CSc 553 — Principles of Compilation

27 : Optimization II

Christian Collberg
Department of Computer Science
University of Arizona
collberg@gmail.com

March 31, 2011

1 Introduction

2 Global Optimization I

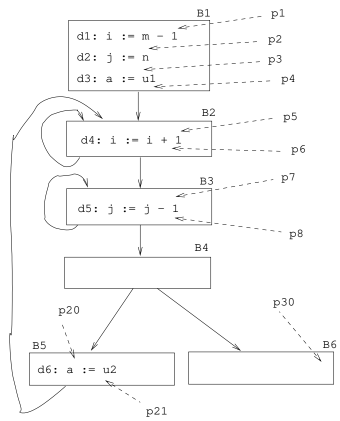
- Central to all the global optimizations are that they make use of **global data-flow analysis**. We are going to start out by discussing the algorithms for performing global optimizations, assuming that the information computed during data-flow analysis is available. Later on we will describe how this information is actually computed.
- Global data-flow analysis answers questions like: "will a value v computed at a point p in a procedure be available at some other point q?"
- In other words, global data-flow analysis tracks values and computations as they cross basic block boundaries.

3 Global Optimization II

- In general we can't know how control will flow in a flow-graph, so our data-flow analysis has to be conservative: it has to assume that any possible control flow may be an actual control flow. This means that
  1. any optimization that we perform will not change the semantics of the program, and
  2. sometimes we **won't** be able to perform an optimization that would be perfectly legal, just because our data-flow information is imprecise (too conservative).

4 Global Optimization III

- In order to perform global optimizations we need access to global data flow information, i.e. we need to track values as they cross basic block boundaries.



5

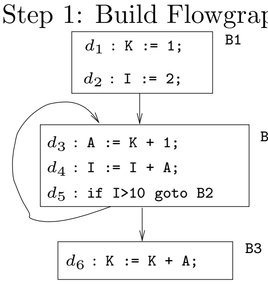
Loop Invariants

6 Loop Invariants I

- Let C be a computation in a loop body. C is **invariant** if it computes the same value during all iterations. C can sometimes be moved out of the loop.

```
K := 1; I := 2;
REPEAT
  A := K + 1; I := I + A;
UNTIL I <= 10;
K := K + A;
```

Step 1: Build Flowgraph



7 Loop Invariants II

- To detect (in the previous slide) that `A := K + 1` is invariant, we must first detect that `K` is not changed within the loop.
- We really want to know is where the value that `K` has in the loop could have been computed. If `K`'s value could only have been computed outside the loop (as is, in fact, the case) then we can conclude that `K` is not changed within the loop, and hence `A := K + 1` is invariant.
- Information on where the value of a particular variable (at a particular point in the program) could have been computed is often stored in **use-definition-chains**, or **ud-chains**.

8 Loop Invariants III

- If, at a point `p` in the program, we use a variable `i`, and the point `q` is on `i`'s ud-chain at point `p`, then the value of `i` at `p` could have been computed at `q`.
- Note that we have no way of knowing exactly where `i`'s value has been computed (this depends on the actual control-flow at run-time and is in general non-computable [*equiv* the halting problem]), so our ud-chains will be overly pessimistic: there may be points on `i`'s ud-chain that are never reached at run-time (and hence `i` could never receive a value there), but we can't be sure of this and therefore we must include those points as well.

9 Reaching Definitions I

- To detect loop invariant computations we must first compute **reaching definitions**, or **use-definition-chains**.
- `p: ud[i]`, the `ud`-chain for a variable `i` at a point `p`, lists the points in the program where `i`'s value could have been computed.

Example:

```
d1:  i := m - 1;
d2:  j := n;
     REPEAT
d3:     i := i + 1;  ud[i]=⟨d1, d3, d6⟩
d4:     j := j - 1;  ud[j]=⟨d2, d4⟩
        IF ··· THEN d5:  a := ···;
        ELSE d6:  i := ···;
        END;
     UNTIL ···
```

10 Reaching Definitions II

- Ud-chains are built from the solution to a data-flow problem known as **reaching definitions**. For example, in the previous slide
  1. the definition of `i` at point d_1 **reaches** point d_3, because the value of `i` on the right hand side of `i := i + 1` could have been computed at d_1.
  2. the definition of `i` at point d_3 reaches d_3 itself, since if the first branch of the IF-statement is taken d_6 won't be executed, and `i`'s value computed at d_3 will still be valid the next time d_3 is reached.
  3. if the second branch of the IF-statement is taken, however, `i`'s value computed at d_6 will reach d_3.

