Dataflow Analysis Frameworks

CIS410/510 Program Analysis and Transformation

Live variable analysis

What are the live variables at each program point?

Method:
1. Define sets of live variables
2. Build constraints
3. Solve constraints

```plaintext
if (c)
x = y + 1;
y = 2 * z;
if (d)
x = y + z;

z = 1;
z = x;
```
Derive constraints

Constraints for each instruction:
\[
\text{in}[I] = (\text{out}[I] - \text{def}[I]) \cup \text{use}[I]
\]

Constraints for control flow:
\[
\text{out}[B] = \bigcup_{B' \in \text{succ}(B)} \text{in}[B']
\]
Initialization

Constraints:
\[ L_1 = L_2 \cup \{c\} \]
\[ L_2 = L_3 \cup L_{11} \]
\[ L_3 = (L_4\setminus\{x\}) \cup \{y\} \]
\[ L_4 = (L_5\setminus\{y\}) \cup \{z\} \]
\[ L_5 = L_6 \cup \{d\} \]
BB: \[ L_6 = L_7 \cup L_9 \]
\[ L_7 = (L_8\setminus\{x\}) \cup \{y,z\} \]
BB: \[ L_8 = L_9 \]
\[ L_9 = L_{10}\setminus\{z\} \]
BB: \[ L_{10} = L_1 \]
\[ L_{11} = (L_{12}\setminus\{z\}) \cup \{x\} \]

if (c)
\[ x = y + 1; \]
\[ y = 2 \cdot z; \]
if (d)
\[ x = y + z; \]
\[ z = 1; \]
\[ z = x; \]

Iteration 1

Per instruction:
\[ \text{in}[I] = (\text{out}[I]\setminus\text{def}[I]) \cup \text{use}[I] \]

Per basic block:
\[ \text{out}[B] = \bigcup_{B' \in \text{succ}(B)} \text{in}[B'] \]
**Iteration 2**

Per instruction:
in[I]= (out[I]-def[I]) U use[I]

Per basic block:
out[B] = \( \bigcup_{B' \in \text{succ}(B)} \text{in}[B'] \)

Constraints:

- \( L_1 = L_2 \cup \{c\} \)
- \( L_2 = L_3 \cup L_{11} \)
- \( L_3 = (L_4 - \{x\}) \cup \{y\} \)
- \( L_4 = (L_5 - \{y\}) \cup \{z\} \)
- \( L_5 = L_6 \cup \{d\} \)
  
  BB: \( L_6 = L_7 \cup L_9 \)
  
  \( L_7 = (L_8 - \{x\}) \cup \{y, z\} \)
  
  BB: \( L_9 = L_1 \)
  
  \( L_{10} = L_1 \)
  
  \( L_{11} = (L_{12} - \{z\}) \cup \{x\} \)

\[
\begin{align*}
L_2 &= \{x,y,z,c,d\} \\
L_3 &= \{y,z,c,d\} \\
L_4 &= \{x,z,c,d\} \\
L_5 &= \{x,y,z,c,d\} \\
L_6 &= \{x,y,z,c,d\} \\
L_7 &= \{y,z,c,d\} \\
L_8 &= \{x,y,c,d\} \\
L_9 &= \{x,y,c,d\} \\
L_{10} &= \{x,y,z,c,d\} \\
L_{11} &= \{x\} \\
L_{12} &= \{} \\
\end{align*}
\]

**Fixed point!**

Per instruction:
in[I]= (out[I]-def[I]) U use[I]

Per basic block:
out[B] = \( \bigcup_{B' \in \text{succ}(B)} \text{in}[B'] \)

Constraints:

- \( L_1 = L_2 \cup \{c\} \)
- \( L_2 = L_3 \cup L_{11} \)
- \( L_3 = (L_4 - \{x\}) \cup \{y\} \)
- \( L_4 = (L_5 - \{y\}) \cup \{z\} \)
- \( L_5 = L_6 \cup \{d\} \)
  
  BB: \( L_6 = L_7 \cup L_9 \)
  
  \( L_7 = (L_8 - \{x\}) \cup \{y, z\} \)
  
  BB: \( L_9 = L_1 \)
  
  \( L_{10} = L_1 \)
  
  \( L_{11} = (L_{12} - \{z\}) \cup \{x\} \)

\[
\begin{align*}
L_2 &= \{x,y,z,c,d\} \\
L_3 &= \{y,z,c,d\} \\
L_4 &= \{x,z,c,d\} \\
L_5 &= \{x,y,z,c,d\} \\
L_6 &= \{x,y,z,c,d\} \\
L_7 &= \{y,z,c,d\} \\
L_8 &= \{x,y,c,d\} \\
L_9 &= \{x,y,c,d\} \\
L_{10} &= \{x,y,z,c,d\} \\
L_{11} &= \{x\} \\
L_{12} &= \{} \\
\end{align*}
\]
Final result: sets of live variables at each program point

Characterize all executions

The analysis detects that there is an execution that uses the value x = y+1
Generalization

• Live variable analysis and detection of available copies are similar:
  – Define some information that they need to compute
  – Build constraints for the information
  – Solve constraints iteratively:
    • The information always “increases” during iteration
    • Eventually, it reaches a fixed point.

