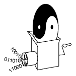


CS 412
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 32: Finishing optimization
19 Apr '01

Administrivia

- Prelim 2 graded
- $\mu = 65$, $\sigma = 12$

Aliasing

- Problem: don't know when two memory operands might refer to same location (*alias* one another)
- Needed everywhere:
  - IR generation (MEM/MOVE commute?)
  - CSE optimization (is node [x] available?)
  - Instruction scheduling (accurate dependence info)
- What information do we need?
- How can we compute it?

Aliasing information

- Must vs. may alias
  "p may alias q"
  "p must alias q"
- Flow-sensitive vs. insensitive analysis:
  - Flow-insensitive: "*x* may alias *y*"
  - Flow-sensitive : "*x* may alias *y* at program point *p*" (flowgraph edge *p*)
- Pointer vs. shape analysis:
  - Pointer: *p* may alias *q*
  - Shape: *p.x* may alias *q.y*
  - Array index bounds analysis: *p* may alias *a*[*i*]

CSE on memory locations

- Previously computed value can be reused if *available expression*
- Memory is slow ⇒ want to avoid fetches

n	*kill*[*n*]
a=b + c	*uses*(a)
a=[b]	*uses*(a)
[a]=b	*uses*([*x*]) (for all *x* that may alias a)
a=f(b_1,…b_n)	*uses*([*x*]) for all *x* (except stack offsets?)

Loop invariants

- Loop invariants allow:
  - induction variable optimizations
  - loop-invariant code motion
- Expression [*x*] is loop-invariant only if no writes [*y*] = a for *y* aliasing *x*

Conclusion:
- *may alias* information more important than *must alias*
- Shape analysis usually overkill

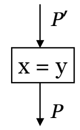
Heuristics

- Flow-sensitive may-alias pointer analysis: for each program point, variable *x*, determine which things [*x*] might alias
- Use high-level knowledge
  - objects of unrelated types cannot be aliases
  - stack and heap locations cannot be aliases (no aliasing of spilled temps!)
- Do early in optimization:
  - high-level language information available
  - many optimizations can rely on alias info
  - propagate needed information down

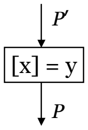
Abstraction

- Can't analyze full run-time behavior of program. Idea: create abstraction of memory
- Variable *x* points to a set of possible locations $ptr_P(x)$ at each program point *P*:
  - null: the null pointer
  - loc(v): static or stack storage for variable v
  - heap(ty): heap-allocated storage w/ type ty (abstracts all heap locations of type ty)
  - globals(ty): data-segment storage w/ type ty
  - anon(ty): any storage with type ty (abstracts all stack, heap, global locations of ty)
- Vars x_1 and x_2 cannot be aliases if sets do not contain possibly overlapping elements

Flow functions

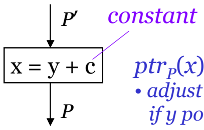


$ptr_P(x) \supseteq ptr_P(y) \supseteq ptr_{P'}(y)$



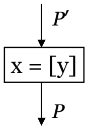
$ptr_P(x) = ptr_{P'}(x)$

Flow functions



$ptr_P(x) \supseteq ptr_P(y)$

- *adjust* $ty_P(x)$ *to tail of structure if y points to structure*

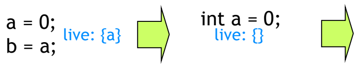


$ptr_P(x) = anon(ty_P(y))$

Shape analysis:
$ty_P(y)$ has points-to information

Interleaving optimizations

- Problem: optimizations invalidate analysis!
- Examples: live variable analysis, dead code elimination



```
a = 0;
b = a;
```

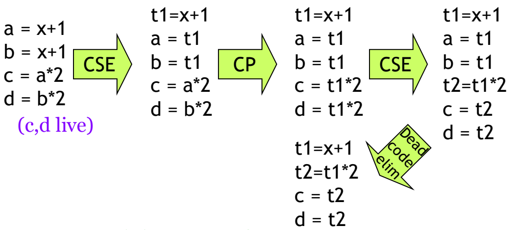
live: {a}

```
int a = 0;
```

live: {}

Another example

- Available expressions, reaching definitions, available expressions, live variable analysis



```
a = x+1
b = x+1
c = a*2
d = b*2
```