  Hence, the definitions of `i` at d_1, d_3, d_6 all reach d_3, and are on `i`'s ud-chain at d_3.

11 Data-Flow Problems I

- In order to build ud-chains, we have to solve the data-flow problem **reaching definitions**.
- Data-flow problems are problems on **sets**. These sets can be sets of points in the control flow graph, sets of expressions, sets of variables, etc, depending on the nature of the problem.

- In the reaching definitions problem we work on sets of **definitions**, or places in the control flow graph where a particular variable could have received its value.
- In general, data-flow problems come in two parts: a local part which solves the problem within each basic block, and a global part which combines the local solutions into a global solution.

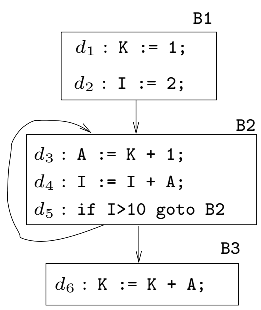
12 Data-Flow Problems II

- The local part of the data-flow problem consists of (for each basic block `B`), computing two sets, often called `gen[B]` and `kill[B]`.
- For the reaching definitions problem `gen[B]` is the set of definitions that occur within `B` and that reach the end of `B`. I.e., if there is more than one assignment to a in `B`, then only the last one reaches the end of `B`.
- Similarly, `kill[B]` is the set of definitions (anywhere in the routine) that are killed by definitions in `B`. I.e., if some definition d reached the beginning of `B` and d is in `kill[B]`, we could be sure that that definition would not reach the end of `B`.

13 Loop Invariants IV

- For each basic block we compute `gen` and `kill`.
- `gen[B]` is the set of definitions (assignments to variables) that occur within `B`. `kill[B]` is the set of definitions outside `B`, killed by definitions within `B`.

Step 2: Compute `gen & kill`:



`gen[B1]` = $\{d_1, d_2\}$
`kill[B1]` = $\{d_4, d_6\}$

`gen[B2]` = $\{d_3, d_4\}$
`kill[B2]` = $\{d_2\}$

`gen[B3]` = $\{d_6\}$
`kill[B3]` = $\{d_1\}$

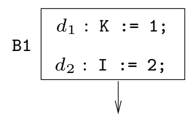
14 Data-Flow Problems III

- The global part of a data-flow problem computes sets called `in[B]` and `out[B]`.
- For the reaching definitions problem, `in[B]` is the set of definitions that reach the beginning of `B`, i.e. the set of points in the control flow graph that computes values that may still be valid when we (at run time) reach the beginning of `B`.
- Similarly, `out[B]` is the set of definitions that may reach the end of `B`, i.e. the set of points in the flow graph that computes values that may still be valid when control reaches the end of `B`.
- Consider the example in the previous slide:
  1. Since no definitions are going into `B1`, `out[B1]` is simply the set of defs generated within `B1`, i.e. `gen[B1]`.

15 Loop Invariants V (a)

- Next we must compute `in` and `out`.
- `in[B]` is the set of definitions that reach the beginning of `B`.
- `out[B]` the set that reach `B`'s end.

Step 2: Compute `in` & `out` for `B1`:



`gen[B1]` = $\{d_1, d_2\}$
`kill[B1]` = $\{d_4, d_6\}$
`in[B1]` = $\{\}$

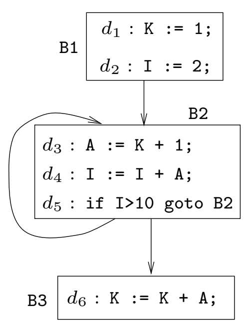
`out[B1]=gen[B1]`$\cup$`(in[B1]-kill[B1])`
=$\{d_1, d_2\} \cup (\{\} - \{d_4, d_6\}$
=$\{d_1, d_2\}$

16

`Out[B2]`= definitions active at the end of `B2`, i.e. generated within the `B2`, plus those incoming definitions that weren't overridden by definitions within `B2`.

`in[B2]`= definitions that compute values that could be active at the entrance of `B2`. These could have been computed in any of `B2`'s predecessors.