• We would like a general framework
  – Framework applicable to many other analyses
  – Live variable/copy propagation = instances of the framework

Dataflow analysis framework

• Dataflow analysis = a common framework for many compiler analyses
  – Computes some information at each program point
  – The computed information characterizes all possible executions of the program

• Basic methodology:
  – Describe information about the program using an algebraic structure called a lattice
  – Build constraints that show how instructions and control flow influence the information in terms of values in the lattice
  – Iteratively solve constraints
Constraint-based analysis

- Program analysis using constraints is divisible into
  - Constraint generation: produces constraints from a program text that give a declarative specification of the desired information about the program.
  - Constraint resolution (i.e., solving the constraints) then computes the desired information

- Advantages of this constraint-based approach
  - Constraints separate specification from implementation
    - soundness of an analysis can be proven solely on the basis of the constraint systems used – there is no need to resort to reasoning about a particular algorithm for solving the constraints
    - algorithms for solving classes of constraint problems can be presented and analyzed independent of any particular program analysis
  - Constraints yield natural specifications
    - Constraints are (usually) local – each piece of program syntax contributes its own constraints in isolation from the rest of the program
  - Constraints enable sophisticated implementations

History

- Previously, different analyses, such as dataflow or type equations (used for type inference in functional languages and template-style polymorphism in OO languages) were viewed as separate lines of research with their own techniques, problems, and terminology
- Today it is understood that these problems are related and can be treated as instances of a more general setting
  - Techniques from each of the classical algorithms can be combined to create new program analyses.
Partial order relations

- Lattice definition builds on the concept of a partial order relation

- A partial order \((P,\subseteq)\) consists of:
  - A set \(P\)
  - A partial order relation \(\subseteq\) that is:
    1. Reflexive \(x \subseteq x\)
    2. Anti-symmetric \(x \subseteq y, y \subseteq x \Rightarrow x = y\)
    3. Transitive: \(x \subseteq y, y \subseteq z \Rightarrow x \subseteq z\)

- Called a “partial order” because not all elements are comparable, in contrast with a total order, in which
  4. Total \(x \subseteq y \text{ or } y \subseteq x\)

Example

- \(P\) is \{red, blue, yellow, purple, orange, green\}
- \(\subseteq\)

\[
\begin{align*}
\text{red} & \subseteq \text{purple}, & \text{red} & \subseteq \text{orange}, \\
\text{blue} & \subseteq \text{purple}, & \text{blue} & \subseteq \text{green}, \\
\text{yellow} & \subseteq \text{orange}, & \text{blue} & \subseteq \text{green}, \\
\text{red} & \subseteq \text{red}, & \text{yellow} & \subseteq \text{green}, \\
\text{blue} & \subseteq \text{blue}, & \text{yellow} & \subseteq \text{yellow}, \\
\text{purple} & \subseteq \text{purple}, & \text{purple} & \subseteq \text{purple}, \\
\text{orange} & \subseteq \text{orange}, & \text{orange} & \subseteq \text{orange}, \\
\text{green} & \subseteq \text{green} & \text{green} & \subseteq \text{green}
\end{align*}
\]
Hasse diagrams

- A graphical representation of a partial order, where
  - $x$ and $y$ are on the same level when they are **incomparable**
  - $x$ is below $y$ when $x \leq y$ and $x \neq y$
  - $x$ is below $y$ and connected by a line when $x \leq y$, $x \neq y$, and there is no $z$ such that $x \leq z$, $z \leq y$, $x \neq z$, and $y \neq z$

![Hasse diagram](image)

Lower/upper bounds

- If $(P, \sqsubseteq)$ is a partial order and $S \subseteq P$, then:
  1. $x \in P$ is a lower bound of $S$ if $x \sqsubseteq y$, for all $y \in S$
  2. $x \in P$ is an upper bound of $S$ if $y \sqsubseteq x$, for all $y \in S$

- There may be multiple lower and upper bounds of the same set $S
Example, cont.

purple   orange   green
|        |        |
red     blue     yellow

red is lower bound for \{purple, orange\}
blue is lower bound for \{purple, green\}
yellow is lower bound for \{orange, green\}
no lower bound for \{purple, orange, green\}
no lower bound for \{red, blue\}
no lower bound for \{red, yellow\}
no lower bound for \{blue, yellow\}, etc.

purple is upper bound for \{red, blue\}
orange is upper bound for \{red, yellow\}
green is upper bound for \{orange, green\}
no upper bound for \{red, blue, yellow\} no upper bound for \{purple, orange\}
no upper bound for \{orange, green\}
no upper bound for \{purple, green\} etc.

