Data-Flow Analysis
Compilers

Source code

Lexing (P2)

Tokens

Parsing (P3)

Syntax tree

Analysis (P4)

Optimized Linear IR

Checked AST + Symtables

IR Code Gen (P5)

Optimization Passes

Linear IR

Machine code

Machine Code Gen
```c
int a;
a = 0;
while (a < 10) {
    a = a + 1;
}
```
Optimization is Hard

• **Problem**: it's hard to reason about all possible executions
  - Preconditions and inputs may differ
  - Optimizations should be correct and efficient in all cases
• Optimization tradeoff: investment vs. payoff
  - "Better than naïve" is fairly easy
  - "Optimal" is impossible
  - Real world: somewhere in between
    • Better speedups with more static analysis
    • Usually worth the added compile time
• Also: linear IRs (e.g., ILOC) don't explicitly expose control flow
  - This makes analysis and optimization difficult
Control-Flow Graphs

- **Basic blocks**
  - "Maximal-length sequence of branch-free code"
  - "Atomic" sequences (instructions that always execute together)

- **Control-flow graph (CFG)**
  - Nodes/vertices for basic blocks
  - Edges for control transfer
    - Branch/jump instructions (explicit) or fallthrough (implicit)
    - p is a predecessor of q if there is a path from p to q
      - p is an immediate predecessor if there is an edge directly from p to q
    - q is a successor of p if there is a path from p to q
      - q is an immediate successor if there is an edge directly from p to q
Control-Flow Graphs

- Conversion: linear IR to CFG
  - Find **leaders** (initial instruction of a basic block) and build blocks
    - Every call or jump target is a leader
  - Add edges between blocks based on jumps/branches and fallthrough
  - Complicated by indirect jumps (none in our ILOC!)

```
foo:
  loadAI [bp-4] => r1
  cbr r1 => l1, l2
l1:
  loadI 5 => r2
  jump l3
l2:
  loadI 10 => r2
l3:
  storeAI r2 => [bp-4]
```
Static CFG Analysis

- Single block analysis is easy, and trees are too
- General CFGs are harder
  - Which branch of a conditional will execute?
  - How many times will a loop execute?
- How do we handle this?
  - One method: iterative data-flow analysis
  - Simulate all possible paths through a region of code
  - “Meet-over-all-paths” conservative solution
  - Meet operator combines information across paths
Semilattices

• In general, a **semilattice** is a set of values $L$, special values $\top$ (top) and $\bot$ (bottom), and a **meet operator** $\wedge$ such that
  - $a \geq b$ iff $a \wedge b = b$
  - $a > b$ iff $a \geq b$ and $a \neq b$
  - $a \wedge \top = a$ for all $a \in L$
  - $a \wedge \bot = \bot$ for all $a \in L$

• Partial ordering
  - Monotonic

Figure 9.22 from Dragon book: semilattice of definitions using $\cup$ (set union) as the meet operation
Constant propagation

- For sparse simple constant propagation (SSCP), the lattice is very shallow
  - $c_i \land \top = c_i$ for all $c_i$
  - $c_i \land \bot = \bot$ for all $c_i$
  - $c_i \land c_j = c_i$ if $c_i = c_j$
  - $c_i \land c_j = \bot$ if $c_i \neq c_j$

- Basically: each SSA value is either unknown ($\top$), a known constant ($c_i$), or it is a variable ($\bot$)
  - Initialize to unknown ($\top$) for all SSA values
  - Interpret operations over lattice values (always lowering)
  - Propagate information until convergence
Constant propagation example

```
loadAI [bp-4] => r1
cbr r1 => l1, l2
loadI 5 => r2
jump l3
loadI 10 => r2
storeAI r2 => [bp-4]
```

Semilattice for Constant Propagation
Data-Flow Analysis

- Define **properties** of interest for basic blocks
  - Usually **sets** of blocks, variables, definitions, etc.
- Define a **formula** for how those properties change within a block
  - $F(B)$ is based on $F(A)$ where $A$ is a predecessor or successor of $B$
  - This is basically the *meet* operator for a particular problem
- Specify **initial information** for all blocks
  - Entry/exit blocks usually have special initial values
- Run an **iterative update** algorithm to propagate changes
  - Keep running until the properties converge for all basic blocks
- Key concept: **finite descending chain property**
  - Properties must be monotonically increasing or decreasing
  - Otherwise, termination is not guaranteed
Data-Flow Analysis

