

Final Review

Today

- Overview of what we have learned so far
- data-flow analysis review
- pointer and alias analysis
- interprocedural analysis
- register allocation
- loop skewing

Studying

- make sure to review terminology
 - (i.e. what does flow-sensitive mean?)

Big Picture: Traditional View of Compilers

Compiling down

- Translate high-level language to machine code

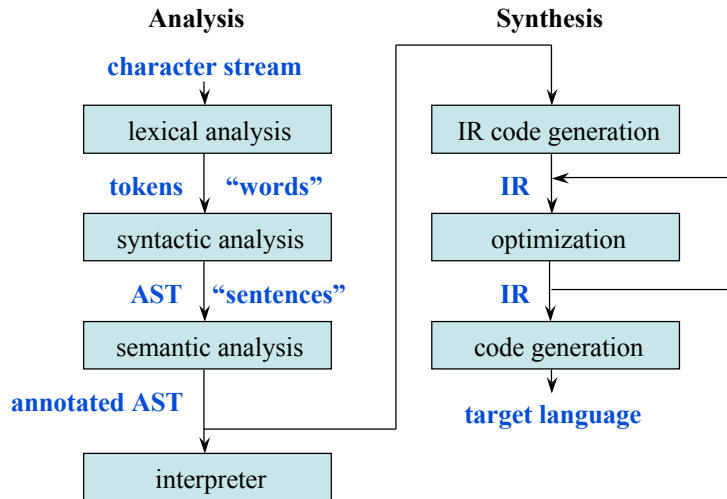
High-level programming languages

- Increase programmer productivity
- Improve program maintenance
- Improve portability

Low-level architectural details

- Instruction set
- Addressing modes
- Pipelines
- Registers, cache, and the rest of the memory hierarchy
- Instruction-level parallelism

Structure of a Typical Compiler



CS553 Lecture

Midterm Review

3

Topics

I. The Basics

- Scanning and parsing
- Dataflow analysis (review the projects, especially project 2)
 - Theoretic framework built on lattices
 - How might we improve the performance of iterative data-flow analysis by using a worklist-based algorithm?
- Control flow analysis: control-flow graphs, dominators, dominance frontiers, irreducibility

II. Analyses and Representations

- SSA Form: types of data dependencies, how to translate to minimal SSA
- Program optimizations
 - dead-code elimination, constant propagation (simple constants), CSE, loop-invariant code motion, PRE, copy propagation, induction variable elimination, strength reduction, global value numbering
- Aliases
 - how do data-flow analysis algorithms use aliasing information?
 - how do we characterize alias analysis algorithms?
- Interprocedural Analysis
 - how do different levels of context information affect analysis results?

CS553 Lecture

Midterm Review

4

Topics (cont)

III. Low-Level Optimizations

- Register allocation
 - difference between Briggs and Chaitin
 - heuristics to determine spilling will be provided
- Instruction scheduling
 - list scheduling
- Profile-guided and dynamic optimizations
 - what types of profiling information can we collect and how is it useful?
 - when are dynamic optimizations (or run-time reordering transformations) profitable?
- Predication and speculation
 - what HW features do these techniques attempt to deal with?

Topics (cont)

IV. High-Level Optimizations

- Dependence analysis
 - is this problem decidable? can we formulate it as decidable?
- Loop transformations
 - unimodular transformation framework
 - generating a schedule based on schedule constraints
- Tiling
 - what is it used for?
- Object-oriented optimizations
 - how does the compiler handle dynamic binding?