(c,d live)

```
t1=x+1
a = t1
b = t1
c = a*2
d = b*2
```

```
t1=x+1
a = t1
b = t1
c = t1*2
d = t1*2
```

```
t1=x+1
a = t1
b = t1
t2=t1*2
c = t2
d = t2
```

```
t1=x+1
t2=t1*2
c = t2
d = t2
```

Avoiding recomputation

- How to avoid redoing every dataflow analysis from scratch at every change?
- Option 1: design a *cascading analysis* that identifies all optimization possibilities in one pass
  - constant folding + simple constant propagation = “classic” constant propagation

Dead code elimination

- Cascading live variable analysis + removal of dead variables: dead code elimination
- Variable is dead if it does not contain an *essential* value
- x essential before:
  $[x] = y, y = f(x)$ (always)
  $y = x, y = x + z, y = [x]$ (where y essential)
  Another dataflow analysis...
  - *Better* than repeated analysis: i=i+1

Incremental data-flow analysis

- Existing analysis is usually mostly right
  - some values may be too low in lattice
  - set values possibly affected by change to $\top$, restart iteration until convergence
  - sometimes can do better than $\top$
- Example: deleting statement a = b OP c
  - b, c may not be live on entry
  - algorithm: delete b,c from all live sets, add updated nodes to worklist, restart analysis

Modularity Conflict

- Localized code inspection & transformation. How to implement?
- Two orthogonal organizing principles: node types and phases (rows or columns)

phases

types

	typeNames	typeCheck	foldConst
Add	×	×	×
Num	×	×	×
Id	×	×	×
Stmt	×	×	×

Objects vs. Operations

- Modularity by objects (rows): different methods share basic traversal code -- boilerplate code
- Modularity by operations (columns): lots of copied boilerplate:

```
Node foldConstants(Node n) {
  if (n instanceof Add)  { Add a = (Add) n; ...}
  else if (n instanceof Id)  { Id x = (Id) n; ... }
  else ...
}
```

Visitors

- Idea: avoid repetition by providing one set of standard traversal code
- Knowledge of particular phase embedded in *visitor* object
- Standard traversal code is done by object methods, reused by every phase
- Visitor invoked at every step of traversal to allow it to do phase-specific work

"Classic" Visitors

```
abstract class Visitor {
   void binOp(BinOp b) {}
   void unOp(UnOp u) {}
   …
   void doIf(IfStmt i) {}
   void doWhile(WhileStmt w) {}
}

abstract class Node { void visit(Visitor v); }

class UnOp extends Expr {
   Expr operand;
   void visit(Visitor v) { operand.visit(v); v.doUnOp(this); }
}
```

One method per node type

JLtools Visitor Methodology

- Class Node is superclass for all AST nodes
- NodeVisitor is superclass for all visitor classes (one visitor class per phase)

```
abstract class Node {
   public final Node visit (NodeVisitor v) {
      Node n = v.override (this);        // default: null
      if (n != null) return n;
      else {
         NodeVisitor v_ = v.enter(this);  // default: v_=v
         n = visitChildren (v_);          // visit children
         return v.leave(this, n, v_);     // default: n
      }}
   abstract Node visitChildren(NodeVisitor v);
      // apply visitor to all children and rebuild node
```

Folding constants with visitors

```
public class ConstantFolder extends NodeVisitor {
  public Node leave (Node old, Node n, NodeVisitor v) {
    return n.foldConstants();
        // note: all children of n already folded
  }}
class Node { Node foldConstants( ) { return this; } }
class BinaryExpression { int op; Expr lhs, rhs;
   Node foldConstants( ) { switch(op) {
        case PLUS: if (lhs instanceof Constant && …)
          return new Constant(lhsc.val + rhsc.val);
        else return this; case MINUS: …}}
   Node visitChildren(Visitor v) {
        Expr newlhs = lhs.visit(v), newrhs = rhs.visit(v);
        if (newlhs == lhs && newrhs = rhs) return this;
        return new BinaryExpression(newlhs, newrhs);     }}}
```

Lazy reconstruction

Visitors

- Generic, efficient AST traversal & reconstruction code shared across all visitors
- Compiler work easily partitioned into small, cleanly separated passes
- Functional programming style works
- Applicable to tree-structured IR

What's next?

- The stuff you'll never hear anywhere else
  - How linking and loading *really* work
  - How memory management *really* works
- More language features
  - First-class functions
  - Exceptions
  - Parametric polymorphism
  - Dynamic (run-time) types and meta-object protocols