Step 2: Compute `in` & `out` for `B2`:



`out[B1]`=$\{d_1, d_2\}$

`gen[B2]` = $\{d_3, d_4\}$
`kill[B2]` = $\{d_2\}$

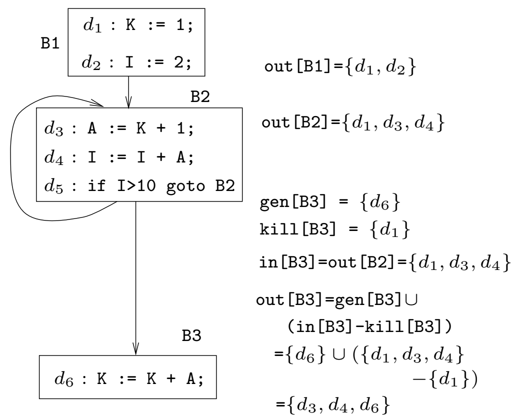
`in[B2]=out[B1]`$\cup$ `out[B2]`
=$\{d_1, d_2\} \cup \{d_1, d_3, d_4\}$
=$\{d_1, d_2, d_3, d_4\}$

`out[B2]=gen[B2]`$\cup$
`(in[B2]-kill[B2])`
=$\{d_3, d_4\}\cup$
$(\{d_1, d_2, d_3, d_4\} - \{d_2\})$
=$\{d_1, d_3, d_4\}$

17

- $d_1 \in$ `in[B3]` since `K` is not assigned a value on any path from `B1` to `B3`.
- $d_1 \notin$ `out[B3]` since `K` is assigned a new value in `B3`.

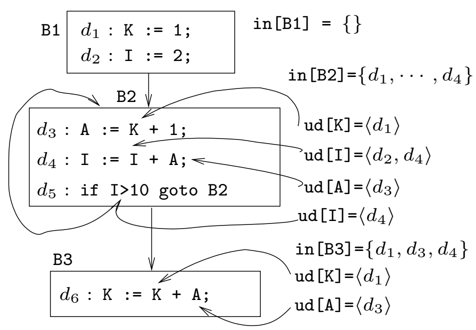
Step 2: Compute `in` & `out` for `B3`:



18 Loop Invariants VI

- Reaching definitions information is often stored as `ud`-chains.
- $d_k \in$ `ud[v]` at point `p`, if `v`'s value at `p` could have been computed in d_k.
- If `v` is used at point `p` in block `B`, then `du[v]` at `p` is the list of definitions of `v` in `in[B]`. (Well, almost.)

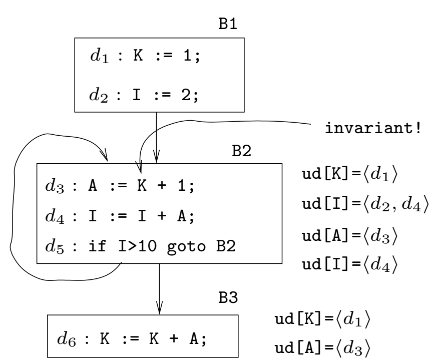
Step 3: Compute `du`-chains:



19 Loop Invariants VII

- With `du`-chains available, we can now mark invariant computations in loops.
- d_k is invariant if either all arguments are constants, or have their reaching definitions outside the loop.

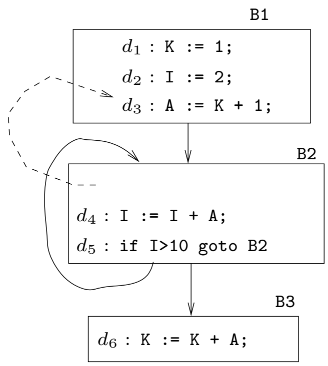
Step 4: Compute Invariant Expr:



20 Loop Invariants VIII

- Finally we can move invariant computations out of loops, subject to certain conditions (see the Dragon book, p. 641).

Step 5: Perform Code-Motion:



21 Loop Invariants IX

Algorithm

1. Build a flowgraph for the procedure.
2. Compute ud-chains:
   (a) Compute reaching definitions:
      i. For each block `B`, compute `gen[B]` and `kill[B]`.
      ii. For each block `B`, compute `in[B]` and `out[B]`.
   (b) Use `in[B]` to build ud-chains.
3. Use ud-chains to detect invariant computations.
4. When possible, perform code motion, by moving invariant computations out of loops.

