Loop Invariant Code

- **Idea**: if a computation produces same result in all loop iterations, move it out of the loop
- **Example**: for (i=0; i<10; i++)
  \[ a[i] = 10*i + x*x; \]
- **Expression** \( x*x \) produces the same result in each iteration; move it of the loop:
  \[ t = x*x; \]
  for (i=0; i<10; i++)
  \[ a[i] = 10*i + t; \]

Loop Invariant Computation

- An instruction \( a = b \ OP \ c \) is loop-invariant if each operand is:
  - Constant, or
  - Has all definitions outside the loop, or
  - Has exactly one definition, and that is a loop-invariant computation
- Reaching definitions analysis computes all the definitions of \( x \) and \( y \) which may reach \( t = x \ OP \ y \)

Code Motion

- Suppose \( a = b \ OP \ c \) is loop-invariant
- We want to hoist it out of the loop
- Code motion of a definition \( d; a = b \ OP \ c \) in pre-header is valid if:
  1. Definition \( d \) dominates the nodes after the loop exit where \( a \) is live
  2. There is no other definition of \( a \) in loop
  3. All uses of \( a \) in the loop can only be reached from \( d \)
     otherwise, just move the computation \( b \ OP \ c \)

Other Issues

- Preserve dependencies between loop-invariant instructions
  when hoisting code out of the loop
  for (i=0; i<N; i++)
  \[
  \begin{align*}
  x &= y*z; \\
  t &= x*x;
  \end{align*}
  \]
  \[ a[i] = 10*i + x*x; \]
  for (i=0; i<N; i++)
  \[ a[i] = 10*i + t; \]
- Nested loops: apply loop invariant code motion algorithm multiple times
  for (i=0; i<N; i++)
  \[
  \begin{align*}
  t1 &= x*x; \\
  t2 &= t1 + 10*i; \\
  a[i][j] &= x*x + 10*i + 100*j;
  \end{align*}
  \]
  for (j=0; j<M; j++)
  \[ a[i][j] = t2 + 100*j; \]

Induction Variables

- An induction variable is a variable in a loop, whose value is linear with respect to the loop iteration number
  \[ v = f(i) \]
  \[ f(i) = c*i + d \]
- Observation: linear combinations of linear functions are linear functions
  - Consequence: linear combinations of induction variables are induction variables
Induction Variables
- Two categories of induction variables
- Basic induction variables: only incremented in loop body
  \[ i = i + c \]
  where \( c \) is a constant (positive or negative)
- Derived induction variables: expressed as a linear function of an induction variable
  \[ k = c^j + d \]
  where:
  - either \( j \) is basic induction variable
  - or \( j \) is derived induction variable relative to \( i \) and:
    1. No definition of \( j \) outside the loop reaches definition of \( k \)
    2. \( k \) is not defined between the definitions of \( j \) and \( k \)

Families of Induction Variables
- Each basic induction variable defines a family of induction variables
  - Each variable in the family of \( i \) is a linear function of \( i \)
- A variable \( k \) is in the family of basic variable \( i \) if:
  1. \( k = i \) (the basic variable itself)
  2. \( k \) is a linear function of other variables in the family of \( i \):
    \[ k = c^j + d \]
    where \( j \) is a family
- A triple \( <i, a, b> \) denotes an induction variable \( k \) in the family of \( i \) such that:
  - Triple for basic variable \( i \) is \( <i, 1, 0> \)

Dataflow Analysis Formulation
- Detection of induction variables: can formulate problem using the dataflow analysis framework
  - Analyze loop body sub-graph, except the back edge
  - Analysis is similar to constant folding
- Dataflow information: a function \( F \) that assigns a triple to each variable:
  \[ F(k) = <i, a, b> \]
  if \( k \) is an induction variable in family of \( i \)
  \[ F(k) = \bot \] : \( k \) is not an induction variable
  \[ F(k) = \top \] : don't know if \( k \) is an induction variable

