

Program Optimizations using Data-Flow Analysis

Last time

- Lattice theoretic framework for data-flow analysis

Today

- Dead-code elimination
- Copy propagation
- Constant propagation
- Common sub-expression elimination (CSE)

Dead Code Elimination

Remove statements that define only one variable and the variable being defined is not in the live out set.

Algorithm

```
do {  
  1) generate a FlowGraph from the list of instructions  
  2) perform liveness  
  3) for each node in FlowGraph  
     if the defs set contains only one temporary  
       if the temporary being defined is not in the live out set  
         remove the node from the FlowGraph  
  4) generate a list of instructions from the modified FlowGraph  
  
} while (changes);
```

Dead code elimination with the MiniJava compiler

- 1) Method in FlowGraph for removing a node that attaches predecessors with successors.
- 2) Modify Targets class so that we can query getFallThrough() and get a label for constructing a new list of Assem.Instr.
- 3) Method to trace through a FlowGraph to generate an Assem.Instr list.
- 4) DataFlowSet interface (will be provided)
- 5) DataFlowProblem interface (will be provided)
- 6) DataFlowSolver class where the constructor takes a DataFlowProblem as input and solves it with IDFA.
- 7) Reimplement Liveness using the data-flow framework (an implementation for LiveDataFlowSet will be provided)

Copy Propagation

Propagate the results of a move statement.

Algorithm

```
do {  
  1) generate a FlowGraph from the list of instructions  
  2) perform reaching definitions  
  3) for each node in FlowGraph  
     for each use  
       if the use is reached by a def in a move statement  
         change the use to the rhs in the move statement  
  4) generate a list of instructions from the modified FlowGraph  
  
} while (changes);
```

Copy propagation with the MiniJava compiler

Assume the DataFlowSolver has already been written.

- The ReachDefs class should implement the DataFlowProblem interface.
- Implement a ReachDefsDataFlowSet class with reaching def statements paired with the temporary they define.
- Implement a CopyProp class.

Constant Propagation using Reaching Definitions

Propagate the results of a CONSTANT move statement.

Algorithm

```
do {  
  1) generate a FlowGraph from the list of instructions  
  2) perform reaching definitions  
  3) for each node in FlowGraph  
     for each use  
       if the use is reached by a def in a constant move statement  
         change the use to the rhs in the move statement  
  4) generate a list of instructions from the modified FlowGraph  
} while (changes);
```

Constant Propagation with the MiniJava compiler

Assume the DataFlowSolver has already been written.

- Subclass Assem.Instr with a CONSTMOVE instruction type.
- The ReachDefs class should implement the DataFlowProblem interface.
- Implement a ReachDefsDataFlowSet class with reaching def statements paired with the temporary they define.
- Implement a ConstPropReachDef class.

Constant Propagation

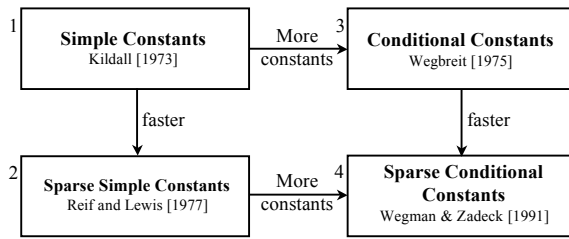
Goal

- Discover constant variables and expressions and propagate them forward through the program

Uses

- Evaluate expressions at compile time instead of run time
- Eliminate dead code (*e.g.*, debugging code)
- Improve efficacy of other optimizations (*e.g.*, value numbering and software pipelining)

Roadmap



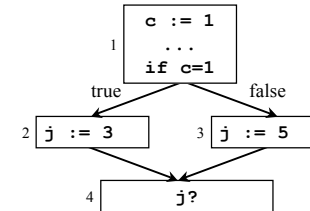
Kinds of Constants

Simple constants Kildall [1973]

- Constant for all paths through a program

Conditional constants Wegbreit [1975]

- Constant for actual paths through a program (when only one direction of a conditional is taken)



Data-Flow Analysis for Simple Constant Propagation

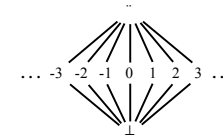
Simple constant propagation: analysis is “reaching constants”

- D: $2^{V \times C}$
- \sqcap : \cap
- F:
 - Kill($x \leftarrow \dots$) = $\{(x, c) \forall c\}$
 - Gen($x \leftarrow c$) = $\{(x, c)\}$
 - Gen($x \leftarrow y \oplus z$) = $\text{if } (y, c_y) \in \text{In} \ \& \ (z, c_z) \in \text{In}, \{(x, c_y \oplus c_z)\}$
 - ...

Data-Flow Analysis for Simple Constant Propagation (cont)

Reaching constants for simple constant propagation

- D: $\{\text{All constants}\} \cup \{\top, \perp\}$ **Using tuples of lattices**
- \sqcap : $c \sqcap \top = c$
 $c \sqcap \perp = \perp$
 $c \sqcap d = \perp$ if $c \neq d$
 $c \sqcap d = c$ if $c = d$
- F:
 - $F_{x \leftarrow c}(\text{In}) = c$
 - $F_{x \leftarrow y \oplus z}(\text{In}) = \text{if } c_y = \text{In}_y \ \& \ c_z = \text{In}_z, \text{ then } c_y \oplus c_z, \text{ else } \top \text{ or } \perp$
 - ...



