Compilers

Intermediate Representations and Data-Flow Analysis

Yannis Smaragdakis, U. Athens
(original slides by Sam Guyer@Tufts)
Modern optimizing compiler

Front end:
1. Lexical analysis
2. Parsing
3. Semantic analysis

Middle end:
4. Analysis / Optimization

Back end:
5. Instruction selection
6. Register allocation
7. Instruction scheduling
A bit more detail

- Intermediate representations and code generation

**Diagram:**
- Scanner → Parser → Semantic checker
- Intermediate code generation
- Back end
- High-level IR
- Low-level IR
Low-level IR

- Linear stream of *abstract instructions*
- Instruction: single operation and assignment

\[
\begin{align*}
x &= y \text{ op } z \\
x &\leftarrow y \text{ op } z \\
\text{op } x, y, z
\end{align*}
\]

- Must break down high-level constructs
  - Example:
    \[
    \begin{align*}
z &= x - 2 \times y
\end{align*}
    \]
    \[
    \begin{align*}
t &= 2 \times y \\
z &= x - t
\end{align*}
    \]
  - Introduce temps as necessary: called *virtual registers*

- Simple control-flow
  - Label and goto

\[
\text{label1:} \quad \text{goto label1} \\
\text{if}_\text{goto } x, \text{ label1}
\]

*Jump to label1 if x has non-zero value*
Stack machines

- Originally for stack-based computers

- What are advantages?
  - Introduced names are *implicit*, not *explicit*
  - Simple to generate and execute code
  - Compact form – who cares about code size?
    - Embedded systems
    - Systems where code is transmitted (the ‘Net')
IR Trade-offs

for (i=0; i<N; i++)
    A[i] = i;

Loop invariant

Strength reduce to temp2 += 4

loop:
    temp1 = &A
    temp2 = i * 4
    temp3 = temp1 + temp2
    store [temp3] = i
    ...
goto loop
Towards code generation

```c
if (c == 0) {
    while (c < 20) {
        c = c + 2;
    }
} else {
    c = n * n + 2;
}
```

```
t1 = c == 0
if goto t1, lab1
t2 = n * n
c = t2 + 2
goto end
lab1:
t3 = c >= 20
if goto t3, end
c = c + 2
goto lab1
end:
```
Motivating Example: Dead code elimination

- **Idea:**
  - Remove a computation if result is never used

  \[
  y = w - 7; \\
  x = y + 1; \\
  y = 1; \\
  x = 2 * z;
  \]

  \[
  y = w - 7; \\
  y = 1; \\
  x = 2 * z;
  \]

- **Safety**
  - Variable is dead if it is never used after defined
  - Remove code that assigns to dead variables

- This may, in turn, create more dead code
  - Dead-code elimination usually works transitively
Dead code

- Another example:

  ```
  x = y + 1;
  y = 2 * z;
  x = y + z;
  z = 1;
  z = x;
  ```

- Which statements can be safely removed?

- Conditions:
  - Computations whose value is never used
  - Obvious for straight-line code
  - What about control flow?
Dead code

- With if-then-else:

  \[
  \begin{align*}
  x &= y + 1; \\
  y &= 2 * z; \\
  \text{if (c)} & \quad x = y + z; \\
  z &= 1; \\
  z &= x;
  \end{align*}
  \]

  \text{Which statements are can be removed?}

- Which statements are dead code?
  - What if “c” is false?
  - Dead only on some paths through the code
Dead code

- And a loop:

```java
while (p) {
    x = y + 1;
    y = 2 * z;
    if (c) x = y + z;
    z = 1;
}
```

*Which statements are can be removed?*

- Now which statements are dead code?
Dead code

- And a loop:

Which statements are can be removed?

```java
while (p) {
    x = y + 1;
    y = 2 * z;
    if (c) x = y + z;
    z = 1;
}
```

- Statement “x = y+1” not dead
- What about “z = 1”?
Low-level IR

- Most optimizations performed in low-level IR
  - Labels and jumps
  - No explicit loops

- Even harder to see possible paths

```
label1:
  jumpifnot p label2
  x = y + 1
  y = 2 * z
  jumpifnot c label3
  x = y + z
label3:
  z = 1
label2:
  jump label1
label1:
  z = x
```
Optimizations and control flow