Dataflow Analysis Formulation
- Meet operation: if \( F_1 \) and \( F_2 \) are two functions, then:
  \[ (F_1 \cap F_2)(v) = \begin{cases} <i, a, b> & \text{if } F_1(v) = <i, a, b> \text{ and } F_2(v) = <i, a, b> \\ \bot, & \text{otherwise} \end{cases} \]
  (in other words, use a flat lattice)
- Initialization:
  - Detect all basic induction variables
  - At loop header: \( F(i) = <i, 1, 0> \) for each basic variable \( i \)
- Transfer function:
  - consider \( F \) is information before node \( n \)
  - Compute information \( F \) after \( n \)

Dataflow Analysis Formulation
- For a definition \( k = j + c \), where \( k \) is not basic induction variable
  \[ F(v) = <i, a, b + c>, \text{ if } v = k \text{ and } F(j) = <i, a, b> \]
  \[ F(v) = F(v), \text{ otherwise} \]
- For a definition \( k = j + c \), where \( k \) is not basic induction variable
  \[ F(v) = <i, a + c, b + c>, \text{ if } v = k \text{ and } F(j) = <i, a, b> \]
  \[ F(v) = F(v), \text{ otherwise} \]
- For any other instruction and any variable \( k \) in \( \text{def}(n) \):
  \[ F(v) = \bot, \text{ if } F(v) = <k, a, b> \]
  \[ F(v) = F(v), \text{ otherwise} \]

Strength Reduction
- Basic idea: replace expensive operations (multiplications) with cheaper ones (additions) in definitions of induction variables
  \[ s = 3i + 1 \]
  while \( i < 10 \) {
    \( j = -1 \)
    while \( j < 10 \) {
      \( \text{a[j]} = \text{a[j]} - 2; \)
      \( i = i + 1; \)
    }
    \( s = s + 6; \)
  }
- Benefit: cheaper to compute \( s = s + 6 \) than \( j = 3i \)
  - \( s = s + 6 \) requires an addition
  - \( j = 3i \) requires a multiplication
General Algorithm

- **Algorithm:**
  For each induction variable $j$ with triple $<i,a,b>$ whose definition involves multiplication:
  1. create a new variable $s$
  2. replace definition of $j$ with $j=s$
  3. immediately after $i=i+c$, insert $s = s+a*c$
     (here $a*c$ is constant)
  4. insert $s = a^n+b$ into preheader
- **Correctness:**
  this transformation maintains the invariant that $s = a^n+b$

Strength Reduction

- Gives opportunities for copy propagation, dead code elimination

$$s = 3^n+1;$$
$$\text{while } (i<10) \{$$
  $$j = j;$$
  $$a[j] = a[j] - 2;$$
  $$i = i+2;$$
  $$s = s+6;$$
$$\}$$

$$s = 3^n+1;$$
$$\text{while } (i<10) \{$$
  $$a[i] = a[i] - 2;$$
  $$i = i+2;$$
  $$s = s+6;$$
$$\}$$

Induction Variable Elimination

- **Idea:** eliminate each basic induction variable whose only uses are in loop test conditions and in their own definitions $i = i+c$
  - rewrite loop test to eliminate induction variable

$$s = 3^n+1;$$
$$\text{while } (i<10) \{$$
  $$a[i] = a[i] - 2;$$
  $$i = i+2;$$
  $$s = s+6;$$
$$\}$$

- When are induction variables used only in loop tests?
  - Usually, after strength reduction
  - Use algorithm from strength reduction even if definitions of induction variables don’t involve multiplications

Where We Are

- Defined dataflow analysis framework
- Used it for several analyses
  - Live variables
  - Available expressions
  - Reaching definitions
  - Constant folding
- Loop transformations
  - Loop invariant code motion
  - Induction variables
- Next:
  - Pointer alias analysis
**Pointer Alias Analysis**

- Most languages use variables containing addresses
  - E.g. pointers (C, C++, Java), call-by-reference parameters (Pascal, C++, Fortran)
- **Pointer aliases**: multiple names for the same memory location, which occur when dereferencing variables that hold memory addresses
- **Problem**:
  - Don’t know what variables read and written by accesses via pointer aliases (e.g. *p=y, x=*p, p=f=y, x=p,f, etc.)
  - Need to know accessed variables to compute dataflow information after each instruction

**Alias Analysis Problem**

- **Goal**: for each variable v that may hold an address, compute the set Ptr(v) of possible targets of v
  - Ptr(v) is a set of variables (or objects)
  - Ptr(v) includes stack- and heap-allocated variables (objects)
- Is a “may” analysis: if x ∈ Ptr(v), then v may hold the address of x in some execution of the program
- **No alias information**: for each variable v, Ptr(v) = V, where V is the set of all variables in the program

