \| kill_{s1}$

- The worklist algorithm then uses blocks rather than statements.

Plan 3 – Elimination Algorithms

- Transfer functions can be built for other program structures than sequences of statements:
- Loops and if-then-elses can be similarly handled.
- for if-then-elses:

$gen_{if} \leftarrow gen_{true} \,||\, gen_{false}$

$kill_{if} \leftarrow kill_{true} \;\&\&\; kill_{false}$

Plan 3 – Elimination Algorithms

- It is possible to parse any *reducible* program into sequences, simple loops and if-then-elses.
- Process the transfer functions bottom up, then top down and you have the solution.
- Multiple entry loops must be processed by iteration.
- For many years this was the method of choice because it was much faster than the worklist.
- Allen Cocke and Schwartz were the first to propose this.
- Graham and Wegman was the method of choice.

Plan 4 – Better Worklists

- Hecht demonstrated that better management of the worklist could yield an algorithm that was just as fast as elimination algorithms.
  - For forward problems, multiple passes in reverse postorder are made.
  - For reverse problems, multiple passes in postorder are made.
- Works for all functions, even irreducible ones.
- Easy to implement.

Plan 4.1 – Even Better Worklists

- Atkinson and Griswold demonstrated that doing a little depth first search in the middle of Hecht’s algorithm generally speeded things up.
- This is what is used in df.

Roadmap

- Introduction
- Live Variables – Working Example
- Solving Dataflow Equations Efficiently
- Common Dataflow Problems
- Incremental Analysis
- Other Dataflow Problems

Common Dataflow Problems

- There are 5 common fast dataflow problems:
  - Live Variables.
  - Uninitialized Variables
  - Reaching Uses
  - Reaching Definitions
  - Code Placement

Common Dataflow Problems

- These problems are common because they are easy to understand and formulate for text books and student compilers.
- Real compilers have more hair:
  - subregs
  - aliasing
  - thread synchronization
- You generally can use these problems as stating points for the solution to your exact problem.

Live Variables

- Backward.
- Each slot is for one variable.
- Kill = the set of variables clobbered
  + the set of variables defined at *s*.
- Gen = the set of variables used at *s*.
- Confluence = or (set union).

Uninitialized Variables

- Forward.
- Each slot is for one variable.
- Kill = the set of variables clobbered at *s*.
- Gen = the set of variables defined at *s*.
- Confluence = or (set union).

Reaching Uses

- Backward.
- One use for each slot in the bit vectors.
- Kill = if *v* is clobbered or defined at *s*
  add all uses of *v.*
- Gen = the set of uses at the statement.
- Confluence = or (set union).

Reaching Defs

- Forward.
- One def for each slot in the bit vectors.
- Kill = if *v* is clobbered or defined at *s*
  add all defs of *v*.
- Gen = the set of defs at *s*.
- Confluence = or (set union).

Code Placement

- Originally proposed as bidirectional problem by Morel and Renvoise.
- Large number of reformulations in the literature.
- People tend to use Chow's second formulation. This is a backward problem followed by a simple forward cleanup.
- Some variant of Chow's second formulation is used in GCC.

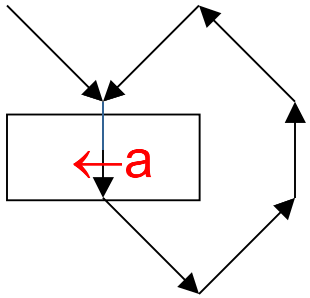
Roadmap

- Introduction
- Live Variables – Working Example
- Solving Dataflow Equations Efficiently
- Common Dataflow Problems
- Incremental Analysis
- Other Dataflow Problems

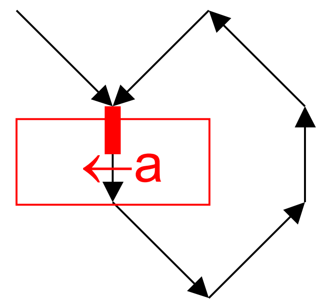
Incremental Dataflow Analysis

- Incremental dataflow analysis is *hard:*
  - It is easy to get a *correct* solution.
  - It is hard to get a *minimal* solution.

