Alias Analysis &
Redundant Computation
Elimination Optimizations

CS2210
Lecture 19

CS2210 Compiler Design 2004/5

Alias Analysis
- Determine whether a storage location
  may be accessed in more than one way
- Example in C:
  p = & x; x and *p are aliases for the
  same location
- Other sources of aliasing:
  - ?

Memory Disambiguation
- Want to disambiguate memory
  accesses for optimization:

```c
exam1() {  
    int a, k;
    extern int* q;
    /*
    k = a+5;  
    f(a,k1);  
    *q = 13;  
    k = a+5; /* is this redundant? */
    */
    ...
}  
```

CS2210 Compiler Design 2004/5
## Alias Information Classification

<table>
<thead>
<tr>
<th></th>
<th>May Alasing</th>
<th>Must alasing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flow-sensitive:</strong></td>
<td>If (d)</td>
<td>If (d)</td>
</tr>
<tr>
<td></td>
<td>p = &amp; x; {(x,*p)}</td>
<td>p = &amp; x; {(x,*p)}</td>
</tr>
<tr>
<td></td>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>p = &amp; y; {(y,*p)}</td>
<td>p = &amp; y; {(y,*p)}</td>
</tr>
<tr>
<td></td>
<td>L: {(x,*p),(y,*p)}</td>
<td>L: {}</td>
</tr>
<tr>
<td><strong>Flow-insensitive:</strong></td>
<td>If (d)</td>
<td>const int* p = &amp;x;</td>
</tr>
<tr>
<td></td>
<td>p = &amp; x; {(x,*p),(y,*p)}</td>
<td>((x,*p))</td>
</tr>
<tr>
<td></td>
<td>else</td>
<td>no other assignments</td>
</tr>
<tr>
<td></td>
<td>p = &amp; y; {(x,*p),(y,*p)}</td>
<td>to p throughout program</td>
</tr>
<tr>
<td></td>
<td>L: {(x,*p),(y,*p)}</td>
<td>L: {}</td>
</tr>
</tbody>
</table>

## Steensgaard’s Algorithm

- A flow-insensitive algorithm for C
- \(O(n \alpha(n,n))\) complexity
- Near linear, \(n\) number of statements
- Computes **points-to sets**
- \(P(t(p) = \{x,y\}) p\) may point to variables \(x\) and \(y\)
- May information (flow insensitive)

## Alias-relevant Statements

- \(x = y\)
- \(x = &y\)
- \(x = *y\)
- \(x = \text{op}(y_1, ..., y_n)\)
- \(x = \text{allocate}(y)\)
- \(*x = y\)
- \(x = \text{fun}(f_1, ..., f_n) \rightarrow (r_1, ..., r_m)\)
- \(x_1, ..., x_n = p(y_1, ..., p_n)\)
Representation of Points-to Sets

\[ \alpha :: \tau \times \lambda \]
\[ \tau \text{ represents memory locations} \]
\[ \lambda \text{ represents procedures} \]

\[ \tau :: \bot \mid \text{ref}(\alpha) \]
\[ \lambda :: \bot \mid \text{lam}(\alpha_1 ... \alpha_n)(\alpha_{n+1} ... \alpha_{n+m}) \]

"Type-Rules"

- Analysis is expressed in "type rules"
- \[ A |- x : \text{ref}(\alpha_1) \]
- \[ A |- y : \text{ref}(\alpha_2) \]
  \[ \alpha_2 \leq \alpha_1 \]

\[ A |- \text{welltyped}(x=y) \]

More Rules

- \[ A |- x : \text{ref}(\tau \times \_ \_ \_) \]
  \[ A |- y : \tau \]

\[ A |- \text{welltyped}(x=y) \]

- \[ A |- x : \text{ref}(\text{ref}(\alpha_1) \times \_ \_ \_) \]
  \[ A |- y : \text{ref}(\alpha_2) \]
  \[ \alpha_2 \leq \alpha_1 \]

\[ A |- \text{welltyped}(\_ \_ x=y) \]
Example

\[
\begin{align*}
& a = &x \\
& b = &y \\
& \text{if } p \text{ then} \\
& \quad y = &z; \\
& \text{else} \\
& \quad y = &x; \\
& \text{fi} \\
& X = &y
\end{align*}
\]