**Dataflow Alias Analysis**

- **Dataflow analysis**: for each variable v, compute points-to set Ptr(v) at each program point

  **Dataflow information**: set Ptr(v) for each variable v
  - Can be represented as a graph G ⊆ 2 x V
  - Nodes = V (program variables)
  - There is an edge v→u if u ∈ Ptr(v)

  \[
  \begin{align*}
  \text{Ptr}(x) &= \{y\} \\
  \text{Ptr}(y) &= \{x,t\}
  \end{align*}
  \]

  **Example**

  \[
  \begin{array}{c}
  z \\
  y \\
  x \\
  t
  \end{array}
  \]

**Pointer Alias Analysis**

- **Worst case scenarios**
  - *p = y* may write any memory location
  - x = *p may read any memory location
  - Such assumptions may affect the precision of other analyses
- **Example 1**: Live variables
  - before any instruction x = *p, all the variables may be live
- **Example 2**: Constant folding
  - a = 1; b = 2, *p = 0; c = a+b;
  - c = 3 at the end of code only if *p is not an alias for a or b!
  - **Conclusion**: precision of result for all other analyses depends on the amount of alias information available
  - hence, it is a fundamental analysis

**Simple Alias Analyses**

- **Address-taken analysis**:
  - Consider AT = set of variables whose addresses are taken
  - Then, Ptr(v) = AT, for each pointer variable v
  - Addresses of heap variables are always taken at allocation sites (e.g. x = new int[2], x = malloc(6))
  - Hence AT includes all heap variables
- **Type-based alias analysis**:
  - If v is a pointer (or reference) to type T, then Ptr(v) is the set of all variables of type T
  - Example: p.f and q.f can be aliases only if p and q are references to objects of the same type
  - Works only for strongly-typed languages

**Dataflow Alias Analysis**

- **Dataflow Lattice**: \((2^{x+y}, \supseteq)
  - V x V is set of all possible points-to relations
  - “may” analysis: top element is \(\emptyset\), meet operation is \(\cup\)
- **Transfer functions**: use standard dataflow transfer functions:
  - out[I] = \((\{I\} \cap \text{kill[I]}) \cup \text{gen[I]}\)
  - \(p = \text{addr q}\) \(\quad \text{kill[I]} = \{p\} \times V \quad \text{gen[I]} = \{(p,q)\}\)
  - \(p = q\) \(\quad \text{kill[I]} = \{p\} \times V \quad \text{gen[I]} = \{p\} \times \text{Ptr}(q)\)
  - \(p = q\) \(\quad \text{kill[I]} = \{p\} \times V \quad \text{gen[I]} = \{p\} \times \text{Ptr}(q)\)
  - For all other instruction, \(\text{kill[I]} = \emptyset\), \(\text{gen[I]} = \emptyset\)
- **Transfer functions**: are monotonic, but not distributive!
Alias Analysis Example

Program

\[
\begin{align*}
  &x = \&a; \\
  &y = \&b; \\
  &c = \&i; \\
  &\text{if}(i) \ x = y; \\
  &*x = c
\end{align*}
\]

\[
\begin{align*}
  &x = \&a \\
  &y = \&b \\
  &c = \&i \\
  &\text{if}(i) \\
  &x = y \\
  &*x = c
\end{align*}
\]

CFG

Points-to Graph (at the end of program)

\[
\begin{align*}
  &x \rightarrow a \\
  &y \rightarrow b \\
  &c \rightarrow i
\end{align*}
\]

Alias Analysis Uses

- Once alias information is available, use it in other dataflow analyses.

- Example: Live variable analysis

  Use alias information to compute \( \text{use}[I] \) and \( \text{def}[I] \) for load and store statements:

  \[
  \begin{align*}
  x = \ast y & \quad \text{use}[n] = \{ y \} \cup \text{Pr}(y) \quad \text{def}[n] = \{ x \} \\
  *x = y & \quad \text{use}[n] = \{ x, y \} \quad \text{def}[n] = \text{Pr}(x)
  \end{align*}
  \]