Example, cont.

purple   orange   green
|        |        |
red'   red     blue     yellow

red is lower bound for \{purple, orange\}
blue is lower bound for \{purple, green\}
yellow is lower bound for \{orange, green\}
no lower bound for \{purple, orange, green\}
no lower bound for \{red, blue\}
no lower bound for \{red, yellow\}
no lower bound for \{blue, yellow\}, etc.

purple is upper bound for \{red, blue\}
orange is upper bound for \{red, yellow\}
green is upper bound for \{orange, green\}
no upper bound for \{red, blue, yellow\} no upper bound for \{purple, orange\}
no upper bound for \{orange, green\}
no upper bound for \{purple, green\} etc.

red' is also a lower bound for \{purple, orange\}
LUB and GLB

- Define least upper bound (LUB) and greatest lower bound (GLB) as follows:
- If \((P, \subseteq)\) is a partial order and \(S \subseteq P\), then:
  1. \(x \in P\) is GLB of \(S\) if:
     a) \(x\) is a lower bound of \(S\)
     b) \(y \subseteq x\), for any lower bound \(y\) of \(S\)
  2. \(x \in P\) is a LUB of \(S\) if:
     a) \(x\) is an upper bound of \(S\)
     b) \(x \subseteq y\), for any upper bound \(y\) of \(S\)
- ... are GLB and LUB unique?

Example, cont.

```
    purple    orange    green
    |          |            |
red     blue     yellow
```

- \textit{red} is GLB for \{\textit{purple}, \textit{orange}\}
- \textit{blue} is GLB for \{\textit{purple}, \textit{green}\}
- \textit{yellow} is GLB for \{\textit{orange}, \textit{green}\}
- \textit{purple} is LUB for \{\textit{red}, \textit{blue}\}
- \textit{orange} is LUB for \{\textit{red}, \textit{yellow}\}
- \textit{green} is LUB for \{\textit{orange}, \textit{green}\}
Example’

\[
\begin{align*}
\text{purple} & \quad \text{orange} & \quad \text{green} \\
\text{red’} & \quad \text{red} & \quad \text{blue} & \quad \text{yellow}
\end{align*}
\]

\begin{itemize}
  \item \textit{blue} is GLB for \{\textit{purple}, \textit{green}\}
  \item \textit{yellow} is GLB for \{\textit{orange}, \textit{green}\}
  \item \textit{red’} is a lower bound for \{\textit{purple}, \textit{orange}\}
  \item \textit{red} is a lower bound for \{\textit{purple}, \textit{orange}\}
  \item There is no GLB for \{\textit{purple}, \textit{orange}\}
\end{itemize}

\begin{itemize}
  \item \textit{purple} is LUB for \{\textit{red}, \textit{blue}\}
  \item \textit{orange} is LUB for \{\textit{red}, \textit{yellow}\}
  \item \textit{green} is LUB for \{\textit{orange}, \textit{green}\}
  \item \textit{purple} is LUB for \{\textit{red’}, \textit{blue}\}
  \item \textit{orange} is LUB for \{\textit{red’}, \textit{yellow}\}
\end{itemize}

\textit{Lattices}

\begin{itemize}
  \item A pair \((L, \sqsubseteq)\) is a lattice if:
    \begin{enumerate}
      \item \((L, \sqsubseteq)\) is a partial order
      \item Any \textbf{finite} non-empty subset \(S \subseteq L\) has a LUB and a GLB
    \end{enumerate}
\end{itemize}

\textbf{Example lattices:}

\textbf{Not lattices:}
Every finite set $A$ defines a lattice $(2A, \subseteq)$, where $\bot = \emptyset$, $\top = A$, $x \sqcup y = x \cup y$, and $x \sqcap y = x \cap y$. For example, consider the lattice corresponding to a poset of four elements:

![Diagram of a lattice](image)

