The Role of Program Analysis

Program analysis must precede many optimizations.
– Control flow analysis determines where program execution goes next, what is dependent on branches.
– Data flow analysis determines how program variables are affected by program sections.
– Data-dependence analysis determines which data references in a program access the same storage location.
(Sometimes data flow analysis is used as a generic term for all these analyses)

Control Flow Analysis
Control flow imposes dependences
– the execution of a code block must wait until the control conditions have been evaluated.
– This is true even if there are no data dependences.

Exercise: draw the control flow graph of this program segment

\[\begin{align*}
&\text{d=100} \\
&\text{IF (a==b) THEN c=d+e} \\
&\text{ELSE} \\
&\text{IF (c==e) THEN d=0} \\
&\text{ENDIF} \\
&b=b*2 \\
&\text{DO i=d,e} \\
&\text{a}[i]=0 \\
&\text{ENDDO}
\end{align*}\]

Simple Control Flow Analysis

A
IF (cond) THEN
B
ELSE
C
ENDIF
D

Data flow analysis takes place over a control flow graph

• Data flow analysis attempts to track, at compile time, the flow of variable values, pointer values, expression types, etc.
• At compile time, we don’t know the effect of control flow, and we must approximate the behavior of the program at “join” points in the program.
• Intra-procedural parts of algorithm are either “flow sensitive” or “flow insensitive” depending on how they treat branches, and joins from branches.
• Inter-procedural algorithms are either “context sensitive” or “context insensitive” depending on how they treat information flowing into (and out of) function calls.
Data flow equations

• At each point in a program, the effect of data flow is modeled by examining the information (not necessarily variable values) that
  – comes in to the block/statement,
  – is generated by the block/statement,
  – is killed by the block/statement,
  – goes out of the block/statement.

• \( \text{out}[B] = \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B]) \)

Global Dataflow Analysis

Analysis of how program attributes change across basic blocks

Dataflow analysis has many applications.

A few examples:

– global live variable analysis
– uninitialized variable analysis
– available expression analysis
– busy expression analysis (used along all paths before being killed)

Aside: Finding Local Def, Use

• determining Def and Use sets in a basic block is easy.

Exercise: determine Def, Use sets for
\[
\begin{align*}
  c &= d + e \\
  d &= a[j] \\
  a[j] &= sb[i, k] \\
  *p &= *q
\end{align*}
\]
Sound and unsound analyses

- A safe or sound analysis is one in which any imprecision reduces the opportunities to do a transformation or optimization, but doesn’t cause incorrect transformations or optimizations to be performed.
- An unsafe or unsound analysis may allow incorrect optimizations to occur.
- Conservative assumptions often necessary for an analysis to be sound, e.g. may need to treat “may alias” as aliased, or not aliased, depending on what transformation is being used.

Live Variable Analysis

Application: the value of dead variables need not be saved at the end of a basic block

A := B + C
\[ \ldots = A \]

LiveOut(b) \(\subseteq\) LiveUse(b) \(\subseteq\) Gen

LiveIn(b) \(\subseteq\) LiveOut(b) - Def(b) \(\subseteq\) Kill

LiveIn(b) = LiveUse(b) \(\cup\) (LiveOut(b) - Def(b))

This is a backward-flow problem

Live Variable Analysis - kills along one path do not kill along all paths

A := 1
if A=B then
  B := 1
else
  C := 1
end if
D := A+B

A:=1
A=B

LiveOut(b) \(\subseteq\) LiveIn(l) \(\subseteq\) S

LiveIn(b) \(\subseteq\) LiveUse(b)
LiveIn(b) \(\subseteq\) LiveOut(b) - Def(b)

LiveIn(b) = LiveUse(b) \(\cup\) (LiveOut(b) - Def(b))

Def is kill
LiveUse is Gen
Live Variable Analysis – kills along all paths kill

A := 1
if A=B then
B := 1
else
C := 1
end if
D := A+B

LiveOut(b) = \cup \text{LiveIn}(i)