22

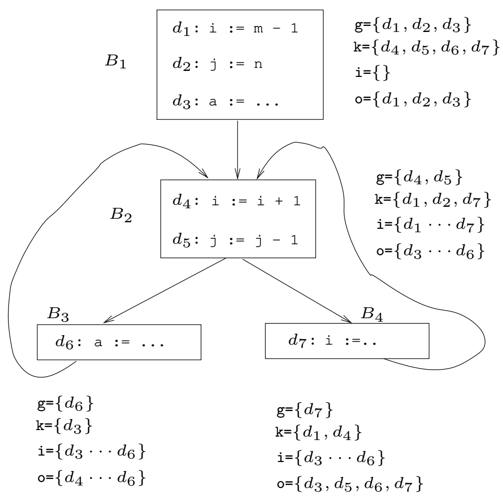
Reaching Definitions II

23 Reaching Def. Example I

- Let's take another look at the computations of ud-chains, through reaching definitions.

```
d1:  i := m - 1;
d2:  j := n;
d3:  m := ···;
     REPEAT
d4:    i := i + 1;
d5:    j := j - 1;
       IF ··· THEN
d6:      a := ···;
       ELSE
d7:      i := ···;
       END;
     UNTIL ···
```

24 Reaching Def. Example II



25 Reaching Def. Example III (a)

Block **B1**

- out[B1]={d_1, d_2, d_3}, since d_1, d_2, d_3 reach the end of the block (once i has been defined, there are no further assignments to it in the block).

Block **B2**

- in[B2]={$d_1, \cdots, d_7$}, since we can reach the entrance to B2 either through B2 (when definitions d_1, d_2, d_3 are active), through B3 (when definitions d_4, d_5, d_6 are active), or through B4 (when d_3, d_5, d_6, d_7 are active). The union of these sets is $\{d_1, \cdots, d_7\}$.
- out[B2]={$d_3, \cdots, d_6$}, since B2 definitely kills $\{d_1, d_2, d_7\}$in B1.

26 Reaching Def. Example III (b)

Block **B3**

- in[B3]=out[B2]=$\{d_3, \cdots, d_6\}$, since if some value is available at the end of B2, then certainly it is available at the beginning of B3.
- out[B3]=$\{d_4, \cdots, d_6\}$ since all the incoming definitions ($\{d_3, \cdots, d_6\}$) plus those definitions generated within B3 (d_4) are available at B3's exit, except those that were killed by B3 (d_4).

27 Data-Flow Equations I

$$\begin{aligned}
\texttt{out}[B] &= \texttt{gen}[B] \cup (\texttt{in}[B] - \texttt{kill}[B]) \\
\texttt{in}[B] &= \bigcup_{\substack{\texttt{predecessors} \\ P \texttt{ of } B}} \texttt{out}[P]
\end{aligned}$$

Local Equations:

- $d \in \texttt{gen}[\texttt{B}] \Rightarrow d$ reaches the end of B.
- $d \in \texttt{kill}[\texttt{B}] \Rightarrow d$ **does not** reach the end of B.

Global Equations:

- $d \in \texttt{in}[\texttt{B}] \Rightarrow d$ reaches the beginning of B.
- $d \in \texttt{out}[\texttt{B}] \Rightarrow d$ reaches the end of B.

28

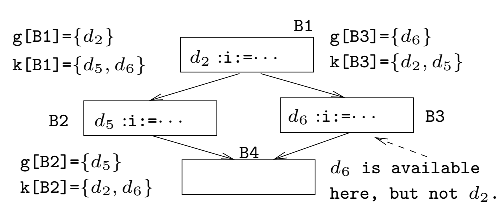
Global Data-Flow (Equations):

$$\texttt{out}[B] = \texttt{gen}[B] \cup (\texttt{in}[B] - \texttt{kill}[B])$$

Global Data-Flow (English):

out[B] (the definitions which are available at the end of B) contains the definitions generated within B (gen[B]), and the definitions that are available at the beginning of B (in[B]), except those that where superseded by definitions in B (kill[B]).

Example:



29

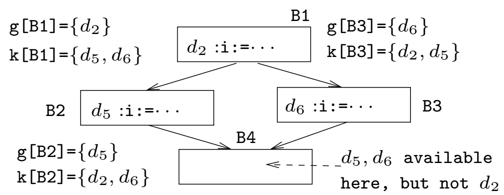
Global Data-Flow (Equations):

$$\texttt{in}[B] = \bigcup_{\substack{\texttt{predecessors} \\ P \texttt{ of } B}} \texttt{out}[P]$$

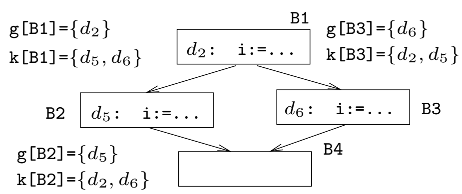
Global Data-Flow (English):

`in[B]` (the definitions which are available at the beginning of `B`) contains all definitions available at the end of any basic block from which control can flow into `B`.

