Dataflow Analysis

February 28, 2001

CS 132: Compiler Design

Control Flow

- Definition: The *control flow* of a program is the possible sequences of instructions or blocks that a program may execute
  - Obviously undecidable to compute perfectly
  - Generally we ignore all expressions and just look at the graph structure of the program
    - Nodes are instructions or basic blocks
    - Each edge represents a syntactically possible flow of control. (E.g., every CJUMP has two successors)
- Control-flow analysis involves finding this graph and looking for various properties (e.g., identifying loops)

Data Flow

- The purpose of *dataflow analysis* is to analyze how code uses its values.
  - Connecting computations to where they're used
  - What values are available at a given point?
  - Where did they come from?
  - What variables might be referred to later?
- NB: Once you have indirect jumps (such as higher-order functions) control flow becomes dependent upon data flow.

Reaching Definitions

- Question:
  - For each use of a variable, which assignments in the program could have set the value being used?
- Setup:
  - Give each assignment/definition in the program a unique label.
  - Answers are then sets of these labels.

Reaching Definitions

- For each temporary t, define $defs(t)$ to be the set of all assignments/definitions for the temporary t.
  - Ambiguous vs. unambiguous definitions
- For each instruction s, define $gen(s)$ to be the set of definitions generated by this instruction.
  - Singleton set if s is a definition
  - Empty set otherwise
- For each instruction s, define $kill(s)$ to be the set of all definitions not in force after this statement.
  - $defs(t) \setminus \{d\}$ if is the definition d to the temporary t.
  - Empty set otherwise

Reaching Definitions Example

```
                             GEN    KILL
1.       a ← 5               1      6
2.       c ← 1               2      4,7
3.   L1: if c>a goto L2
4.       c ← c+c             4      2,7
5.       goto L1
6.   L2: a ← c-a             6      1
7.       c ← 0               7      2,4
```

Reaching Definitions

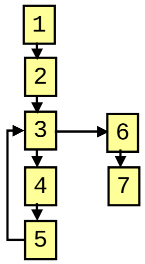
- For each instruction s, define $in(s)$ to be the set of definitions that might have generated the values of variables before s executes.
- For each instruction s, define $out(s)$ to be the set of definitions that might have generated the values of variables after s executes.

$$in(s) = \bigcup_{p \in pred(s)} out(p)$$
$$out(s) = gen(s) \cup (in(s) \setminus kill(s))$$

Reaching Definitions Example

```
                         GEN KILL
1.       a ← 5           1    6
2.       c ← 1           2    4,7
3.   L1: if c>a goto L2
4.       c ← c+c         4    2,7
5.       goto L1
6.   L2: a ← c-a         6    1
7.       c ← 0           7    2,4
```

$$in(1) = \varnothing$$
$$in(s) = \bigcup_{p \in pred(s)} out(p)$$
$$out(s) = gen(s) \cup (in(s) \setminus kill(s))$$



Reaching Definitions Example

- We get the following equations for this example:

- Solution?

Solution by Iteration

- Initialize all the sets to be empty
- Use the equations and the current values of the sets to compute new values of the sets
  - Repeat until none of the sets change
  - Optimization: Use the new values as soon as they're available.

- Why does this work?

Application

- Constant propagation optimization
  - Suppose d is a definition of the form a ← N for some constant N.
  - Suppose we have statement that uses a. If d is the only definition of a reaching this statement, then the use of a can be replaced by N.

  - Note: may result in dead code

Application

- Copy propagation elimination
  - Suppose d is a definition of the form a ← b for some variable b.
  - Suppose we have statement that uses a.
  - If d is the only definition of a reaching this statement and there is no definition of b on any path from a to this use, then the use of a can be replaced by b.

```
a ← y+z
u ← y
c ← u+z
```

Observations

- We want analysis to be *sound* but cannot expect it to be *complete*
  - That is, analysis must err on the side of caution
  - Depending on the optimization, we must overestimate or underestimate sets
    - Want to overestimate "possibilities"
    - Want to underestimate "guarantees"

Liveness

- Definition
  - A variable is said to be *live* at a program point if there is path to a use of this variable that does not include an assignment to the variable
    - That is, the control flow graph suggests we may use the current value of this variable later.
- Question:
  - For every program point, determine the variables live variables at that point
  - Answer will be a set of variables for each point.

Liveness

- For each instruction s, define $def(s)$ to be the variables potentially modified by this instruction.
- For each instruction s, define $use(s)$ to be the set of variables accessed by this instruction.

$$livein(s) = use(s) \cup (liveout(s) \setminus def(s))$$

$$liveout(s) = \bigcup_{p \in succ(s)} livein(p)$$

$$liveout(terminal\ node) = \emptyset$$

Liveness

- For each instruction s, define $in(s)$ to be the set of variables live before s executes.
- For each instruction s, define $out(s)$ to be the set of definitions live after s executes.

$$in(s) = use(s) \cup (out(s) \setminus def(s))$$

$$out(s) = \bigcup_{p \in succ(s)} in(p)$$

$$out(last\ node) = \emptyset$$

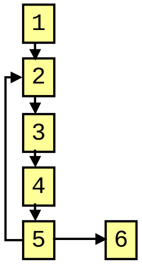
Forward vs. Backward Analysis

- A dataflow analysis is said to be a *forward analysis* if the *out* set depends only on the *in* set of this node.
- A dataflow analysis is a *backward analysis* if the *in* set depends only on the *out* set for this node.
- A dataflow analysis is said to be *bidirectional* if neither of these is true.

Liveness Example

```
                          DEF USE
1.      a ← 0             a
2.  L1: b ← a+1           b    a
3.      c ← c+b           c    b,c
4.      a ← b*2           a    b
5.      if a<5 goto L1         a
6.      return c               c
```

$$in(s) = use(s) \cup (out(s) \setminus def(s))$$
$$out(s) = \bigcup_{p\in succ(s)} in(p)$$



Liveness Solutions

```
1.      a ← 0
2.  L1: b ← a+1
3.      c ← c+b
4.      a ← b*2
5.      if a<5 goto L1
6.      return c
```

in	**out**
c	ac
ac	bc
bc	bc
bc	ac
ac	ac
c	

in	**out**
cd	acd
acd	bcd
bcd	bcd
bcd	acd
acd	acd
c	

$$in(s) = use(s) \cup (out(s) \setminus def(s))$$
$$out(s) = \bigcup_{p\in succ(s)} in(p)$$

Application

- Dead code elimination
  - Assume there is a definition of the form a ← e where e has no side-effects.
    - Including overflow exceptions, writes to memory, function calls that might have effects, etc.
  - If a is not in the live-out set of this instruction, then the definition can be deleted.

Application

- Register allocation
  - Any two distinct variables that are simultaneously live at some point in the program cannot be stored in the same register (or the same stack location)
    - Unless they are guaranteed to have the same value
  - We will come back to this after the break