LiveIn(b) \subseteq \text{LiveUse}(b)
LiveIn(b) \subseteq \text{LiveOut}(b) - \text{Def}(b)

block Def LiveUse
b1 (A) (B)
b2 (B) \emptyset
b3 (B) \emptyset
b4 (D) (A,B)

block LiveIn LiveOut
b1 (B) (A)
b2 (A) (A,B)
b3 (A) (A,B)
b4 (A,B) \emptyset

Uninitialized Variable Analysis

Determine variables that may not be initialized

\text{Predecessors } P
\text{Init}(b): \text{variables known to be initialized}
\text{Uninit}(b): \text{variables that become uninitialized}
\text{Uninit}(b) \text{ is Kill}
\text{Uninit}(b) \text{ is Gen}
\text{Uninit}(b) = (\text{Uninit}(b) - \text{Init}(b)) \cup \text{Uninit}(b)

This is a forward-flow problem

Solving Data Flow Equations

- Acyclic CFG (trivial case):
  - Start at first node, then successively solve equations for successors or predecessors (depending on forward or backward flow problem.)
- Iterative Solutions (cycles in the CFG):
  - Begin with the sets computed locally for each basic block (generically called the Gen and Kill sets)
  - Iterate until the In and Out sets converge
  - Is convergence guaranteed?
  - yes, for dataflow graphs with (i) a unique starting node and one or more ending nodes, and (ii) whose sets can be represented as bit vectors.
- Solutions specific to structured languages:
  - exploit knowledge about the language constructs that build flow graphs: if and loop statements

Any-Path Flow Problems

- So far, we have considered “any-path” problems: a property holds along some path, i.e. along any of the paths
  - variable is uninitialized for basic block if it is uninitialized after any predecessor of \( b \)
  - variable is live if it is used in any successor
All-Paths Flow Problems

- All-Paths problems require a property to hold along all possible paths.
  Examples:
  - Availability of expressions
    \[ \text{AvailIn}(b) = \bigcap \text{AvailOut}(i) \]
    \[ \text{AvailOut}(b) = \text{Computed}(b) \cup (\text{AvailIn}(b) \setminus \text{Killed}(b)) \]
  - Very-busy expressions
    \[ \text{VeryBusyOut}(b) = \bigcap \text{VeryBusyIn}(i) \]
    \[ \text{VeryBusyIn}(b) = \text{Used}(b) \cup (\text{VeryBusyOut}(b) \setminus \text{Killed}(b)) \]

Exercise: Available Expression Analysis

\[
egin{align*}
  & b_1: \text{c = x} \\
  & b_2: \text{e = 5} \\
  & b_3: \text{a = b+c/d} \\
  & b_4: \text{e = f - g} \\
  & b_5: \text{g = b+c} \\
  & b_6: \text{c = 5} \\
  & b_7: \text{e = c/d} \\
  & b_8: \text{a = f - g} \\
  & b_9: \text{b = c/d + a}
\end{align*}
\]

Notes:
- Precise definition of the meaning of the information to be analyzed is important. Here: an available expression is an expression (given by its relevant variables and operators) whose up-to-date value is available in a temporary variable.
- Note that the value of an available expression from two merged control paths is not necessarily the same. E.g., the value of \( c/d \) in \( b_4 \) depends on the control path taken.
- Be clear about the information sets to be used in the data flow analysis. Here: the sets of all expressions and subexpressions.

General Data Flow Equations

Forward Flow

\[
\begin{align*}
\text{Out}(b) &= \text{Gen}(b) \cup (\text{In}(b) \setminus \text{Killed}(b)) \\
\text{In}(b) &= \bigcup_{i \in P(b)} \text{Out}(i)
\end{align*}
\]

Backward Flow

\[
\begin{align*}
\text{In}(b) &= \text{Gen}(b) \cup (\text{Out}(b) \setminus \text{Killed}(b)) \\
\text{Out}(b) &= \bigcup_{i \in S(b)} \text{In}(i)
\end{align*}
\]