- This kind of algorithm is called **fixed-point iteration**
  - It runs until it converges to a “fixed point”

- **Forward vs. backward** data-flow analysis
  - Forward: along graph edges (based on predecessors)
  - Backward: reverse of forward (based on successors)

- Particular data-flow analyses:
  - Constant propagation
  - Dominance
  - Liveness
  - Available expressions
  - Reaching definitions
  - Anticipable expressions
Dominance

- Block A **dominates** block B if A is on every path from the entry to B
  - Block A **immediately** dominates block B if there are no blocks between them
  - Block B **postdominates** block A if B is on every path from A to an exit
  - Every block both dominates and postdominates itself

- **Simple dataflow analysis formulation**
  - \( \text{preds}(b) \) is the set of blocks that are predecessors of block b
  - \( \text{Dom}(b) \) is the set of blocks that dominate block b
    - (similar definition using \( \text{succs}(b) \))
  - \( \text{PostDom}(b) \) is the set of blocks that postdominate block b
    - (similar definition using \( \text{succs}(b) \))

*Initial conditions*: \( \text{Dom}(\text{entry}) = \{ \text{entry} \} \)

\[
\forall b \neq \text{entry}, \quad \text{Dom}(b) = \{ \text{all blocks} \}
\]

*Updates*: \( \text{Dom}(b) = \{ b \} \cup \bigcap_{p \in \text{preds}(b)} \text{Dom}(p) \)
Dominance example

Initial conditions:  \( \text{Dom}(\text{entry}) = \{\text{entry}\} \)
\[ \forall b \neq \text{entry}, \ \text{Dom}(b) = \{\text{all blocks}\} \]

Updates: \( \text{Dom}(b) = \{b\} \cup \bigcap_{p \in \text{preds}(b)} \text{Dom}(p) \)

\[
\begin{align*}
\text{Dom}(\text{foo}) &= \{\text{foo}\} \\
\text{Dom}(\text{l1}) &= \{\text{foo, l1}\} \\
\text{Dom}(\text{l2}) &= \{\text{foo, l2}\} \\
\text{Dom}(\text{l3}) &= \{\text{foo, l3}\}
\end{align*}
\]
Liveness

- Variable $v$ is live at point $p$ if there is a path from $p$ to a use of $v$ with no intervening assignment to $v$
  - Useful for finding uninitialized variables (live at function entry)
  - Useful for optimization (remove unused assignments)
  - Useful for register allocation (keep live vars in registers)

- Initial information: $UEVar$ and $VarKill$
  - $UEVar(B)$: variables used in $B$ before any redefinition in $B$
    - (“upwards exposed” variables)
  - $VarKill(B)$: variables that are defined (“killed”) in $B$

- Textbook notation note: $X \cap \overline{Y} = X - Y$

Initial conditions: $\forall b, \ LiveOut(b) = \emptyset$

Updates: $LiveOut(b) = \bigcup_{s \in \text{succs}(b)} UEVar(s) \cup (LiveOut(s) - VarKill(s))$
Liveness example

\[
\forall b, \quad \text{LiveOut}(b) = \emptyset \quad \text{LiveOut}(b) = \bigcup_{s \in \text{succs}(b)} \text{UEVar}(s) \cup (\text{LiveOut}(s) - \text{VarKill}(s))
\]