Example:



30



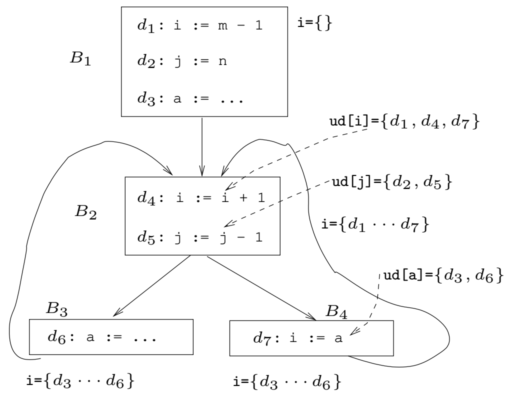
$$\begin{aligned}
\texttt{in}[B2] &= \{d_2\} \\
\texttt{in}[B3] &= \{d_2\} \\
\texttt{out}[B2] &= \texttt{gen}[B2] \cup (\texttt{in}[B2] - \texttt{kill}[B2]) \\
&= \{d_5\} \cup (\{d_2\} - \{d_2, d_6\}) \\
&= \{d_5\} \\
\texttt{out}[B3] &= \texttt{gen}[B3] \cup (\texttt{in}[B3] - \texttt{kill}[B3]) \\
&= \{d_6\} \cup (\{d_2\} - \{d_2, d_5\}) \\
&= \{d_6\} \\
\texttt{in}[B4] &= \texttt{out}[B2] \cup \texttt{out}[B3] \\
&= \{d_5, d_6\}
\end{aligned}$$

31 Computing UD-Chains

- Assume that variable a is used at point p in block B. Assume that there is no def. of a between the beginning of B and p.

- Then the ud-chain of a at p is the definitions of a in in$[B]$.

Example



32 Global Common Sub-Expression Elimination

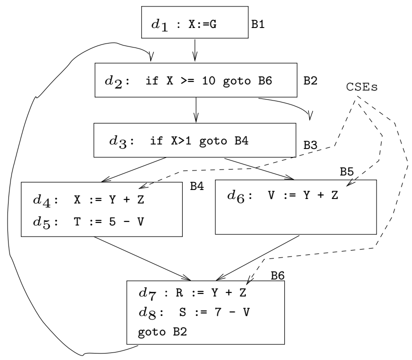
33 Global CSE I

- Reaching definitions are only one of many possible global data flow problems. Next we will see how we can perform **global common sub-expression elimination**, based on information computed by a global data-flow analysis problem called **Available Expressions**.
- In the next slide we see that
  1. when control reaches B6, the expression Y+Z will have been computed, regardless of which IF-statement branch has been taken,
  2. Y+Z is used within B6 and is therefore a common subexpression.

  Can we detect this so that we need not recompute Y+Z in B6 but rather can reuse the value computed in B4 & B5 in B6.

34 Global CSE II

```
X := G;
WHILE X < 10 DO
  IF X > 1 THEN X := Y + Z; T := 5 - V;
  ELSE V := Y + Z;
  END; R := Y + Z; S := 7 - V;
END;
```

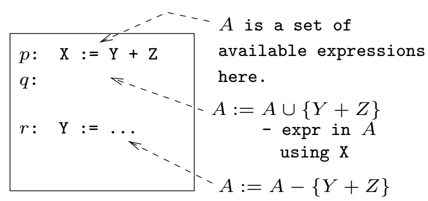


35 Available Expressions I (a)

- An expression E is available at some point p if (regardless of the actual execution path from the initial node to p) E's value will have been computed when p is reached.

Local Available Expr:

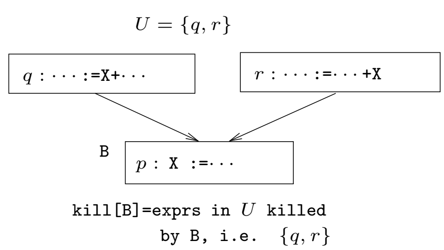
`gen[B]` is the set of expressions generated within `B` (and not killed by `B` itself).