Other Data Flow Problems and Applications of DF Analyses

- Reaching Definitions (Use-Def Chains)
  - determining use points of assigned values
- Def-Use Chains
  - Can be computed from U-D chains or as a new data flow problem.
- Constant Propagation
  - determining variables that hold constant values.
    - Can be computed based on Reaching Definition information or with an extended data flow analysis
- Copy propagation
  - replacing variables by their assigned expressions and eliminating assignments. A more advanced form of constant propagation.
Optimizations at Subroutine Calls

- Inlining
  - Saves call overhead for small subroutines.
  - Eliminates the need for interprocedural analysis
- Interprocedural analysis
  - eliminates (or reduces) conservative assumptions at subroutine boundaries, such as
    - the assumption that all variables that can be seen by a subroutine (parameters, global variables) are read and written.
    - the assumption that all subexpressions are killed

Interprocedural Analysis

IPA propagates knowledge gathered in one subroutine to the others. This analysis often needs to be iterative.

Example:
constant propagation

\[
\begin{align*}
a &= 3 \\
call \text{sub1}(a) \\
a &= a + 1 \\
call \text{sub1}(a)
\end{align*}
\]

Interprocedural Analysis – context sensitive (CS) analysis

Can either analyze calls with union of all information (i.e. $x=3$ OR $x=4$), or can analyze with context $x=3$ for the first call, and context $x=4$ for the second call.

CS takes longer, but might discover facts that lead to guidance for inlining, and more optimization.

I.e. with CS, can clone, or inline first call, because we know “huge computation” will vanish.

Example:
constant propagation

\[
\begin{align*}
a &= 3 \\
call \text{sub1}(a) \\
a &= a + 1 \\
call \text{sub1}(a)
\end{align*}
\]

\[
\begin{align*}
\text{subroutine sub1}(x) \\
\text{...} \\
\text{if} \ (x < 4) \ {\}
\text{a huge computation} \\
\text{...}
\end{align*}
\]

Interprocedural Analysis – context insensitive (CI) analysis

Analyze both call with union of all information (i.e. $x=3$ OR $x=4$),

(CI) is faster, but may not discover some optimization routine
because of the merging of information.

With CS analyses, can clone, or inline first call, because we know “huge computation” will vanish, but need to analyze sub1 twice.
Call graphs represent calling structure of a program

- Like flow graphs, only represent calling structure. Nodes may contain a CFG.

#define graphs

    Call graphs represent calling structure of a program
    \[\text{Like flow graphs, only represent calling structure. Nodes may contain a CFG.}\]

    \begin{center}
    \begin{tikzpicture}[scale=0.8]
    \node (a) at (2,2) {a()};
    \node (b) at (4,2) {b()};
    \node (c) at (4,0) {c()};
    \node (d) at (6,0) {d()};
    \path
    (a) edge [->] node [above] {2 or 3} (b)
    (b) edge [->] node [above] {2 or 3} (c)
    (c) edge [->] node [above] {1} (d)
    (d) edge [->] node [above] {4} (a)
    ;
    \end{tikzpicture}
    \end{center}

    - Cycles indicate recursive calls
    - Cycles imply iteration when solving data flow equations
    - Strongly connected components involving an edge will contain recursive calls
    - Leaf nodes have no outgoing edges
    - Numbers indicate bottom up traversal order

Finding Local Def, Use

- determining Def and Use sets in a basic block is easy.
  - Def, Use sets
  - \( d = d + e \)
  - \( a[i] = sb[i,k] \)
  - \( *p = *q \)

- Branches can either be handled by unioning info from all predecessors, or by being flow-invariant (ignore branches)

Definition and Use sets

Building Definition and Use sets
- Def set: the set of variables written to
- Use set: the set of variables read
these sets are used by many optimizations

Interprocedural Dataflow Analysis

Building Definition and Use sets
- Def set: the set of variables written to
- Use set: the set of variables read
these sets are used by many optimizations

Given LocalUse and LocalDef for each subroutine:
- \( \text{Use}(P) = \text{LocalUse}(P) \cup \text{Use}(Q) \)
- \( \text{Def}(P) = \text{LocalDef}(P) \cup \text{Def}(Q) \)
  \( Q \in \text{called}(P) \)
Algorithm for Computing Def, Use Sets Interprocedurally