- $L$ is natural numbers $\{0, 1, 2, 3, \ldots\}$
- $\subseteq$ is $\leq$

Every *finite* subset of $L$ has a LUB
Every subset of $L$ has a GLB
Therefore $(L, \leq)$ is a *lattice*
No infinite subset of $L$ has a LUB
Complete lattices

- A pair \((L, \sqsubseteq)\) is a complete lattice if:
  1. \((L, \sqsubseteq)\) is a partial order
  2. Any non-empty subset \(S \subseteq L\) has a LUB and a GLB

- Can identify and name two special elements:
  1. Bottom element: \(\bot = \text{GLB}(L)\)
  2. Top element: \(\top = \text{LUB}(L)\)

- All finite lattices are complete

Example””

- \(L\) is natural numbers \(\{0, 1, 2, 3, \ldots\}\)
- \(\sqsubseteq\) is \(\leq\)

Every finite subset of \(L\) has a GLB and LUB
Therefore \((L, \sqsubseteq)\) is a lattice
Every infinite subset of \(L\) has a LUB
Therefore \((L, \sqsubseteq)\) is a complete lattice
However, \(L\) has infinite ascending chains
Meet and Join

- By definition, for any lattice L, GLBs and LUBs are defined for finite sets

- Define operators meet (\(\cap\)) and join (\(\cup\)) as
  - \(x \cap y = \text{GLB}\{x, y\}\)
  - \(x \cup y = \text{LUB}\{x, y\}\)
  - For any finite set \(S \subseteq L\)
    - \(\cap S = \text{GLB}(S)\)
    - \(\cup S = \text{LUB}(S)\)
Example’’’’ lattice

- Consider \( S = \{a,b,c\} \) and its power set \( P = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a,b\}, \{b,c\}, \{a,c\}, \{a,b,c\}\} \)

- Define partial order as set inclusion: \( X \subseteq Y \)
  - Reflexive: \( X \subseteq X \)
  - Anti-symmetric: \( X \subseteq Y, Y \subseteq X \Rightarrow X = Y \)
  - Transitive: \( X \subseteq Y, Y \subseteq Z \Rightarrow X \subseteq Z \)

- Also, for any two elements of \( P \), there is a set that includes both and another set that is included in both

- Therefore \( (P, \subseteq) \) is a (complete) lattice

Power set lattice

- Partial order: \( \subseteq \)
  (set inclusion)

- Meet: \( \cap \)
  (set intersection)

- Join: \( \cup \)
  (set union)

- Top element: \( \{a,b,c\} \)
  (whole set)

- Bottom element: \( \emptyset \)
  (empty set)
Reversed lattice

- Partial order: \( \supseteq \)  
  (set inclusion)
- Meet: \( \sqcup \)  
  (set union)
- Join: \( \sqcap \)  
  (set intersection)
- Top element: \( \emptyset \)  
  (empty set)
- Bottom element: \{a,b,c\}  
  (whole set)

Relation to dataflow analysis

- Information computed by live variable analysis and available copies can be expressed as elements of lattices

- Live variables: if \( V \) is the set of all variables in the program and \( P \) the power set of \( V \), then:
  - \((P, \subseteq)\) is a lattice
  - sets of live variables are elements of this lattice
Relation to dataflow analysis

- **Copy Propagation:**
  - $V$ is the set of all variables in the program
  - $V \times V$ the Cartesian product representing all possible copy instructions
  - $P$ the power set of $V \times V$

- **Then:**
  - $(P, \subseteq)$ is a lattice
  - sets of available copies are lattice elements

Using lattices

- Assume information we want to compute in a program is expressed using a lattice $L$

- To compute the information at each program point we need to:
  - Determine how each instruction in the program changes the information
  - Determine how information changes at join/split points in the control flow
Transfer functions

- Dataflow analysis defines a transfer function \( F : L \rightarrow L \) for each instruction in the program
- Describes how the instruction modifies the information
- Consider \( \text{in}[I] \) is information before \( I \), and \( \text{out}[I] \) is information after \( I \)
  
  - Forward analysis: \( \text{out}[I] = F(\text{in}[I]) \)
  - Backward analysis: \( \text{in}[I] = F(\text{out}[I]) \)

Basic blocks

- Can extend the concept of transfer function to basic blocks using function composition
- Consider:
  - Basic block \( B \) consists of instructions \( (I_1, \ldots, I_n) \) with transfer functions \( F_1, \ldots, F_n \)
  - \( \text{in}[B] \) is information before \( B \)
  - \( \text{out}[B] \) is information after \( B \)

- Forward analysis:
  \[
  \text{out}[B] = F_n(\ldots(F_1(\text{in}[B]))) = F_n \circ \ldots \circ F_1(\text{in}[B])
  \]
- Backward analysis:
  \[
  \text{in}[I] = F_1(\ldots(F_n(\text{out}[i]))) = F_1 \circ \ldots \circ F_n(\text{out}[B])
  \]
Split/join points

- Dataflow analysis uses meet/join operations at split/join points in the control flow.