Initialization for Reaching Constants

Pessimistic

- Each variable is initially set to \perp in data-flow analysis
- Forces merges at loop headers to go to \perp conservatively

Optimistic

- Each variable is initially set to \top in data-flow analysis
- Each variable is set to \perp at the entry node in the flow graph.
- What assumption is being made when optimistic reaching constants is performed?

Simple Constants with the MiniJava compiler

Assume the DataFlowSolver has already been written.

- Subclass Assem.Instr with a CONSTMOVE instruction type.
- Subclass Assem.Instr with a BINOPMOVE instruction type that can be queried for the operator type, each operand, and the temporary being defined.
- The SimpleConstants class should implement the DataFlowProblem interface.
 - In the transfer function, BINOPMOVE instructions may be replaced with CONSTMOVE instructions.
 - Can we replace operands to a BINOPMOVE instruction with a constant?
- The SimpleConstants class should take a reference to a list of Assem.Instrs as input and as a side-effect modify that list.

Common Subexpression Elimination

Idea

- Find common subexpressions whose *range* spans the same basic blocks and eliminate unnecessary re-evaluations
- Leverage available expressions

Recall available expressions

- An expression (e.g., $x+y$) is **available** at node n if **every** path from the entry node to n evaluates $x+y$, and there are no definitions of x or y after the last evaluation along that path

Strategy

- If an expression is available at a point where it is evaluated, it need not be recomputed

CSE Example

Before CSE

```
c := a + b
d := m & n
e := b + d
f := a + b
g := -b
h := b + a
a := j + a
k := m & n
j := b + d
a := -b
if m & n goto L2
```



After CSE

```
t1 := a + b
c := t1
t2 := m & n
d := t2
t3 := b + d
e := t3
f := t1
g := -b
h := t1
a := j + a
k := t2
j := t3
a := -b
if t2 goto L2
```

Summary

11 instructions
12 variables
9 binary operators

Summary

14 instructions
15 variables
4 binary operators

CSE Approach 1

Notation

- Avail(b) is the set of expressions available at block b
- Gen(b) is the set of expressions generated and not killed at block b

If we use e and $e \in \text{Avail}(b)$

- Allocate a new name n
- Search backward from b (in CFG) to find statements (one for each path) that most recently generate e
- Insert copy to n after generators
- Replace e with n

Example

```
a := b + c
t1 := a
t2 := a
e := b1 + c
f := b2 + c
```

Problems

- Backward search for each use is expensive
- Generates unique name for each use
 - $|\text{Uses}| > |\text{Avail}|$
 - Each generator may have many copies

CSE Approach 2

Idea

- Reduce number of copies by assigning a unique name to each unique expression

Summary

- $\forall e \text{ Name}[e] = \text{unassigned}$
- if we use e and $e \in \text{Avail}(b)$
 - if $\text{Name}[e] = \text{unassigned}$, allocate new name n and $\text{Name}[e] = n$
 - else $n = \text{Name}[e]$
 - Replace e with n
- In a subsequent traversal of block b, if $e \in \text{Gen}(b)$ and $\text{Name}[e] \neq \text{unassigned}$, then insert a copy to $\text{Name}[e]$ after the generator of e

Problem

- May still insert unnecessary copies
- Requires two passes over the code

Example

```
a := b + c
t1 := a
```

CSE Approach 3

Idea

- Don't worry about temporaries
- Create one temporary for each unique expression
- Let subsequent pass eliminate unnecessary temporaries

At an evaluation of e

- Hash e to a name, n, in a table
- Insert an assignment of e to n

At a use of e in b, if $e \in \text{Avail}(b)$

- Lookup e's name in the hash table (call this name n)
- Replace e with n

Problems

- Inserts more copies than approach 2 (but extra copies are dead)
- Still requires two passes (2nd pass is very general)

Extraneous Copies

Extraneous copies degrade performance

Let other transformations deal with them

- Dead code elimination
- Coalescing

Coalesce assignments to t1 and t2 into a single statement

```
t1 := b + c
t2 := t1
```

- Greatly simplifies CSE

CSE Approach 4

Idea

- Create one temporary for each unique expression
- Calculate reaching expressions at the same time as available expressions

At an evaluation of e

- Hash e to a name, n, in a table

At a use of e in b, if $e \in \text{Avail}(b)$

- Lookup e's name in the hash table (call this name n)
- Replace e with n
- Insert an assignment of e to n after each reaching expression

Problems?

- Only requires modification of available expressions analysis to include reaching definitions

CSE with the MiniJava compiler

Assume the DataFlowSolver has already been written.

- Subclass Assem.Instr with a CONSTMOVE instruction type.
- Subclass Assem.Instr with a BINOPMOVE instruction type that can be queried for the operator type, each operand, and the temporary being defined.
- The AvailExpr class should implement the DataFlowProblem interface.
- The CSE class should take a reference to a list of Assem.Instrs as input and as a side-effect modify that list.

Next Time

Reading

- Ch 17.4 and Ch 18 intro

Lecture

- Speeding up data-flow analysis