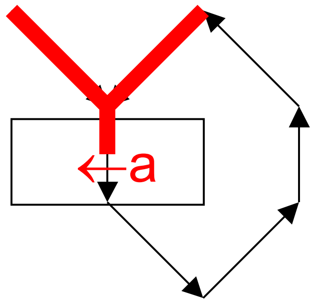
Incremental Dataflow Analysis



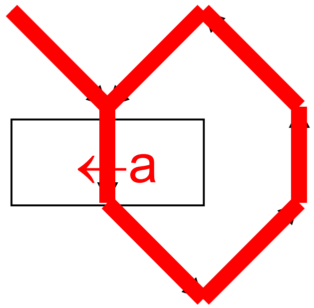
Incremental Dataflow Analysis



Incremental Dataflow Analysis

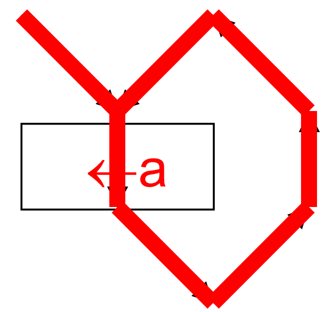


Incremental Dataflow Analysis

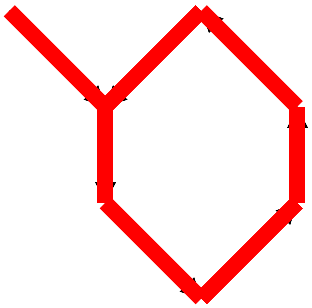


Incremental Dataflow Analysis

- For live variables, bits get *stuck on* in loops after the deletion of a bit in a *gen* set.

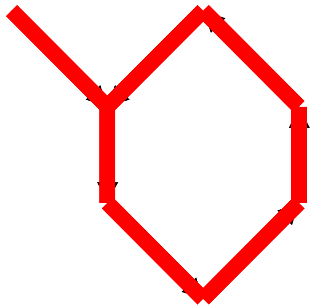


Incremental Dataflow Analysis



- For live variables, bits get *stuck on* in loops after the deletion of a bit in a *gen* set.
- That *in* is defined in terms of *out* and *out* is defined *in* terms of *in* of the *preds* means that it is hard to break cycles.
- This is a *correct* solution, just not *minimal*.

Incremental Dataflow Analysis



- For live variables, bits get *stuck on* in loops after the deletion of a bit in a *gen* set.
- That *in* is defined in terms of *out* and *out* is defined *in* terms of *in* of the *preds* means that it is hard to break cycles.
- This is a correct solution, just not minimal.
- *This is what flow does and we do not want to do this.*

Incremental Dataflow Algorithms

- You must delete all of the bits that were associated with the *gen* that was removed without deleting bits associated with *gen*s that are preserved.

Incremental Dataflow Algorithms

- You must delete all of the bits that were associated with the *gen* that was removed without deleting bits associated with *gen*s that are preserved.
- All algorithms fall into two categories:
  - Ryder: find the smallest region that encompasses all of the possible bits from the deleted gen.
  - Zadeck: go after the bits one by one.

Incremental Dataflow Algorithms

- You must delete all of the bits that were associated with the *gen* that was removed without deleting bits associated with *gen*s that are preserved.
- All algorithms fall into two categories:
  - Ryder: find the smallest region that encompasses all of the possible bits from the deleted gen.
  - Zadeck: go after the bits one by one.
- Neither approach yielded a practical algorithm for compilers.

Incremental Dataflow Algorithms

- You must delete all of the bits that were associated with the *gen* that was removed without deleting bits associated with *gen*s that are preserved.
- All algorithms fall into two categories:
  - Ryder: find the smallest region that encompasses all of the possible bits from the deleted gen.
  - Zadeck: go after the bits one by one.
- Neither approach yielded a practical algorithm for compilers.
- I have let it go.

Roadmap

- Introduction
- Live Variables – Working Example
- Solving Dataflow Equations Efficiently
- Common Dataflow Problems
- Incremental Analysis
- Other Dataflow Problems

Other Dataflow Problems

There are a lot of dataflow problems that do not fit into the rapid framework:

- You cannot build efficient transfer functions.
- The slots in the vectors are not independent.
- The values in the slots are not single bits.

- Constant Propagation.
- Alias Analysis.