(a) Code for the Basic Blocks

\[
\begin{align*}
B_0: & \quad i \leftarrow 1 \quad \to B_1 \\
B_1: & \quad a \leftarrow \ldots \quad c \leftarrow \ldots \quad (a \leq c) \to B_2, B_5 \\
B_2: & \quad b \leftarrow \ldots \quad c \leftarrow \ldots \quad d \leftarrow \ldots \quad \to B_3 \\
B_3: & \quad y \leftarrow a + b \quad z \leftarrow c + d \quad i \leftarrow i + 1 \quad (i \leq 100) \to B_1, B_4 \\
B_4: & \quad \text{return} \\
B_5: & \quad a \leftarrow \ldots \quad d \leftarrow \ldots \quad (a \leq d) \to B_6, B_8 \\
B_6: & \quad d \leftarrow \ldots \quad \to B_7 \\
B_7: & \quad b \leftarrow \ldots \quad \to B_3 \\
B_8: & \quad c \leftarrow \ldots \quad \to B_7 
\end{align*}
\]

(b) Control-Flow Graph

(c) Initial Information

\[
\begin{array}{cccccccc}
\text{UEVAR} & B_0 & B_1 & B_2 & B_3 & B_4 & B_5 & B_6 & B_7 & B_8 \\
\emptyset & \emptyset & \emptyset & \{a,b,c,d,i\} & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\
\text{VARKILL} & \{i\} & \{a,c\} & \{b,c,d\} & \{y,z,i\} & \emptyset & \{a,d\} & \{d\} & \{b\} & \{c\}
\end{array}
\]
Alternative definition

- Define **LiveIn** as well as **LiveOut**
  - Two formulas for each basic block
  - Makes things a bit simpler to reason about
  - Separates change *within* block from change *between* blocks

\[ \forall b, \text{LiveIn}(b) = \emptyset, \text{LiveOut}(b) = \emptyset \]

\[ \text{LiveIn}(b) = \text{UEVar}(b) \cup (\text{LiveOut}(b) - \text{VarKill}(b)) \]

\[ \text{LiveOut}(b) = \bigcup_{s \in \text{succs}(b)} \text{LiveIn}(s) \]
Liveness example

\[ \forall b, \text{LiveIn}(b) = \emptyset, \text{LiveOut}(b) = \emptyset \]

\[ \text{LiveIn}(b) = \text{UEVar}(b) \cup (\text{LiveOut}(b) - \text{VarKill}(b)) \]

\[ \text{LiveOut}(b) = \bigcup_{s \in \text{succs}(b)} \text{LiveIn}(s) \]

LiveIn (foo) = {}
LiveOut(foo) = {}

LiveIn (l1) = {}
LiveOut(l1) = {r2}

LiveIn (l2) = {}
LiveOut(l2) = {r2}

LiveIn (l3) = {r2}
LiveOut(l3) = {}
Block orderings

- Forwards dataflow analyses converge faster with reverse postorder processing of CFG blocks
  - Visit as many of a block’s predecessors as possible before visiting that block
  - Strict reversal of normal postorder traversal
  - Similar to concept of topological sorting on DAGs
  - NOT EQUIVALENT to preorder traversal!
  - Backwards analyses should use reverse postorder on reverse CFG

Valid preorderings:
- A, B, D, C (left first)
- A, C, D, B (right first)

Valid postorderings:
- D, B, C, A (left first)
- D, C, B, A (right first)

Depth-first search:
- A, B, D, B, A, C, A (left first)
- A, C, D, C, A, B, A (right first)

Valid reverse postorderings:
- A, C, B, D
- A, B, C, D
\[\text{Dom}(\text{entry}) = \{\text{entry}\}\]
\[\forall b \neq \text{entry}, \; \text{Dom}(b) = \{\text{all blocks}\}\]
\[\text{Dom}(b) = \{b\} \cup \bigcap_{p \in \text{preds}(b)} \text{Dom}(p)\]

\[\forall b, \; \text{LiveOut}(b) = \emptyset\]
\[\text{LiveOut}(b) = \bigcup_{s \in \text{succs}(b)} \text{UEVar}(s) \cup (\text{LiveOut}(s) - \text{VarKill}(s))\]

\[\forall b, \; \text{LiveIn}(b) = \emptyset, \; \text{LiveOut}(b) = \emptyset\]
\[\text{LiveIn}(b) = \text{UEVar}(b) \cup (\text{LiveOut}(b) - \text{VarKill}(b))\]
\[\text{LiveOut}(b) = \bigcup_{s \in \text{succs}(b)} \text{LiveIn}(s)\]