Beyond Scalar Variables

Arrays: Treat as single variable, or use dependence analysis
Records: Treat as single variable, or each field a single variable
Pointers: More difficult! Can provide alternate "name" for variable

Ex.  
\[
\begin{align*}
\text{int } &\text{n} \\
\text{int } &\text{*p} \\
\text{p} &\text{=} &\text{&n} \\
\text{n} &\text{=} \\
\text{*p} &\text{=} \\
\end{align*}
\]
both define n!

Conservative approach:
Ultra: All variables can be referenced by any pointer
Ultra+: All variables whose addresses are computed (eg &) can be referenced by any pointer

Aliasing

Two variables are aliased if they can refer to the same storage location.
Possible Sources: pointers, parameter passing, storage overlap...

Ex. Program main  
\[
\begin{align*}
\text{inf } &\text{a} \\
\text{call P(a,c)} &\text{ address of a passed} \\
\text{Proc P(x,y)} &\text{ x := } \\
\text{ y := x and y are aliased!} \\
\end{align*}
\]

Pointer Analysis, Alias Analysis

to get less conservative info

Needed for correct, aggressive optimization
Procedures: terminology

```
int a, e
Proc foo(b, c)  
  b := sqrt(c)  
  a := c  
end
Program main    
  int d
  call foo(a, d)  
end
```

At procedure call, formals bound to actuals, may be aliased

Ex. (b, a) , (c, d)

Globals, actuals may be modified, used

Ex. a, b

Alias Analysis

 Determines which variables aliased
 may vs must      some path vs all paths
 flow-sensitive vs flow-insensitive
                     depends on CFG paths vs independent

Flow-insensitive may: X and Y may refer to same location
                         (but at different points!)
Flow-insensitive must: X and Y must refer to same location
                                 (throughout program!)
Flow-sensitive may: At program point p, X may refer to same
                      location as Y1, Y2, ...
Flow-sensitive must: At point p, X must refer to location L
**Alias Information**

Language dependent

*Ex. C*

- Union type: fields can overlap in storage
- &: can compute address of variables
- pointers (with arithmetic)
- call-by-value parameter passing (but with pointers)

*Ex. Pascal*

- Variant records, variable parameters, typed pointers...

Once gathered, can propagate in language-independent manner

---

**Call Graphs**

Determines possible flow of control, interprocedurally

\[ G = (N, LE, s) \]

- \( N \) set of nodes (procedures)
- \( LE \) set of labelled edges
- \( n \rightarrow m \) edge if \( n \) calls \( m \)
- \( s \) start node (main procedure)

Qu: Why need call site labels? Why list?
Example Call Graph

```
proc main()
    call g()   (1)
    call g()   (2)
    call h()   (3)
end
proc g()
    call h()   (4)
    call i()   (5)
end
proc h()
end
proc i()
    proc j()
    end
    call g()   (6)
    call j()   (7)
end
```

Interprocedural Dataflow Analysis

Based on call graph: forward, backward

Gen, Kill: Need to summarize procedures per call

Flow sensitive: take procedure’s control flow into account

Flow insensitive: ignore procedure’s control flow

Difficulties:

- Hard, complex Flow sensitive alias analysis intractable
- Separate compilation?

Scale compiler can do both flow sensitive and insensitive

Most compilers ultraconservative, or flow insensitive
**Optimization**

Transformations that attempt to improve code by some measure
usually time or space

Not optimal, in general (undecidable)
Need to be safe, accurate same I/O behavior

Biggest open problem: Guidance of optimization choices

Can be classified by time of application
(early, middle, late, link-time)

- scope of application
  (local, bb, proc, interproc.)

Safety, Profitability, Cost, Effect on other optimizations

---

**Constant Folding**

Compile-time evaluation of expressions with constant operands

Used by many optimizations, especially constant propagation

Ex. \( cache\_size = 1024 \)

\[ = 25 \times 3 + cache\_size \]

1099

Constant propagate

Constant fold

Easy! But

May need type conversions
Handle exceptions?
Match target arithmetic?
Scalar Replacement of Aggregates

Use scalar temporary instead of aggregate variable

Why?
- Compiler may limit optimization to such scalars
- Can do better register allocation, constant propagation,...

```c
Ex. type complex
    real:
    imaj:
    C.real = ... C.imaj => T1 = ... T2
```

Particulary useful when small number of constant values

Can use constant propagation, dead code elimination to specialize code

Reassociation of expressions

Rewrite integer-valued expressing using algebraic rules

(commutativity, associativity, ...)

to obtain constant, loop-invariant, and variable subexpressions

Why?
- may make other optimizations possible or more effective
  (constant prop., code motion, ...)
- Can be applied to arithmetic, Boolean, relational, bit operators, ... 