Topics (cont)

V. Emerging Topics

- Garbage collection
 - pros and cons between explicit and implicit garbage collection
 - reference counting, mark and sweep, generational collection, etc.
- Run-time reordering transformations
 - data reordering improves ?? locality
 - iteration reordering improves ?? locality
- Security and program checking
 - four different overall strategies

Data-flow Equations for Reaching Definitions

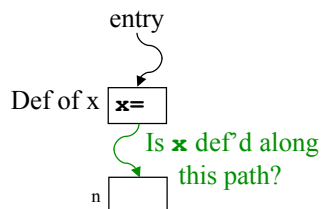
Symmetry between reaching definitions and liveness

- Swap $in[]$ and $out[]$ and swap the directions of the arcs

Reaching Definitions

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

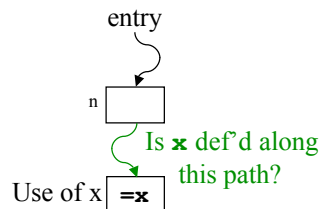
$$out[n] = gen[n] \cup (in[n] - kill[n])$$



Live Variables

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

$$in[n] = gen[n] \cup (out[n] - kill[n])$$



Reality Check!

Some definitions and uses are ambiguous

- We can't tell whether or what variable is involved
e.g., `*p = x;` */* what variable are we assigning?! */*
- Unambiguous assignments are called **strong updates**
- Ambiguous assignments are called **weak updates**

Solutions

- Be conservative
 - Sometimes we assume that it could be everything
e.g., Defining `*p` (generating reaching definitions)
 - Sometimes we assume that it is nothing
e.g., Defining `*p` (killing reaching definitions)
- Try to figure it out: alias/pointer analysis (more later)

Using Alias Information

Example: reaching definitions

- Compute at each point in the program a set of (s, v) pairs, indicating that statement s may define variable v

Flow functions

- $s: *p = x;$
$$\text{out}_{\text{reach}}[s] = \{(s, z) \mid (p \rightarrow z) \in \text{in}_{\text{may-pt}}[s]\} \cup$$
$$(\text{in}_{\text{reach}}[s] - \{(t, y) \mid (p \rightarrow y) \in \text{in}_{\text{must-pt}}[s]\})$$
- $s: x = *p;$
$$\text{out}_{\text{reach}}[s] = \{(s, x)\} \cup (\text{in}_{\text{reach}}[s] - \{(t, x) \mid \forall t\})$$
- ...

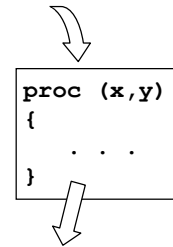
Interprocedural Analysis: Two Types of Information

Track information that flows into a procedure

- Sometimes known as **propagation problems**
e.g., What formals are constant?
e.g., Which formals are aliased to globals?

Track information that flows out of a procedure

- Sometimes known as **side effect problems**
e.g., Which globals are def'd/used by a procedure?
e.g., Which locals are def'd/used by a procedure?
e.g., Which actual parameters are def'd by a procedure?



Examples

Propagation Summaries

- MAY-ALIAS: The set of formals that may be aliased to globals and each other
- MUST-ALIAS: The set of formals that are definitely aliased to globals and each other
- CONSTANT: The set of formals that must be constant

Side-effect Summaries

- MOD: The set of variables possibly modified (defined) by a call to a procedure
- REF: The set of variables possibly read (used) by a call to a procedure
- KILL: The set of variables that are definitely killed by a procedure (*e.g.*, in the liveness sense)

Computing Interprocedural Summaries

Top-down

- Summarize information about the caller (MAY-ALIAS, MUST-ALIAS)
- Use this information inside the procedure body

```
int a;
void foo(int &b, &c){
    . . .
}
foo(a,a);
```

Bottom-up

- Summarize the effects of a call (MOD, REF, KILL)
- Use this information around procedure calls

```
x = 7;
foo(x);
y = x + 3;
```

Side-Effect Analysis

```
main() {
    int *a,*b,c,d;
    a = &c;
    b = &d;
    foo(&a,&a);
    foo(&b,&a);
}
void foo(int** x, int **y){
    *x = *y;
    **x = 3;
}
```

Register Allocation: Spilling

If we can't find a k-coloring of the interference graph

- Spill variables (nodes) until the graph is colorable

Choosing variables to spill

- Choose least frequently accessed variables
- Break ties by choosing nodes with the most conflicts in the interference graph
- Yes, these are heuristics!