36 Available Expressions I (b)

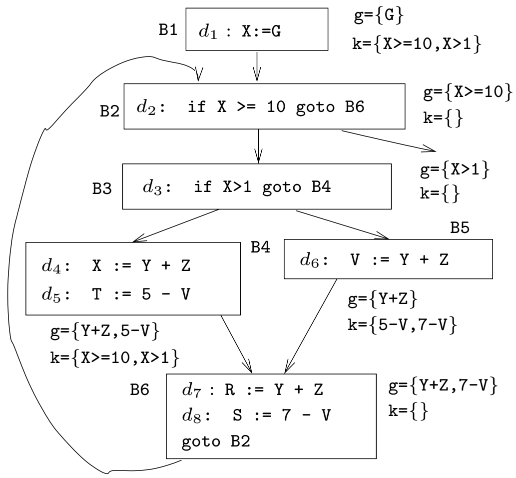
Local Available Expr:

`kill[B]` is the set of expressions killed within `B`, i.e. all expressions $Y + Z$ such that Y or Z is assigned to in `B`.



U is the universal set of expressions in the procedure.

37 Available Expressions II



38 Available Expressions III

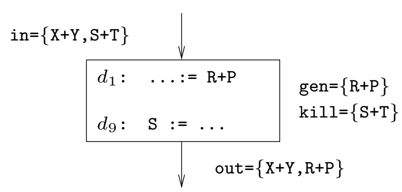
Global Data-Flow (Equations):

$$\texttt{out}[B] \quad = \quad \texttt{gen}[B] \cup (\texttt{in}[B] - \texttt{kill}[B])$$

Global Data-Flow (English):

`out[B]` (the set of expressions available at the end of `B`) contains the expressions generated within `B` (`gen[B]`), and the expressions that are available at the beginning of `B` (`in[B]`), except those that where killed by definitions in `B` (`kill[B]`).

Example:



39 Available Expressions IV

Global Data-Flow (Equations):

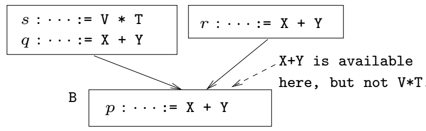
$$\texttt{in}[B_1] \quad = \quad \{\}$$
$$\texttt{in}[B] \quad = \quad \bigcap_{\texttt{preds}\,P \texttt{ of } B} \texttt{out}[P]$$

Global Data-Flow (English):

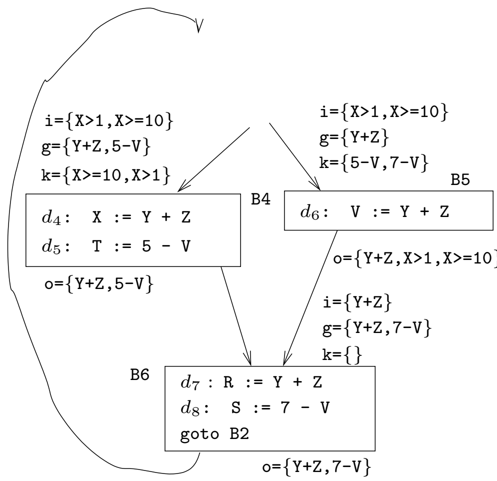
$\texttt{in[B}_1\texttt{]}$ is always empty.

in[B] expressions available at the entrance to B, i.e. any expression available at the exit of all of B's predecessors.

Example:



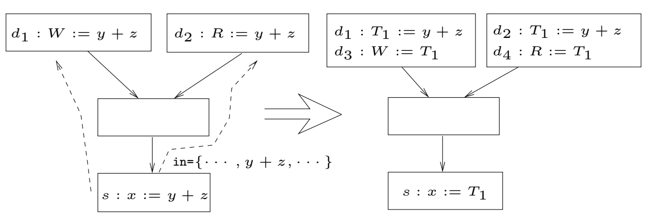
40 Available Expressions V



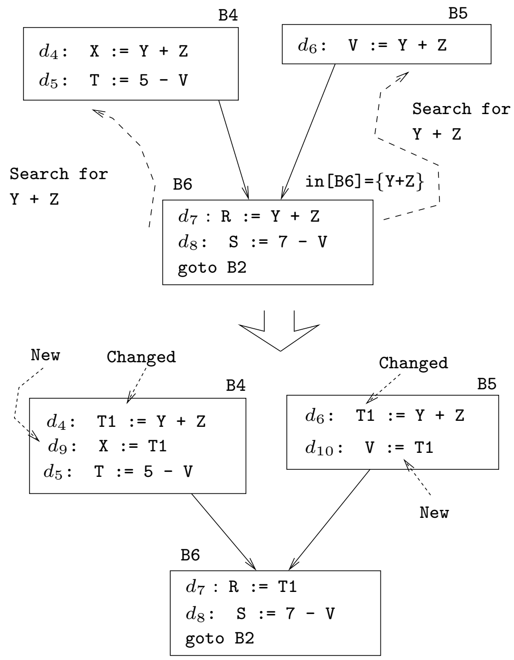
41 Global CSE III – Algorithm

Consider each statement $s : x := y + z$ in block B such that $(y + z) \in$ in[B].