Location Sets

- Use equivalence classes to represent “types”
  - Fast union / join using union-join data structure
  - \( ecr(x) \) is equivalence class for \( x \)
  - \( Type(ecr(x)) \) gives representative element of equivalence class

Inference Rules

- \( x = y \)
  - let \( \text{ref}(\tau_1 \times \lambda_1) = \text{type}(ecr(x)) \)
  - \( \text{ref}(\tau_2 \times \lambda_2) = \text{type}(ecr(y)) \) in
    - if \( \tau_1 \neq \tau_2 \) then \( \text{cj}(\tau_1, \tau_2) \)
    - if \( \lambda_1 \neq \lambda_2 \) then \( \text{cj}(\lambda_2, \lambda_2) \)
  - \( \text{cj} \) conditionally joins the equivalence classes once one becomes \( \neq \perp \)
Rule for \(^*x = y\)

\[
\begin{aligned}
^*x &= y \\
\text{let } &\text{ref}(\tau_1, x) = \text{type}(\text{ecr}(x)) \\
&\text{ref}(\tau_2, \lambda_2) = \text{type}(\text{ecr}(y)) \text{ in} \\
\text{if } &\tau_1 = \perp \text{ then settype}(\tau_1, \text{ref}(\tau_2, \lambda_2)) \\
\text{else let } &\text{ref}(\tau_3, \lambda_3) = \text{type}(\tau_1) \text{ in} \\
&\text{if } \tau_2 \neq \tau_3 \text{ then cjoin}(\tau_3, \tau_2) \\
&\text{if } \lambda_2 \neq \lambda_3 \text{ then cjoin}(\lambda_3, \lambda_2)
\end{aligned}
\]

Steensgaard in Practice

- Commonly used, e.g., in ORC
- One-level Flow (by Das) a recent improvement
  - use directional assignment
  - precision as good as Anderson’s algorithm but as fast as Steensgaard in practice

Early Compiler Optimizations

- early = close to frontend
  - algebraic optimizations, CSE, value numbering & others
Catalog of Early Optimizations

- Constant folding and propagation
  - Covered earlier
  - New: sparse conditional copy propagation -- requires SSA form
- Scalar replacement of aggregates
  - To enable register allocation of record fields
- Algebraic simplification and reassociation
- Value numbering
  - Discover expressions that compute identical values
  - At basic block level & globally
- Copy propagation

Scalar Replacement of Aggregates

- Simple, yet effective
- Given a record r
  - Identify which fields f have scalar values
  - Ensure that neither r nor f are aliased
  - Assign field to temporary of same type
  - Replace accesses to field by accesses to temporary

Algebraic Simplification & Reassociation

- Use algebraic properties of operators to simplify expressions
  - Reassociation = use commutativity / associativity / distributivity to simplify
- Example: for integers i
  - i+0 = 0+i = i
  - 0·i = 0
  - 1·i = i·1 = i
  - i·0 = 0·i = 0
Algebraic Simplification

- Strength reduction:
  - $i^2 = i + i$
  - Addition is often faster than multiplication
    - On the Alpha integer multiply 8 cycles, addition 1 cycle!
    - $i^5 = i\times i^2 + i$ (= 2 cycles versus 8)

- Caveats
  - $(i-j) + (i-j) + (i-j) + (i-j) = 4(i-4j)$
  - RHS overflows on 32-bit machine when $i = 2^{30}$ and $j = 2^{30}-1$
  - Exceptions: $x = 1/0$ (what if code is not reached?)

Addressing Expressions

- Overflow makes no difference
  - Unsigned operations so results guaranteed to be identical
  - All mathematical equalities are guaranteed to produce correct results

- Canonicalization
  - Optimization strategy:
    - Turn expression into sum of products form
    - Use commutativity to collect constants and loop-invariant parts together

Canonicalization Example

- For Pascal (row major order)
  - var a: array[lo1..hi1,lo2..hi2] of etype;
  - do $j = lo2$ to $hi2$ begin $a[i,j] := b + a[i,j]$ end
  
  Address($a[i,j]$) = base_a + $((i-lo1) * (hi2-lo2+1) + j-lo2) * w$
  
  = ($lo1 * (hi2-lo2+1) -lo2) *w + base_a + (hi2-
  lo2+1) * w * j + j * w$
  
  Only have to calculate $j * w$ in loop
  - Can strength-reduce to addition of $w$
Algebraic Simplification of Floating Point Expressions