- Consider in[B] is lattice information at beginning of block B and out[B] is lattice information at end of B.

- Forward analysis: in[B] = \( \prod \{ \text{out}[B'] \mid B' \in \text{pred}(B) \} \)

- Backward analysis: out[B] = \( \sqcup \{ \text{in}[B'] \mid B' \in \text{succ}(B) \} \)

- Can alternatively use join operation \( \sqcup \) (equivalent to using the meet operation \( \prod \) in the reversed lattice).

Cartesian Products

- Let \( L_1, \ldots, L_n \) be sets.

- Cartesian product of \( L_1, \ldots, L_n \) is
  \[ \{ <x_1, \ldots, x_n> \mid x_i \in L_i \} \]

- If \( L_1, \ldots, L_n \) are (complete) lattices then their Cartesian product is a (complete) lattice, where \( \sqsubseteq \) is defined by
  \[ <x_1, \ldots, x_n> \sqsubseteq <y_1, \ldots, y_n> \text{ iff for all } i, x_i \sqsubseteq y_i \]
Information as Cartesian Product

- Consider a program analysis in which \( n \) program analysis variables range over lattice \( L \).
- We view the analysis as computing an \( n \)-tuple of \( L \)-values, i.e., a point in the \( n \)-ary Cartesian product of \( L \).
- Each change of one program analysis variable changes one component of the \( n \)-tuple.
- Analysis will terminate because we will only consider
  - Lattices with no infinite descending chains
  - “Monotonic” transfer functions that move us down (or not at all) in the lattice.

More about lattices

- In a lattice \((L, \sqsubseteq)\), the following are equivalent:
  1. \( x \sqsubseteq y \)
  2. \( x \sqcap y = x \)
  3. \( x \sqcup y = y \)

- Note: meet and join operations were defined using the partial order relation.
Proof (1 & 2)s

• Prove that \( x \sqsubseteq y \) implies \( x \land y = x \):
  – \( x \) is a lower bound of \( \{x, y\} \)
  – All lower bounds of \( \{x, y\} \) are less\( \leq \) than \( x, y \)
  – In particular, they are less\( \leq \) than \( x \)

• Prove that \( x \land y = x \) implies \( x \sqsubseteq y \):
  – \( x \) is a lower bound of \( \{x, y\} \)
  – \( x \) is less\( \leq \) than \( x \) and \( y \)
  – In particular, \( x \) is less\( \leq \) than \( y \)

• The meet and join operators are:
  1. Associative \( (x \land y) \land z = x \land (y \land z) \)
  2. Commutative \( x \land y = y \land x \)
  3. Idempotent: \( x \land x = x \)

• Property: If “\( \land \)” is an associative, commutative, and idempotent operator, then the relation “\( \sqsubseteq \)” defined as \( x \sqsubseteq y \) iff \( x \land y = x \) is a partial order

• Above property provides an alternative definition of a partial orders and lattices starting from the meet (join) operator
ROSE dataflow framework

- Live variable analysis:
  - Source code: `norris/soft/rose-git/src/midend/programAnalysis/genericDataflow/simpleAnalyses`
    - liveDeadVarAnalysis.h and liveDeadVarAnalysis.C
  - Binaries: `norris/soft/src/rose-build-ix-saucy/tests/roseTests/programAnalysisTests/variableLivenessTests`

Work through an example

```c
int foo(bool c, bool d, int x, int y, int z) {
    if (c) {
        x = y + 1;
        y = 2 * z;
        if (d) x = y + z;
        z = 1;
    }
    z = x;
    return z;
}
```

[https://bitbucket.org/brnorris03/pat-exercises](https://bitbucket.org/brnorris03/pat-exercises) git repository, week4/
[Annotated graph](https://ix.cs.uoregon.edu/~norris/pat/images/var-annotated.pdf) for this example:

Test in pat-exercises/week4:
```bash
make
cd simple2
../livenessTest simple2.cpp
```
Example with loop

```c
int foo(bool c, bool d, int x, int y, int z) {
    if (c) {
        x = y + 1;
        y = 2 * z;
        if (d) x = y + z;
        z = 1;
    }
    z = x;
    return z;
}
```

Test in pat-exercises/week4:
```
make
cd simple2
./livenessTest simple2.cpp
```