1. Find all statements $d : R := y + z$ in B's ancestor blocks, such that $y + z$ could reach s.
2. Replace $d : R := y + z$ with $d : T_1 := y + z; R := T_1$.
3. Replace $s : x := y + z$ with $s : x := T_1$.



42



43

Summary

44 Readings and References

- Read the Tiger book: 220–225, 387–393
- Or read the Dragon book: 608–611, 622, 627–628, 633–635, 638–642.

45 Summary I

- Global Data-Flow Analysis tracks the flow of values and computations across basic block boundaries.
- Typically, before we can perform a particular global optimization, we have to collect (one or more kinds of) global data-flow information.
- The data-flow information is normally collected for each basic block of the flow-graph. We can then easily compute the relevant information for points **within** each basic block, on a need-to-know basis.

46 Summary II

- Data-flow problems are represented as **data-flow equations**. Each equation manipulates sets of computations:

  gen is the set of computations generated locally within a basic block.

kill is the set of computations generated outside the block, which could possibly be invalidated by computations within the block.

in is the set of computations available at the beginning of a block.

out is the set of computations available at the end of a block.

- `gen[B]` and `kill[B]` are generated locally for each basic block `B`, without ever looking at computations outside the block.

47 Summary III

- `in[B]` and `out[B]` store the actual global data-flow information for the block `B`. They are computed from `gen[B]` and `kill[B]` and from `in` and `out` of neighboring blocks.
- A typical data-flow equation for some problem `P` might look like this:

$$\mathtt{out}_{\mathtt{P}}[\mathtt{B}] = \mathtt{gen}_{\mathtt{P}}[\mathtt{B}] \cup (\mathtt{in}_{\mathtt{P}}[\mathtt{B}] - \mathtt{kill}_{\mathtt{P}}[\mathtt{B}])$$

It says that the computations available at the the end of the block (`out[B]`) are those available at the entrance of the block, plus those generated within the block, except those computations generated outside the block that were invalidated by computations inside the block.

48 Summary IV

- Keep in mind that a global optimizer will solve many different data-flow problems, all involving sets called `gen, kill, in` and `out`, but that each problem **has its own** sets and equations. So, for a particular block `B`, we may have $\mathtt{in}_{\mathtt{ReachDef}}[\mathtt{B}]$, $\mathtt{out}_{\mathtt{ReachDef}}[\mathtt{B}]$, $\mathtt{gen}_{\mathtt{ReachDef}}[\mathtt{B}]$, $\mathtt{kill}_{\mathtt{ReachDef}}[\mathtt{B}]$, and $\mathtt{in}_{\mathtt{AvailExpr}}[\mathtt{B}]$, $\mathtt{out}_{\mathtt{AvailExpr}}[\mathtt{B}]$, $\mathtt{gen}_{\mathtt{AvailExpr}}[\mathtt{B}]$, $\mathtt{kill}_{\mathtt{AvailExpr}}[\mathtt{B}]$, etc.
- Similary, each data-flow problem sets up its own set of unique data-flow equations, although these equations may look remarkably similar from one problem to the next:

49 Summary V

$$\begin{aligned}
\mathtt{out}_{\mathtt{ReachD}}[\mathtt{B}] &= \mathtt{gen}_{\mathtt{ReachD}}[\mathtt{B}] \cup \\
&\ (\mathtt{in}_{\mathtt{ReachD}}[\mathtt{B}] - \mathtt{kill}_{\mathtt{ReachD}}[\mathtt{B}]) \\
\mathtt{in}_{\mathtt{ReachD}}[\mathtt{B}] &= \bigcup_{P} \mathtt{out}_{\mathtt{ReachD}}[\mathtt{P}] \\
\mathtt{out}_{\mathtt{AvailE}}[\mathtt{B}] &= \mathtt{gen}_{\mathtt{AvailE}}[\mathtt{B}] \cup \\
&\ (\mathtt{in}_{\mathtt{AvailE}}[\mathtt{B}] - \mathtt{kill}_{\mathtt{AvailE}}[\mathtt{B}]) \\
\mathtt{in}_{\mathtt{AvailE}}[\mathtt{B}] &= \bigcup_{P} \mathtt{out}_{\mathtt{ReachD}}[\mathtt{P}] \\
\mathtt{in}_{\mathtt{AvailE}}[\mathtt{B}_1] &= \{\}
\end{aligned}$$