- Dead code is *flow sensitive*
  - Not obvious from program
    - *Dead code example: are there any possible paths that make use of the value?*
  - Must characterize all possible dynamic behavior
  - Must verify conditions at compile-time

- Control flow makes it hard to extract information
  - High-level: different kinds of control structures
  - Low-level: control-flow hard to infer

- Need a unifying data structure
Control flow graph

- **Control flow graph** (CFG):
  
  *a graph representation of the program*
  
  - Includes both computation and control flow
  - Easy to check control flow properties
  - Provides a framework for global optimizations and other compiler passes

- Nodes are **basic blocks**
  
  - Consecutive sequences of non-branching statements

- Edges represent control flow
  
  - From jump to a label
  - Each block may have multiple incoming/outgoing edges
x = a + b;
y = 5;
if (c) {
    x = x + 1;
y = y + 1;
} else {
    x = x - 1;
y = y - 1;
} 
z = x + y;
Multiple program executions

- CFG models all program executions
- An actual execution is a path through the graph
- Multiple paths: multiple possible executions
  - How many?

```
x = a + b;
y = 5;
if (c)
```

```
x = x + 1;
y = y + 1;
```

```
x = x - 1;
y = y - 1;
```

```
z = x + y;
```
Execution 1

- CFG models all program executions

- Execution 1:
  - c is true
  - Program executes BB₁, BB₂, and BB₄

Control flow graph

```
Execution 1:

x = a + b;
y = 5;
if (c)

BB₁

BB₂

BB₃

BB₄

x = x + 1;
y = y + 1;

x = x - 1;
y = y - 1;

z = x + y;
```
Execution 2

- CFG models all program executions

Execution 2:
- c is false
- Program executes BB₁, BB₃, and BB₄

Control flow graph

BB₁
\[
x = a + b; \\
y = 5; \\
\text{if (c)}
\]

BB₂
\[
x = x + 1; \\
y = y + 1;
\]

BB₃
\[
x = x - 1; \\
y = y - 1;
\]

BB₄
\[
z = x + y;
\]
Basic blocks

- **Idea:**
  - Once execution enters the sequence, all statements (or instructions) are executed
  - Single-entry, single-exit region

- **Details**
  - Starts with a label
  - Ends with one or more branches
  - Edges may be labeled with predicates
    - *May include special categories of edges*
      - Exception jumps
      - Fall-through edges
      - Computed jumps (jump tables)
Building the CFG

- Two passes
  - First, group instructions into basic blocks
  - Second, analyze jumps and labels

- How to identify basic blocks?
  - Non-branching instructions
    
    *Control cannot flow out of a basic block without a jump*
  
  - Non-label instruction
    
    *Control cannot enter the middle of a block without a label*
Basic blocks

- Basic block starts:
  - At a label
  - After a jump

- Basic block ends:
  - At a jump
  - Before a label

```
label1:
jumpifnot p label2
x = y + 1
y = 2 * z
jumpifnot c label3
x = y + z
label3:
z = 1
jump label1
label2:
z = x
```
Basic blocks

- **Basic block starts:**
  - At a label
  - After a jump

- **Basic block ends:**
  - At a jump
  - Before a label

- **Note:** order still matters

---

```
label1:
  jumpifnot p label2
  x = y + 1
  y = 2 * z
  jumpifnot c label3

x = y + z

label3:
  z = 1
  jump label1

label2:
  z = x
```
Add edges

- Unconditional jump
  - Add edge from source of jump to the block containing the label

- Conditional jump
  - 2 successors
  - One may be the fall-through block

- Fall-through
Two CFGs

- From the high-level
  - Break down the complex constructs
  - Stop at sequences of non-control-flow statements
  - Requires special handling of break, continue, goto

- From the low-level
  - Start with lowered IR – tuples, or 3-address ops
  - Build up the graph
  - More general algorithm
  - Most compilers use this approach

- Should lead to roughly the same graph
Using the CFG

- Uniform representation for program behavior
  - Shows all possible program behavior
  - Each execution represented as a path
  - Can reason about potential behavior
    *Which paths can happen, which can’t*
  - Possible paths imply possible values of variables

- Example: *liveness* information

- Idea:
  - Define program points in CFG
  - Describe how information flows between points
Program points

