CS293S Iterative Data-Flow Analysis

Yufei Ding
Review: Computing Available Expressions

The Big Picture

1. Build a control-flow graph
2. Gather the initial data: \text{DEEXPR}(b) \& \text{EXPRKILL}(b)
3. Propagate information around the graph, evaluating the equation

\[
\text{AVAIL}(b) = \cap_{x \in \text{pred}(b)} \ (\text{DEEXPR}(x) \cup \text{AVAIL}(x) \cap \text{EXPRKILL}(x))
\]

Entry point of block b \quad Exit point of block x

Works for loops through an iterative algorithm: finding the fixed-point.

All data-flow problems are solved, essentially, this way.
Live Variables

A variable \( v \) is live at a point \( p \) if there is a path from \( p \) to a use of \( v \), and that path does not contain a redefinition of \( v \).

Example: \( I: a \leftarrow b + c \)

A statement/instruction \( I \) is a definition of a variable \( v \) if it may write to \( v \). \( \text{def}[I] = a \)

A statement is a use of variable \( v \) if it may read from \( v \). \( \text{use}[I] = \{ b, c \} \)

```
  e = b + c
  c = x + y

  a = b + c
  c = a

  e = a
  c = e

  a = e + c
```

Point \( p \)
**Live Variables**

A variable \( v \) is **live** at point \( p \) if and only if there is a path from \( p \) to a use of \( v \) along which \( v \) is not redefined.

**Usage**

- Global register allocation
- Improve SSA construction
  - reduce # of f-functions
- Detect references to uninitialized variables & defined but not used variables
- Drive transformations
  - useless-store elimination
Live Variables at Special Points

For an instruction I
LIVEIN[I]: live variables at program point before I
LIVEOUT[I]: live variables at program point after I

For a basic block B
LIVEIN[B]: live variables at the entry point of B
LIVEOUT[B]: live variables at the exit point of B

If I = first instruction in B, then LIVEIN[B] = LIVEIN[I]
If I = last instruction in B, then LIVEOUT[B] = LIVEOUT[I]
How to Compute Liveness?

Question 1: for each instruction I, what is the relation between \( \text{LIVEIN}[I] \) and \( \text{LIVEOUT}[I] \)?

Question 2: for each basic block B, what is the relation between \( \text{LIVEIN}[B] \) and \( \text{LIVEOUT}[B] \)?

Question 3: for each basic block B with successor blocks \( B_1, \ldots, B_n \), what is the relation between \( \text{LIVEOUT}[B] \) and \( \text{LIVEOUT}[B_1], \ldots, \text{LIVEOUT}[B_n] \)?
Part 1: Analyze Instructions

Question: what is the relation between the sets of live variables before and after an instruction \( I \)?

Examples:

<table>
<thead>
<tr>
<th>LIVEIN([I]) = {y,z}</th>
<th>LIVEIN([I]) = {y,z,t}</th>
<th>LIVEIN([I]) = {x,t}</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = y+z; )</td>
<td>( x = y+z; )</td>
<td>( x = x+1; )</td>
</tr>
<tr>
<td>LIVEOUT([I]) = {z}</td>
<td>LIVEOUT([I]) = {x,t}</td>
<td>LIVEOUT([I]) = {x,t}</td>
</tr>
</tbody>
</table>

... is there a general rule?
Analyze Instructions

Two Rules:

Each variable live after I is also live before I, unless I defines (writes) it.

Each variable that I uses (reads) is also live before instruction I

Mathematically:
\[ \text{LIVEIN}[I] = ( \text{LIVEOUT}[I] - \text{def}[I] ) \cup \text{use}[I] \]

where: \( \text{def}[I] \) = variables defined (written) by instruction I
\( \text{use}[I] \) = variables used (read) by instruction I

The information flows **backward!**
Analyze block

Example: block B with three instructions I1, I2, I3:

Live1 = LIVEIN[B] = LIVEIN[I1]
Live2 = LIVEOUT[I1] = LIVEIN[I2]
Live3 = LIVEOUT[I2] = LIVEIN[I3]
Live4 = LIVEOUT[I3] = LIVEOUT[B]

Relation between Live sets:

\[
\text{Live1} = ( \text{Live2} - \{x\} ) \cup \{y\}
\]
\[
\text{Live2} = ( \text{Live3} - \{y\} ) \cup \{x,z\}
\]
\[
\text{Live3} = ( \text{Live4} - \{t\} ) \cup \{d\}
\]
Analyze Block

Two Rules:

Each variable live after B is also live before B, unless B defines (writes) it.