50 Summary VI

- We have seen two data-flow problems:

Reaching Definition Analysis is used to build ud-chains, which in turn are used to detect loop-invariant computations (computations that produce the same value regardless of how many times the loop is executed).

Available Expression Analysis is used to optimize global common subexpressions. For each point p in the program we build the set of expressions that must have been computed, and whose value must still be current.

51 Summary VII

- There are many other data-flow problems:

Live-Variable Analysis is used during code-generation to avoid having to save a value stored in a register, if that value has no future use in the program. A live variable, on the other hand holds a value which may be used later on.

Definition-Use Analysis builds **definition-use chains**, lists which for each definition of a variable v holds the possible uses of v's value. `du`-chains are needed in order to perform **copy propagation**.

52

Homework

53 Homework I

- Build the flow-graph and ud-chains for the procedure body below. Then detect invariant computations, and — if possible — move them out of the loop.

```
K := 1; I := 2;
REPEAT
   IF I = 4 THEN
      A := K + 1;
   ELSE
      A := K + 2;
      I := I + A;
   ENDIF;
UNTIL I <= 10;
K := K + A;
```

54

Exam Problems

55 Exam Problem I (a) [07.430 '95]

- An expression E is *very busy* if – regardless of which path we take through the flow graph – E's value will be used before it is killed. Example:

```
(1)   BEGIN
(2)      IF expr THEN
(3)         V := A + 3;
(4)         R := K + 3;
(5)      ELSE
(6)         Z := A + 3;
(7)         K := 5;
(8)         L := K + 3;
(9)      END;
(10)  END
```

56 Exam Problem I (b) [07.430 '95]

- `A+3` is *very busy* at `(2)`, since – regardless of which branch of the **IF**-statement we take – `A+3` will be used before it is killed. However, `K+3` is *not* very busy, since (in the **ELSE**-branch) it is being killed by the assignment to `K` (at `(7)`) before it is being used.
- Very busy expression information is useful for register allocation (very busy expressions will always be computed, and should probably be allocated to a register) and when performing *code hoisting*, i.e. moving code to a common ancestor in the flow graph. Code hoisting applied to the example above would produce the following:

57 Exam Problem I (b) [07.430 '95]

```
(1)    BEGIN
(1.5)     T$0 := A + 3;
(2)       IF expr THEN
(3)          V := T$0;
(4)          R := K + 3;
(5)       ELSE
(6)          Z := T$0;
(7)          K := 5;
(8)          L := K + 3;
(9)       END;
(10)   END
```

Data-Flow Equations:

- The data-flow equations for computing very busy expressions are:

58 Exam Problem I (c) [07.430 '95]

$$\begin{aligned} \texttt{in}[B] &= \texttt{used}[B] \cup (\texttt{out}[B] - \texttt{killed}[B]) \\ \texttt{out}[B] &= \bigcap_{\substack{\texttt{successors} \\ S \texttt{ of } B}} \texttt{in}[S] \end{aligned}$$

out[B]	is the set of all expressions that are very busy at the end of the basic block `B`.
in[B]	is the set of all expressions that are very busy at the beginning of `B`.
used[B]	is the set of all expressions that are used before they are killed in `B`.
killed[B]	is the set of all expressions that are killed (i.e. their value is invalidated) before they are used in `B`.

59 Exam Problem I (d) [07.430 '95]

- Consider the following routine:

```
BEGIN
   X := 5; Y := 10;
   IF e1 THEN
      IF e2 THEN
         A := X * Y;
      ELSE
         B := 3;
         V := X * Y;
         X := 1;
      END;
   ELSE
      Y := 2; A := X * Y;
   END
END
```

60 Exam Problem I (e) [07.430 '95]

1. Construct the control-flow graph for the routine.
2. Construct the `used[B]` and `killed[B]` sets for each basic block of the control-flow graph.
3. Construct the resulting `in[B]` and `out[B]` sets for each basic block of the control-flow graph.