- In between instructions
  - Before each instruction
  - After each instruction

May have multiple successors or predecessors
Live variables analysis

- **Idea**
  - Determine *live range* of a variable
    Region of the code between when the variable is assigned and when its value is used
  - Specifically:
    - **Def:** A variable \( v \) is live at point \( p \) if
      - There is a path through the CFG from \( p \) to a use of \( v \)
      - There are no assignments to \( v \) along the path
  - Compute a set of live variables at each point \( p \)

- **Uses of live variables:**
  - Dead-code elimination – find unused computations
  - Also: register allocation, garbage collection
Computing live variables

- How do we compute live variables?
  *(Specifically, a set of live variables at each program point)*

- What is a straight-forward algorithm?
  - Start at uses of v, search backward through the CFG
  - Add v to live variable set for each point visited
  - Stop when we hit assignment to v

- Can we do better?
  - Can we compute liveness for all variables at the same time?
    - **Idea:**
      - Maintain a set of live variables
      - Push set through the CFG, updating it at each instruction
Flow of information

- **Question 1**: how does information flow across instructions?

- **Question 2**: how does information flow between predecessor and successor blocks?
Live variables analysis

- At each program point:
  Which variables contain values computed earlier and needed later

- For instruction I:
  - \texttt{in}[I] : live variables at program point before I
  - \texttt{out}[I] : live variables at program point after I

- For a basic block B:
  - \texttt{in}[B] : live variables at beginning of B
  - \texttt{out}[B] : live variables at end of B

- Note: \texttt{in}[I] = \texttt{in}[B] for first instruction of B
  \texttt{out}[I] = \texttt{out}[B] for last instruction of B
Computing liveness

- **Answer question 1**: for each instruction I, what is relation between $\text{in}[I]$ and $\text{out}[I]$?

- **Answer question 2**: for each basic block $B$, with successors $B_1$, ..., $B_n$, what is relationship between $\text{out}[B]$ and $\text{in}[B_1]$ … $\text{in}[B_n]$
Part 1: Analyze instructions

- Live variables across instructions
- Examples:

  \[
  \text{in}[I] = \{y, z\} \\
x = y + z \\
\text{out}[I] = \{x\}
  \]

  \[
  \text{in}[I] = \{y, z, t\} \\
x = y + z \\
\text{out}[I] = \{x, t, y\}
  \]

  \[
  \text{in}[I] = \{x, t\} \\
x = x + 1 \\
\text{out}[I] = \{x, t\}
  \]

- Is there a general rule?
Liveness across instructions

How is liveness determined?

- All variables that I uses are live before I
  *Called the uses of I*

- All variables live after I are also live before I, unless I writes to them
  *Called the defs of I*

Mathematically:

\[
\text{in}[I] = \text{out}[I] - \text{def}[I] \cup \text{use}[I]
\]

\[
\begin{align*}
\text{in}[I] &= \{b\} \\
a &= b + 2 \\
\text{in}[I] &= \{y,z\} \\
x &= 5 \\
\text{out}[I] &= \{x,y,z\}
\end{align*}
\]
Example

- Single basic block
  (obviously: \( \text{out}[l] = \text{in}[\text{succ}(l)] \) )
  - Live1 = \( \text{in}[B] = \text{in}[l1] \)
  - Live2 = \( \text{out}[l1] = \text{in}[l2] \)
  - Live3 = \( \text{out}[l2] = \text{in}[l3] \)
  - Live4 = \( \text{out}[l3] = \text{out}[B] \)

- Relation between live sets
  - Live1 = \((\text{Live2} - \{x\}) \cup \{y\}\)
  - Live2 = \((\text{Live3} - \{y\}) \cup \{z\}\)
  - Live3 = \((\text{Live4} - \{}) \cup \{d\}\)
Flow of information

- Equation:
  \[ \text{in}[l] = ( \text{out}[l] - \text{def}[l] ) \cup \text{use}[l] \]

- Notice: information flows **backwards**
  - Need out[] sets to compute in[] sets
  - Propagate information up

- Many problems are **forward**
  Common sub-expressions, constant propagation, others
Part 2: Analyze control flow

- **Question 2**: for each basic block B, with successors $B_1$, ..., $B_n$, what is the relationship between $\text{out}[B]$ and $\text{in}[B_1] ... \text{in}[B_n]$?