Each variable v that B uses (reads) before any redefinition in B is also live before B

Mathematically:
LIVEIN[B] = (LIVEOUT[B] − VarKill(B)) ∪ UEVar(B)

where:

VARKILL(B) = variables that are defined in B
UEVAR(B) = variables that are used in B before any redefinition in B, i.e., upward-exposed variables
Analyze CFG

Question: for each basic block $B$ with successor blocks $B_1$, ..., $B_n$, what is the relation between $\text{LIVEOUT}[B]$ and $\text{LIVEIN}[B_1]$, ..., $\text{LIVEIN}[B_n]$?

Example:

```
B
\{x,y,z\}

{z} B_1
{y} B_2
{z} B_3
```

General rule?
Analyze CFG

Rule: A variables is live at end of block B if it is live at the beginning of one (or more) successor blocks

Mathematically:

\[ \text{LIVEOUT}[B] = \bigcup_{B' \in \text{succ}(B)} \text{LIVEIN}[B'] \]

\[ = \bigcup_{B' \in \text{succ}(B)} ((\text{LIVEOUT}[B'] - \text{VARKILL}(B')) \bigcup \text{UEVAR}(B')) \]

Again, information flows **backward:** from successors B’ of B to basic block
Equations for Live Variables

\text{LIVEOUT}(B) \text{ contains the name of every variable that is live on exit from n (a basic block)}

\text{UEVAR}(B) \text{ contains the upward-exposed variables in n, i.e. those that are used in n before any redefinition in n}

\text{VARKILL}(B) \text{ contains all the variables that are defined in n}

Equation (n_f is the exit node of the CFG)

\[
\text{LIVEOUT}[B] = \bigcup_{B' \in \text{succ}(B)} ((\text{LIVEOUT}[B'] - \text{VARKILL}(B')) \bigcup \text{UEVAR}(B'))
\]

Note: \( A - B = A^{\cup} \overline{B} \)
Three Steps in Data-Flow Analysis

Build a CFG

Gather the initial information for each block (i.e., (UEVAR and VARKILL))

Use an iterative fixed-point algorithm to propagate information around the CFG
Algorithm

// Get initial sets

for each block b
    UEVAR(b) = Ø
    VARKILL(b) = Ø
for i=1 to number of instr in b
    (assuming inst I is “x = y op z”)
    if y ∉ VARKILL(b) then
        UEVAR(b) = UEVAR(b) ∪ {y}
    if z ∉ VARKILL(b) then
        UEVAR(b) = UEVAR(b) ∪ {z}
    VARKILL(b) = VARKILL(b) ∪ {x}

// update LiveOut version 1

set LIVEOUT(b_i) to Ø for all blocks
Worklist ← {all blocks}
while (Worklist ≠ Ø)
    remove a block b from Worklist
    recompute LIVEOUT(b)
    if LIVEOUT(b) changed then
        Worklist ← Worklist ∪ pred(b)

LIVEOUT[B] = \bigcup_{B'\in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \cup UEVAR(B'))
Algorithm

// Get initial sets

for each block b
    UEVAR(b) = Ø
    VARKILL(b) = Ø

for i=1 to number of instr in b
    (assuming inst I is “x= y op z”)
    if y \notin VARKILL(b) then
        UEVAR(b) = UEVAR(b) ∪ \{y\}
    if z \notin VARKILL(b) then
        UEVAR(b) = UEVAR(b) ∪ \{z\}
    VARKILL(b) = VARKILL(b) ∪ \{x\}