- **Can almost never do it!** In particular the following are NOT true in FP arithmetic:
  - $x \times 0 = 0$
  - IEEE standard has representation for $\infty$ and $-\infty$, $0 = \text{NaN}$
  - NaN = Not a Number
  - Signaling and non-signaling NaNs (former cause exceptions, latter don’t)
  - $x + 0 = x$
  - If $x$ is signaling NaN, exception is raised when ‘+’ is executed
  - $(a+b)+c = (a+b) + c$
  - $1.0+(\text{MF}-\text{MF}) = 1.0$
  - $(1.0+\text{MF})-\text{MF} = 0.0$

Redundancy Elimination

- Several categories:
  - Value Numbering
    - local & global
  - Common subexpression elimination (CSE)
    - local & global
  - Loop-invariant code motion
  - Partial redundancy elimination
    - Most complex
    - Subsumes CSE & loop-invariant code motion

Value Numbering

- **Goal:** identify expressions that have same value
- **Approach:** hash expression to a hash code
- Then have to compare only expressions that hash to same value
Example

\[\begin{align*}
a &:= x \mid y \\
b &:= x \mid y \\
t1 &:= f z \\
  \text{if } t1 \text{ goto L1} \\
x &:= f z \\
c &:= x \& y \\
t2 &:= x \& y \\
  \text{if } t2 \text{ trap 30}
\end{align*}\]  

Global Value Numbering

- Generalization of value numbering for procedures
- Requires code to be in SSA form
- Crucial notion: congruence
  - Two variables are congruent if their defining computations have identical operators and congruent operands
  - Only precise in SSA form, otherwise the definition may not dominate the use

Value Graphs

- Labeled, directed graph
- Nodes are labeled with
  - Operators
  - Function symbols
  - Constants
- Edges point from operator (or function) to its operands
  - Number labels indicate operand position
Example

entry

receive n1(val)

i1 := 1

j1 := 1

B1

B3

i2 := \varphi_3(i1, j1)

j2 := \varphi_3(j1, j1)

i2 mod 2 = 0

B2

i3 := i3 + 1

j3 := j3 + 1

B4

i4 := i4 + 3

j4 := j4 + 3

B5

j2 > n1

exit

Congruence Definition

- Maximal relation on the value graph
- Two nodes are congruent iff
  - They are the same node, or
  - Their labels are constants and the constants are the same, or
  - They have the same operators and their operands are congruent
- Variable equivalence: x and y are equivalent at program point p iff they are congruent and their defining assignments dominate p

Global Value Numbering Algo.

- Fixed point algorithm
  - Partition nodes into congruent sets
  - Initial partition: nodes with same label are in same partition
  - Iterate: split partitions where operands are not congruent
Global CSE

- Data flow problem: available expressions
- Lattice
- Direction
- Gen & Kill sets
- Distributive?

Example

- Gen(a := a + b) = ?
- Gen(x := y op z) = ?
- Gen(x relop y) = ?

Loop-invariant Code Motion

- Goal: recognize computations that compute same value every iteration
- Move out of loop and compute just once
- Commonly useful for array subscript computations, e.g., a[j] in loop over I

- Two steps
  - Analysis: recognize invariant computations
  - Transformation:
    - Code hoisting if used within loop
    - Code sinking if used after loop
Algorithm

- **Base cases:** expression
  - is a constant
  - is a variable all of whose definitions are outside of loop
- **Inductive cases**
  - Idempotent computation all of whose arguments are loop-invariant
  - It's a variable use with only one reaching definition and the rhs of that def is loop-invariant

Example

- See board

Code Motion

- When is code motion of invariant computation $S$: $z := x \text{ op } y$ to loop preheader legal?
  - **Sufficient conditions**
    - $S$ dominates all loop exits
    - Otherwise may execute $S$ when never executed otherwise
    - Can relax if $S$ idempotent (possibly slowing down program)
    - $S$ is only assignment to $z$ in loop & no use of $z$ in loop is reached by any def other than $S$
    - Otherwise may reorder def/uses
    - Unnecessary in SSA form