- Example:

  ```
  B
  out={   }
  /
  /    /
B1    ...    Bn

  in={   }
  w=x+z;
  /         /
  /   ...   /

  in={   }
  q=x+y;
  ```

- What’s the general rule?
Control flow

- Rule: A variable is live at end of block B if it is live at the beginning of **any** of the successors
  - Characterizes all possible executions
  - *Conservative*: some paths may not actually happen

- Mathematically:
  \[
  \text{out}[B] = \bigcup_{B' \in \text{succ}(B)} \text{in}[B']
  \]

- Again: information flows backwards
System of equations

- Put parts together:
  \[
  \begin{align*}
  \text{in}[l] &= (\text{out}[l] - \text{def}[l]) \cup \text{use}[l] \\
  \text{out}[l] &= \text{in}[\text{succ}(l)] \\
  \text{out}[B] &= \bigcup_{B' \in \text{succ}(B)} \text{in}[B']
  \end{align*}
  \]

- Defines a system of equations (or constraints)
  - Consider equation instances for each instruction and each basic block
  - What happens with loops?
    - Circular dependences in the constraints
    - Is that a problem?
Solving the problem

- Iterative solution:
  - Start with empty sets of live variables
  - Iteratively apply constraints
  - Stop when we reach a fixpoint

For all instructions \( \text{in}[l] = \text{out}[l] = \emptyset \)

Repeat

For each instruction \( l \)

\[
\text{in}[l] = ( \text{out}[l] - \text{def}[l] ) \cup \text{use}[l]
\]

\[
\text{out}[l] = \text{in}[\text{succ}(l)]
\]

For each basic block \( B \)

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

Until no new changes in sets
Example

- **Steps:**
  - Set up live sets for each program point
  - Instantiate equations
  - Solve equations

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

- Program points

```plaintext
if (c)
  x = y+1
  y = 2*z
  if (d)
    L1
    L5
    L2
    L3
    L4
    L7
    L8
    L9
    L10
    L11
    L12

x = y+z
z = 1
z = x
```
Example

L1 = L2 ∪ {c}
L2 = L3 ∪ L11
L3 = (L4 – {x}) ∪ {y}
L4 = (L5 – {y}) ∪ {z}
L5 = L6 ∪ {d}
L6 = L7 ∪ L9
L7 = (L8 – {x}) ∪ {y,z}
L8 = L9
L9 = L10 – {z}
L10 = L1
L11 = (L12 – {z}) ∪ {x}
L12 = {}
Questions

- Does this terminate?
- Does this compute the right answer?
- How could generalize this scheme for other kinds of analysis?
Generalization

- **Dataflow analysis**
  - A common framework for such analysis
  - Computes information at each program point
  - Conservative: characterizes all possible program behaviors

- **Methodology**
  - Describe the information (e.g., live variable sets) using a structure called a **lattice**
  - Build a system of equations based on:
    - How each statement affects information
    - How information flows between basic blocks
  - Solve the system of constraints
Parts of live variables analysis

- Live variable sets
  - Called *flow values*
  - Associated with program points
  - Start “empty”, eventually contain solution

- Effects of instructions
  - Called *transfer functions*
  - Take a flow value, compute a new flow value that captures the effects
  - One for each instruction – often a schema

- Handling control flow
  - Called *confluence operator*
  - Combines flow values from different paths
Mathematical model

- Flow values
  - Elements of a lattice $L = (P, \subseteq)$
  - Flow value $v \in P$

- Transfer functions
  - Set of functions (one for each instruction)
  - $F_i : P \to P$

- Confluence operator
  - Merges lattice values
  - $C : P \times P \to P$

- How does this help us?
Lattices

- Lattice $L = (P, \subseteq)$
- A partial order relation $\subseteq$
  
  Reflexive, anti-symmetric, transitive
- Upper and lower bounds
  
  Consider a subset $S$ of $P$
  
  - Upper bound of $S$: $u \in S : \forall x \in S \ x \subseteq u$
  - Lower bound of $S$: $l \in S : \forall x \in S \ l \subseteq x$
- Lattices are complete
  
  Unique greatest and least elements
  
  - “Top” $T \in P : \forall x \in P \ x \subseteq T$
  - “Bottom” $\perp \in P : \forall x \in P \ \perp \subseteq x$
Confluence operator