// update LiveOut version2

set LIVEOUT(b_i) to Ø for all blocks
changed = true
while (changed)
    changed = false
    for i = 1 to N (number of blocks)
        recompute LIVEOUT(i)
        if LIVEOUT(i) changed then
            changed = true

\[
LIVEOUT[B] = \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] \setminus VARKILL[B']) \cup UEVAR[B'])
\]
Example

```
B_0: i ← 1
    i <= 100
    i > 100

B_1:
    a ← ...
    c ← ...

B_2:
    b ← ...
    c ← ...
    d ← ...

B_3:
    a ← ...
    d ← ...

B_4:
    d ← ...

B_5:
    c ← ...

B_6:
    b ← ...

B_7:
    y ← a + b
    z ← c + d
    i ← i + 1
    i <= 100
    i > 100
```
### Example (cont.)

<table>
<thead>
<tr>
<th></th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>B6</th>
<th>B7</th>
</tr>
</thead>
<tbody>
<tr>
<td>UEVar</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>a, b, c, d, i</td>
</tr>
<tr>
<td>VarKill</td>
<td>i</td>
<td>a, c</td>
<td>b, c, d</td>
<td>a, d</td>
<td>d</td>
<td>c</td>
<td>b</td>
<td>y, z, i</td>
</tr>
</tbody>
</table>


**Example (cont.)**

Can the algorithm converge in fewer iterations?

**LiveOut (b)**

<table>
<thead>
<tr>
<th>iteration</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>B6</th>
<th>B7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>1</td>
<td>Ø</td>
<td>Ø</td>
<td>a,b,c,d,i</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>a,b,c,d,i</td>
<td>Ø</td>
</tr>
<tr>
<td>2</td>
<td>Ø</td>
<td>a,i</td>
<td>a,b,c,d,i</td>
<td>Ø</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,b,c,d,i</td>
<td>i</td>
</tr>
<tr>
<td>3</td>
<td>i</td>
<td>a,i</td>
<td>a,b,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,b,c,d,i</td>
<td>i</td>
</tr>
<tr>
<td>4</td>
<td>i</td>
<td>a,c,i</td>
<td>a,b,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,b,c,d,i</td>
<td>i</td>
</tr>
<tr>
<td>5</td>
<td>i</td>
<td>a,c,i</td>
<td>a,b,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,b,c,d,i</td>
<td>i</td>
</tr>
</tbody>
</table>

\[
LIVEOUT[B] = \bigcup_{B' \in \text{succ}(B)} ((LIVEOUT[B']-\text{VARKILL}(B')) \bigcup \text{UEVAR}(B'))
\]
\[ LIV E O U T[B] = \bigcup_{B' \in \text{succ}(B)} ((LIV E O U T[B'] - V A R K I L L(B')) \cup U E V A R(B')) \]

Preorder: parents first.

w/o considering backedges.
$$LIVEOUT[B] = \bigcup_{B' \in \text{succ}(B)} ((LIVEOUT[B'] - \text{VARKILL}(B')) \bigcup \text{UEVAR}(B'))$$

Postorder: children first.

w/o considering backedges.
Algorithm

// Get initial sets

for each block b
    UEVAR(b) = Ø
    VARKILL(b) = Ø

for i=1 to number of instr in b
    (assuming inst I is “x= y op z”)
    if y \notin VARKILL(b) then
        UEVAR(b) = UEVAR(b) \cup \{y\}
    if z \notin VARKILL(b) then
        UEVAR(b) = UEVAR(b) \cup \{z\}
    VARKILL(b) = VARKILL(b) \cup \{x\}

// update LiveOut version2

set LIVEOUT(bi) to Ø for all blocks
changed = true
while (changed)
    changed = false
    for i = 0 to N
        // different orders could be used
        recompute LIVEOUT(i)
        if LIVEOUT(i) changed then
            changed = true

\[
LIVEOUT[B] = \bigcup_{B' \in succ(B)} \left((LIVEOUT[B'] - VARKILL(B')) \cup UEVAR(B')\right)
\]
Postorder (5 iterations becomes 3)