Why not?
- Floating point operations, exceptions, disable CSE elimination

```c
Ex. addr(A) is base(A) + (i) * size
```

Do at each occurrence of A(i)

Data flow analysis can help
Value Numbering of Basic Blocks

Eliminates computations whose values are already computed in BB

Ex. \( \text{read}(i) \)  
\( j \leftarrow i + 1 \)  
\( k \leftarrow j \)  
\( m \leftarrow k + 1 \)  
j, k+1 same value; replace with j

value needn't be constant

Method: Assign value numbers to variables, values

hash number to (operator, value numbers of operands)

Ex. \( x := 3 \)
\( a := x + y \)
\( b := 3 + y \)
\( c := y \)
if \(((x + c) > 0)\) then...

<table>
<thead>
<tr>
<th>Value Number</th>
<th>Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
</tbody>
</table>

Global Copy Propagation

Given \( A := B \), replace later uses of A by B, as long as A,B not redefined

Ex.

\( \text{En} \)

\( \text{read}(b) \)
\( \bar{a} \leftarrow \bar{b} \)
\( \bar{c} \leftarrow \bar{a} \)
\( e := (\bar{a}^2 + 1) \)
\( d := a^2 \)
\( f := a + b \)

(with dead code elim)

\( \text{En} \)

\( \text{read}(b) \)
\( \bar{c} \leftarrow \bar{b} \)
\( e := (\bar{b^2} + 1) \)
\( d := b^2 \)
\( f := b + b \)
Global Copy Propagation Ex.

Ex.

![Diagram of Global Copy Propagation Example]

Solving Global Copy Propagation

$Gen(B)$ set of copy asssts $u = v$ in block $B$ where $u, v$ not reassigned later in $B$

$Kill(B)$ set of copy asssts killed in block $B$ by a later redefinition in $B$ of one of the variables

$CPin(B)$ set of copy asssts available at entry to block $B$

$CPout(B)$ set of copy asssts available at exit of block $B$

$CPin(B) = \bigcap_{P \text{ pred of } B} CPout(P)$

$CPout(B) = Gen(B) \bigcup (CPin(B) - Kill(B))$

what about $x := y \Rightarrow x := y$
Redundancy Elimination

Aim: Eliminate redundant operations in dynamic execution

Why occur?
Loop-invariant code: Ex: constant assignment in loop
Same expression computed Ex: addressing

Value numbering is an example
Requires dataflow analysis
Other optimizations:
  Constant subexpression elimination
  Loop-invariant code motion
  Partial redundancy elimination

Common Subexpression Elimination

Replace recomputation of expression by use of temp which holds value

Ex. (s1) y := a+b
     (s2) z := a+b

Ex. (s1) temp := a+b
     (s1') y := temp
     (s2) z := temp

Illegal?

How different from value numbering?

Ex. (s1) read(i)
     (s2) j := i + 1
     (s3) k := i
     (s4) l := k + 1

i + 1, k+1
no cse, same value number

Why need temp?
Local and Global
Local CSE (BB)

Ex. \( (s1) \ c := a + b \)
\( (s2) \ d := m \& n \)
\( (s3) \ e := a + b \)
\( (s4) \ m := 5 \)
\( (s5) \ if \ (m \& n) \ ... \)

\( (s1') \ t1 := a + b \)
\( (s2') \ d := m \& n \)
\( (s3') \ e := t1 \)
\( (s4') \ m := 5 \)
\( (s5') \ if \ (m \& n) \ ... \)

5 instr, 4 ops, 7 vars \quad 6 instr, 3 ops, 8 vars

Always better?

Method: keep track of expressions computed in block whose
operands have not changed value

\underline{CSE Hash Table}

\(+, a, b\)
\&(m, n)

Global CSE example

Assumes \( b \) is used later
**Global CSE**

An expression e is available at entry to B if on every path p from Entry to B, there is an evaluation of e at B’ on p whose values are not redefined between B’ and B.

**Solve by:**

1. *Find Available Expressions (Data flow problem)*
2. *For each available expression e*

   ![Diagram](image)

   Do backward search from e in CFG to find the evaluations of e

   Create new temp t to hold previous evaluations, and replace e by t

---

**Solving Available Expressions**

*Gen(B)* set of expressions evaluated in B available on exit from B

*Kill(B)* set of expressions killed by B

\[
\begin{align*}
  c &:= a+b \quad \text{Gen}(B) = \{w&z\} \\
  d &:= w&z \quad \text{Kill}(B) = \{\text{all expr. involving c,d,a,e}\} \\
  a &:= 5 \\
  e &:= e*7
\end{align*}
\]

Forward or Backward?

\[
\begin{align*}
  \text{AEin}(B) &= \bigcap_{p \text{ pred of } B} \text{AEout}(p) \\
  \text{AEout}(B) &= \text{Gen}(B) \cup (\text{AEin}(B) \setminus \text{Kill}(B))
\end{align*}
\]
Applying CSE

Create new temps for each occurrence of expression—necessary?

Is CSE always desirable?

When should CSE be applied?

*Forward Substitution: Inverse of CSE*

Replace copy by reevaluation of expression

Legal?