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.

```
int n
int *p
p := &n
n :=
*p :=
```

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.

```
call P(a,a)
 P(x,y)
   x :=
   y :=
```

address of a passed

x and y are aliased!

Ex. Fortran Equivalence, Common

Pointer Analysis, Alias Analysis

to get less conservative info

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

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

Constant propagate

:= 25 * 3 + cache_size

1099 **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,...

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

Ex. addr(A) is base(A) + (i-l) * 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. read(i)
j := i + 1
k := i
m := 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)

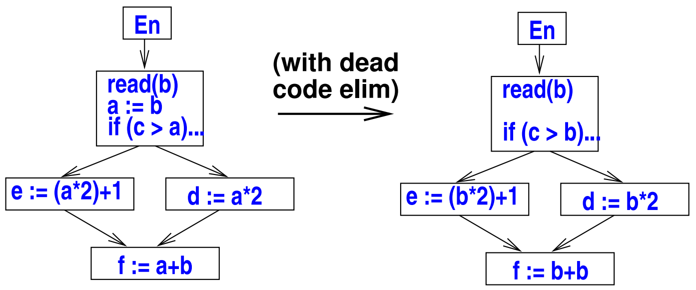
Ex. x := 3
a := x + y
b := 3 + y
c := y
if ((x + c) > 0) then...

	Value Number	**Hash Table**

Global Copy Propagation

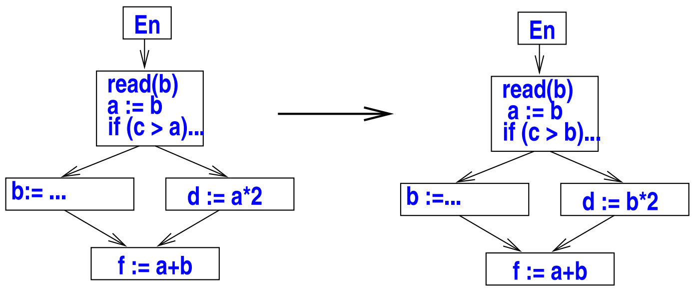
Given A:=B, replace later uses of A by B, as long as A,B not redefined

Ex.



Global Copy Propagation Ex.

Ex.



Solving Global Copy Propagation

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

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

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

CPout(B) set of copy assts 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