<table>
<thead>
<tr>
<th>iteration</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>B6</th>
<th>B7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>1</td>
<td>i</td>
<td>a,c,i</td>
<td>a,b,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,b,c,d,i</td>
<td>Ø</td>
</tr>
<tr>
<td>2</td>
<td>i</td>
<td>a,c,i</td>
<td>a,b,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,b,c,d,i</td>
<td>i</td>
</tr>
<tr>
<td>3</td>
<td>i</td>
<td>a,c,i</td>
<td>a,b,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,b,c,d,i</td>
<td>i</td>
</tr>
</tbody>
</table>
Order

Parent relation does not consider backedges.

**Preorder**: visit parents before children.

also called reverse postorder

**Postorder**: visit children before parents.

Forward problem (e.g., AVAIL):

A node needs the info of its predecessors.
Preorder on CFG.

Backward problem (e.g., LIVEOUT):

A node needs the info of its successors.
Postorder on CFG.
Comparison with AVAIL

Common
Three steps
Fixed-point algorithm finds solution

Differences
AVAIL: domain is a set of expressions
LIVEOUT: domain is a set of variables
AVAIL: forward problem
LIVEOUT: backward problem
AVAIL: intersection of all paths (all path problem)
  Also called Must Problem
LIVEOUT: union of all paths (any path problem)
  Also called May Problem
Other Data Flow Analysis
**Very Busy Expressions**

Def: $e$ is a very busy expression at the exit of block $b$ if 
- $e$ is evaluated and used along every path that leaves $b$, and 
evaluating $e$ at the end of $b$ produces the same result.

useful for code hoisting

saves code space

```
...  
...  
```

```
...  
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
e = a + b
```

...
Very Busy Expressions

VERYBUSY(b) contains expressions that are very busy at end of b
UEEXPR(b): up exposed expressions (i.e. expressions defined in b and not subsequently killed in b)
EXPRKILL(b): killed expressions

A backward flow problem, domain is the set of expressions

\[
\text{VERYBUSY}(b) = \cap_{s \in \text{succ}(b)} \text{UEEXPR}(s) \cup (\text{VERYBUSY}(s) \cap \text{EXPRKILL}(s))
\]

\[
\text{VERYBUSY}(n_f) = \emptyset
\]
Constant Propagation

Def of a constant variable \( v \) at point \( p \):
Along every path to \( p \), \( v \) has same known value
Specialize computation at \( p \) based on \( v \)'s value

\[
\begin{align*}
a &= 7; \\
c &= a \times 2; \\
b &= c - a; \\
a &= 9; \\
d &= c - a; \\
e &= c - b; \\
b &= a;
\end{align*}
\]
Constant Propagation: Another Data Flow Problem

Domain is the set of pairs \(<v_i, c_i>\) where \(v_i\) is a variable and \(c_i \in C\)

\[
\text{CONSTANTS}(b) = \bigwedge_{p \in \text{preds}(b)} f_p(\text{CONSTANTS}(p))
\]

\(\bigwedge\) performs a pairwise meet on two sets of pairs

\(f_p(x)\) is a block specific function that models the effects of block \(p\) on

the \(<v_i, c_i>\) pairs in \(x\)

A forward flow problem, domain is the set of pairs \(<v, c>\).

\(C\): constants or \(\bot\).