- in non-recursive programs:
  - compute Def, Use sets bottom-up in call tree
- in recursive programs:
  - iterate until the sets don’t change
  - (Algorithm on page 632)

Example, page 632

Declare a,b,c,d,e: integer;
Procedure Q is
  write(a);
End Q;

Procedure P is
  a := b + c;
  Q;
End P

Begin
  c := a+b;
  e := c+d;
  P; 
  . . .
End

localUse(P) = \{b,c\}
localUse(Q) = \{a\}
localDef(P) = \{a\}
localDef(Q) = empty set

Approximate Def and Use by localDef and localUSE, i.e.

Use(P) = \{b,c\}
Use(Q) = \{a\}
Def(P) = \{a\}
Def(Q) = empty set

Because P calls Q, Q’s Def’s and Use’s must be included in P’s sets …

Example, page 632

Declare a,b,c,d,e: integer;
Procedure Q is
  write(a);
End Q;

Procedure P is
  a := b + c;
  Q;
End P

Begin
  c := a+b;
  e := c+d;
  P; 
  . . .
End

localUse(P) = \{b,c\}
localUse(Q) = \{a\}
localDef(P) = \{a\}
localDef(Q) = empty set

Approximate Def Use by localDef and localUSE

Use(P) = \{b,c\}
Use(Q) = \{a\}
Def(P) = \{a\}
Def(Q) = empty set

After update with called routines info

Use(P) = \{a,b,c\}
Use(Q) = \{a\}
Def(P) = \{a\}
Def(Q) = empty set

Note that this is flow insensitive!
Example, page 632

Declare a, b, c, d, e: integer;
Procedure Q is
  write(a);
End Q;

Procedure P is
  a := b + c;
  Q;
End P

Begin
  c := a + b;
  e := c + d;
  P; . . .
End

We know that:
- After the call to P, a + b has been killed but c + d is still valid
- If \( a, b, c, d \) and e were in registers, we need to save a, b, c because they are used (and therefore alive)
- Note that better dataflow analysis (see 16.4) could determine that
  - Value of a in the main program is killed in P before being used in Q
  - This would mean value of a does not need to be saved before calling P

Use(P) = \{a, b, c\} 
Use(Q) = \{a\} 
Def(P) = \{a\} 
Def(Q) = empty set

Factoring Loop-invariant Expressions

- Loop-invariant expressions can be moved in front of the loop.
- The idea is similar to common subexpression elimination (reuse computation), however, the actions are different:
  - Find Def Sets for the loop
  - Find relevant variables of expressions (i.e., the variables that are part of the expression)
  - If the two sets are disjoint, it is a loop-invariant expr.

Factoring Loop-invariant Expressions

- Array address expressions are important subjects of this optimization, although this is not obvious from the program text
  - For example, in \( A(i,j,k) \)
    - The implicit expression is \( \text{baseA} + i\times\text{rowsz}\times\text{eltsz} + j\times\text{eltsz} + k\times\text{eltsz} \) (j-invariant, k-invariant)
- Safety and profitability is not always guaranteed
  - Zero-trip loops may never execute the expression adding a guard if in front of expression may satisfy safety, but not profitability requirements

Example

\[
\begin{align*}
\text{do } i = 1, n & \\
\text{do } j = 1, n & \\
\text{do } k = 1, n & \\
\text{a}(i,j,k) = \ldots & \\
\text{end do} & \\
\text{end do} & \\
\text{end do} & \\
\text{a}(i,j,k) \text{ is really} & \\
\text{a}(i\times\text{rowsz} + j\times\text{eltsz} + k\times\text{eltsz}) & \\
\text{Use(j-loop)} = \{i,\text{rowsz},\text{eltsz},j,k,\text{baseA},*'a\} & \\
\text{Def(j-loop)} = \{j,k,*'a\} \\
\text{Use(k-loop)} = \{i,\text{rowsz},\text{eltsz},j,k,\text{baseA},*'a\} & \\
\text{Def(k-loop)} = \{k,*'a\}
\end{align*}
\]