- Combine flow values
  - “Merge” values on different control-flow paths
  - Result should be a safe over-approximation
  - We use the lattice $\subseteq$ to denote “more safe”

- Example: live variables
  - $v_1 = \{x, y, z\}$ and $v_2 = \{y, w\}$
  - How do we combine these values?
  - $v = v_1 \cup v_2 = \{w, x, y, z\}$
  - What is the “$\subseteq$” operator?
  - Superset
Meet and join

- **Goal:**
  
  *Combine two values to produce the “best” approximation*

  - **Intuition:**
    - Given $v_1 = \{x, y, z\}$ and $v_2 = \{y, w\}$
    - A safe over-approximation is “all variables live”
    - We want the smallest set

- **Greatest lower bound**
  
  - Given $x, y \in P$
  - $\text{GLB}(x, y) = z$ such that
    - $z \subseteq x$ and $z \subseteq y$ and
    - $\forall w \ w \subseteq x$ and $w \subseteq y \Rightarrow w \subseteq z$
  - **Meet** operator: $x \land y = \text{GLB}(x, y)$

- Natural “opposite”: Least upper bound, **join** operator
Termination

- **Monotonicity**
  Transfer functions $F$ are *monotonic* if
  - Given $x, y \in P$
  - If $x \subseteq y$ then $F(x) \subseteq F(y)$
  - Alternatively: $F(x) \subseteq x$

- **Key idea:**
  Iterative dataflow analysis terminates if
  - Transfer functions are monotonic
  - Lattice has finite height
  - *Intuition*: values only go down, can only go to bottom
Example

- Prove monotonicity of live variables analysis
  - Equation: \( \text{in}[i] = (\text{out}[i] - \text{def}[i]) \cup \text{use}[i] \)  
    
    \((\text{For each instruction } i)\)
  - As a function: \( F(x) = (x - \text{def}[i]) \cup \text{use}[i] \)
  - Obligation: If \( x \subseteq y \) then \( F(x) \subseteq F(y) \)
  - Prove:
    \[ x \subseteq y \implies (x - \text{def}[i]) \cup \text{use}[i] \subseteq (y - \text{def}[i]) \cup \text{use}[i] \]
    - Somewhat trivially:
      - \( x \subseteq y \Rightarrow x - s \subseteq y - s \)
      - \( x \subseteq y \Rightarrow x \cup s \subseteq y \cup s \)
Dataflow solution

- **Question:**
  - What is the solution we compute?
  - Start at lattice top, move down
  - Called greatest *fixpoint*
  - Where does approximation come from?
  - Confluence of control-flow paths

- **Ideal solution?**
  - Consider each path to a program point separately
  - Combine values at end
  - Called *meet-over-all-paths* solution (MOP)
  - When is the fixpoint equal to MOP?
Dataflow solution

- **Question:**
  - What is the solution we compute?
  - Start at lattice top, move down
  - Called greatest *fixpoint*
  - Where does approximation come from?
  - Confluence of control-flow paths

- **Knaster Tarski theorem**
  - Every monotonic function $F$ over a complete lattice $L$ has a unique least (and greatest) fixpoint
  - (Actually, the theorem is more general)
Composition of functions

Consider if-then-else graph

- If we compute each path:
  - \( \text{in} = F_4(F_2(F_1(\text{out}))) \)
  - \( \text{in} = F_4(F_3(F_1(\text{out}))) \)

- Two solutions
  
  **MOP:**
  - \( \text{in} = F_4(F_2(F_1(\text{out}))) \land F_4(F_3(F_1(\text{out}))) \)

  **Fixpoint:**
  - Merge live vars before applying \( F_4 \)
  - \( \text{in} = F_4( F_2(F_1(\text{out})) \land F_3(F_1(\text{out})) ) \)

- When are these two results the same?
  - When the transfer functions are **distributive**
  - Prove: \( F(x) \land F(y) = F(x \land y) \)
Summary

- Dataflow analysis
  - Lattice of flow values
  - Transfer functions (encode program behavior)
  - Iterative fixpoint computation

- **Key insight:**
  
  *If our dataflow equations have these properties:*
  - Transfer functions are monotonic
  - Lattice has finite height
  - Transfer functions distribute over meet operator
  
  *Then:*
  - Our fixpoint computation will terminate
  - Will compute meet-over-all-paths solution