\(\bot\): non-constant or unknown value
\[ \text{CONSTANTS}(b) = \bigwedge_{p \in \text{preds}(b)} f_p(\text{CONSTANTS}(p)) \]

Meet operation \(<v, c_1> \land <v, c_2>\)

\(<v, c_1> \text{ if } c_1 = c_2, \text{ else } <v, \bot>\)

\(\bot: \text{non-constant or unknown value}\)

What about \(f_p\) ?

if \(p\) has only one statement, update the constant set with

the results if operands are all constants

\(\bot\) if the result is unknown or non-constant

If \(p\) has \(n\) statements then

\[ f_p(\text{CONSTANTS}(p)) = f_n(f_{n-1}(f_{n-2}(\ldots f_2(f_1(\text{CONSTANTS}(p)))\ldots))), \]

where \(f_i\) is the function generated by the \(i\)th statement in \(p\)
\[ \text{CONSTANTS}(b) = \bigwedge_{p \in \text{preds}(b)} f_p(\text{CONSTANTS}(p)) \]

Meet operation \(<v, c_1> \land <v, c_2>\)
\(<v, c_1>\) if \(c_1 = c_2\), else \(<v, \bot>\)

\(\bot\): non-constant or unknown value

Formal definition of \(p\):

If \(p\) has one statement then
\[ x \leftarrow y \text{ with } \text{CONSTANTS}(p) = \{\ldots<x,l_1>,\ldots<y,l_2>\ldots\} \]
then \(f_p(\text{CONSTANTS}(p)) = \{\text{CONSTANTS}(p) - <x,l_1>\} \cup <x,l_2>\)
\[ x \leftarrow y \text{ op } z \text{ with } \text{CONSTANTS}(p) = \{\ldots<x,l_1>,\ldots<y,l_2>\ldots ,\ldots<z,l_3>\ldots\} \]
then \(f_p(\text{CONSTANTS}(p)) = \{\text{CONSTANTS}(p) - <x,l_1>\} \cup <x, l_2 \text{ op } l_3>\)

If \(p\) has \(n\) statements then
\[ f_p(\text{CONSTANTS}(p)) = f_n(f_{n-1}(f_{n-2}(\ldots f_2(f_1(\text{CONSTANTS}(p))))\ldots))) \]
where \(f_i\) is the function generated by the \(i^{th}\) statement in \(p\)

\(f_p\) interprets \(p\) over \text{CONSTANTS}
Data-Flow Analysis Frameworks

Generalizes and unifies data flow problems.

Important components:

- **Direction D**: forward or backward.
- **A Semilattice**: a domain $V$ and a *meet* operator $\wedge$ that captures the effect of path confluence.
- **A transfer function $F(m)$**: compute the effect of passing through a basic block and include function value at boundary conditions.

\[
\text{A semilattice is an algebra } S = (S, \ast) \text{ satisfying, for all } x, y, z \in S,
\]

1. $x \ast x = x$,
2. $x \ast y = y \ast x$,
3. $x \ast (y \ast z) = (x \ast y) \ast z$.

33
Examples

(\(D, V, F, ^\))

LIVE

\(\bullet D: \) backward
\(\bullet V: \) all variables
\(\bullet F_m: \) \(UEVAR(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)})\) \(; \) \(\text{LIVEOUT}(n_f) = \phi\)
\(\bullet ^\land : \cup\)

AVAIL

\(\bullet D: \) forward, \(V: \) all expressions
\(\bullet F_m: \) \(\text{DEEXPR}(m) \cup (\text{AVAIL}(m) \cap \overline{\text{EXPRKILL}(m)})\) \(; \) \(\text{AVAIL}(n_0) = \phi\)
\(\bullet ^\land : \cap\)
## Summary

<table>
<thead>
<tr>
<th>Domain</th>
<th>Direction</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVAIL</td>
<td>Expressions</td>
<td>Forward</td>
</tr>
<tr>
<td>VERYBUSY</td>
<td>Expressions</td>
<td>Backward</td>
</tr>
<tr>
<td>CONSTANT</td>
<td>Pairs &lt;v,c&gt;</td>
<td>Forward</td>
</tr>
</tbody>
</table>
Why to Study Data Flow Analysis

Data-flow analysis

A collection of techniques for compile-time reasoning about the run-time flow of values.

Backbone of scalar optimizing compilers
Limitation of Data-Flow Analysis

Imprecision from pointers, and procedure calls
Assume all paths will be taken

If \( y \) is always no less than \( x \), \( x \) is not live before \( B_2 \). But data-flow analysis may not figure that out.