Example

Sample code

```
do i = 1, 6
  do j = 1, 5
    A(2i, j) = A(i, j-1)
  enddo
enddo
```

Dependence

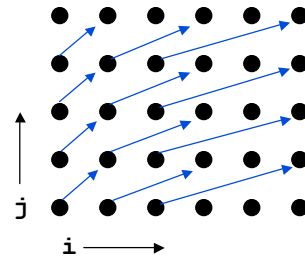
- $2i_1 - i_2 = 0, j_1 = j_2 - 1$, solution: YES

Distance/Direction Vector

- $(i_1, j_1) + (d_i, d_j) = (i_2, j_2), d_j = 1, d_i = ?, d = (<, 1)$

Dependence Relation

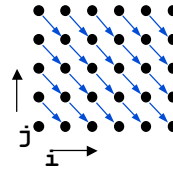
- $\{ [i, j] \rightarrow [2i, j + 1] \mid 1 \leq i \leq 3 \ \&\& \ 1 \leq j \leq 4 \}$



Loop Transformations

Original code

```
do i = 1,6
  do j = 1,5
    A(i,j) = A(i-1,j+1)+1
  enddo
enddo
```



Distance vector: (1, -1)

Which loop can we parallelize?

Scheduling a SARE

```
for (i = 1..n, j = 1..n), A(i,j) = A(i-1,j+1)+1
```

Linear schedule is of the form

$$t(i,j) = a*i + b*j + c$$

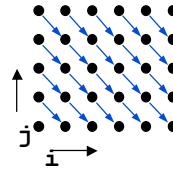
Finding constraints on the schedule is similar to checking transformation legality in omega

$$t(i,j) \leq t(i+1, j-1)$$

Loop Skewing

Original code

```
do i = 1,6
  do j = 1,5
    A(i,j) = A(i-1,j+1)+1
  enddo
enddo
```

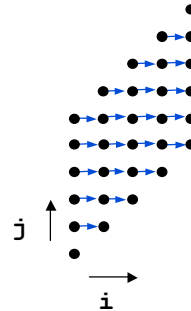


Distance vector: (1, -1)

Can we permute the original loop?

Skewing:

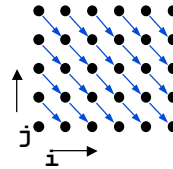
$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i \\ i+j \end{bmatrix}$$



Transforming the Dependences and Array Accesses

Original code

```
do i = 1,6
  do j = 1,5
    A(i,j) = A(i-1,j+1)+1
  enddo
enddo
```



Dependence vector:

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

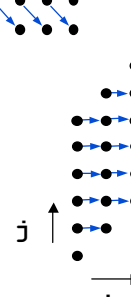
New Array Accesses:

$$A\left(\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right) = A(i, j)$$

$$A\left(\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right) = A(i', j' - i')$$

$$A\left(\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right) = A(i-1, j+1)$$

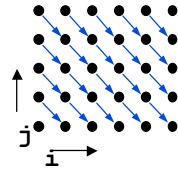
$$A\left(\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right) = A(i'-1, j'-i'+1)$$



Transforming the Loop Bounds

Original code

```
do i = 1, 6
  do j = 1, 5
    A(i, j) = A(i-1, j+1) + 1
  enddo
enddo
```



Bounds:

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix} -1 \\ 6 \\ -1 \\ 5 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i' \\ j' - i' \end{bmatrix} \leq \begin{bmatrix} -1 \\ 6 \\ -1 \\ 5 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} \leq \begin{bmatrix} -1 \\ 6 \\ -1 \\ 5 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} \leq \begin{bmatrix} -1 - i' \\ 6 + i' \\ -1 - i' \\ 5 + i' \end{bmatrix}$$

Transformed code

```
do i' = 1, 6
  do j' = 1 + i', 5 + i'
    A(i', j' - i') = A(i' - 1, j' - i' + 1) + 1
  enddo
enddo
```

