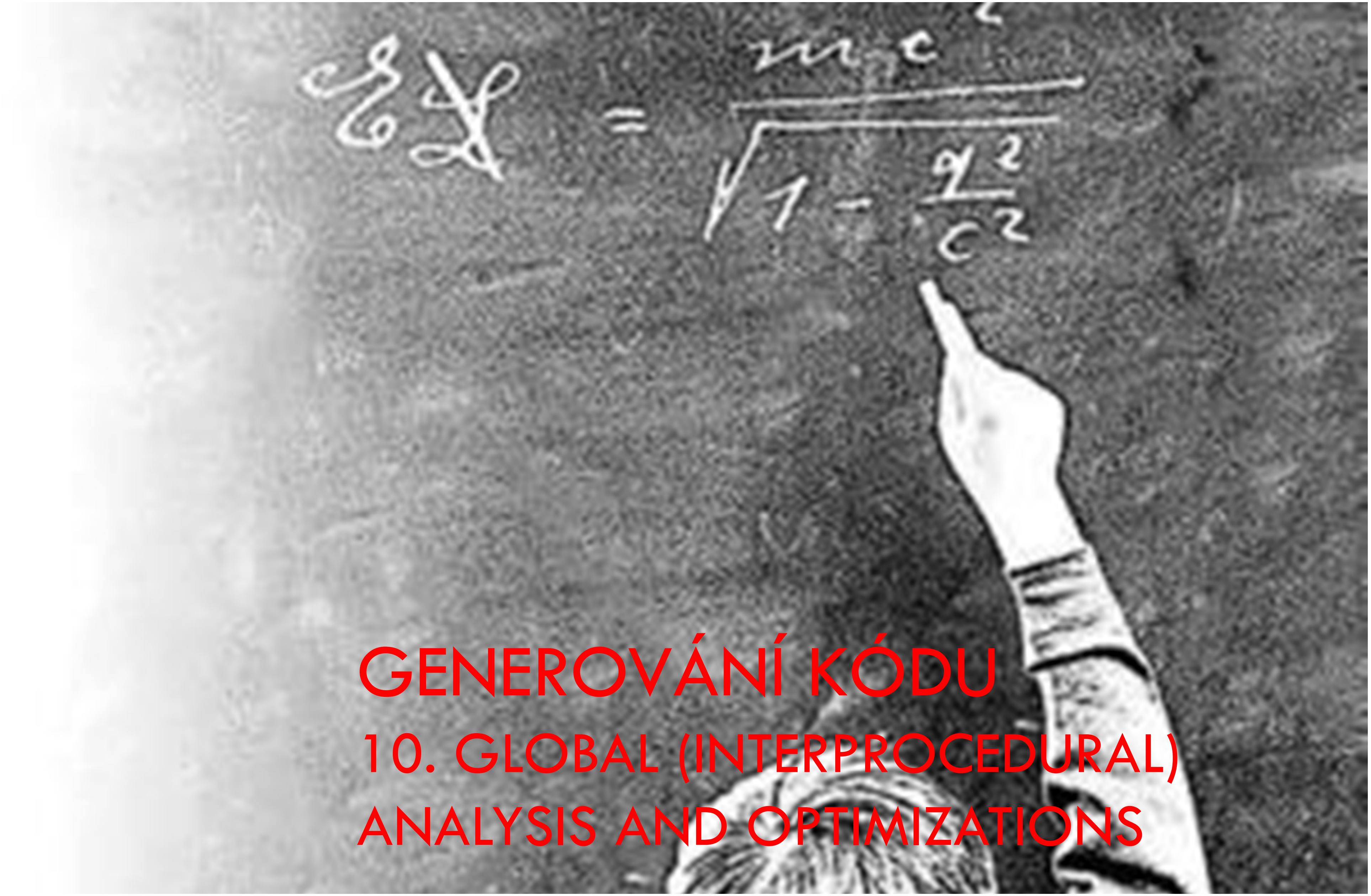


GENEROVÁNÍ KÓDU

10. GLOBAL (INTERPROCEDURAL) ANALYSIS AND OPTIMIZATIONS

2014 Jan Janoušek
MI-GEN

Evropský sociální fond
Praha & EU:
Investujeme do vaší budoucnosti

A detailed order of optimizations (from the book Muchnick: Advanced Compiler Design and Implementation)

Scalar replacement of array references
Data-cache optimizations

Procedure integration
Tail-call optimization
Scalar replacement of aggregates
Sparse conditional constant propagation
Interprocedural constant propagation
Procedure specialization and cloning
Sparse conditional constant propagation

Constant folding
Algebraic simplification and reassociation

Global value numbering
Local and global copy propagation
Sparse conditional constant propagation
Dead-code elimination
Local and global common-subexpression elimination
Loop-invariant code motion
Dead-code elimination
Code hoisting
Induction-variable strength reduction
Linear-function test replacement
Induction-variable removal
Unnecessary bounds-checking elimination
Control-flow optimizations

In-line expansion
Leaf-routine optimization
Shrink wrapping
Machine idioms
Tail merging
Branch optimizations and conditional moves
Dead-code elimination
Software pipelining, loop unrolling
Basic-block and branch scheduling
Register allocation
Basic-block and branch scheduling
Intraprocedural I-cache optimization
Instruction prefetching
Data prefetching
Branch prediction

Interprocedural register allocation
Aggregation of global references
Interprocedural I-cache optimization

Another order of optimizations (from GNU Compiler Collection Internals)

Remove useless statements
Mudflap declaration registration
Lower control flow
Lower exception handling control flow
Build the control flow graph
Find all referenced variables

Enter static single assignment form
Warn for uninitialized variables
Dead code elimination
Dominator optimizations
Redundant phi elimination
Forward propagation of single-use variables
Copy renaming
PHI node optimizations
May-alias optimization
Profiling
Lower complex arithmetic
Scalar replacement of aggregates
Dead store elimination
Tail recursion elimination
Forward store motion
Partial redundancy elimination
Loop invariant motion
Canonical induction variable creation
Induction variable optimizations
Loop unswitching
Vectorization
Tree level if-conversion for vectorizer
Conditional constant propagation
Folding builtin functions
Split critical edges
Partial redundancy elimination
Control dependence dead code elimination
Tail call elimination
Warn for function return without value
Mudflap statement annotation
Leave static single assignment form

RTL generation
Generate exception handling landing pads
Cleanup control flow graph
Common subexpression elimination
Global common subexpression elimination.
Loop optimization
Jump bypassing
If conversion
Web construction
Life analysis
Instruction combination
Register movement
Optimize mode switching
Modulo scheduling
Instruction scheduling
Register class preferencing
Local register allocation
Global register allocation
Reloading
Basic block reordering
Variable tracking
Delayed branch scheduling
Branch shortening
Register-to-stack conversion
Final
Debugging information output

Interprocedural Analyses - Introduction

- Gathering information about the whole program instead of a single procedure

- It can be possibly a part of the linker (in the case of separate modules compiling).

- It often uses the same (or similar) methods as in local (procedural) analysis.

- Not so frequent in current real compilers.

Introduction, contd.

- Interprocedural analysis

Example: alias analysis

- Interprocedural optimization - program transformation that involves more than one procedure in the program

Examples: inlining

Alias analysis

- Two expressions are aliases if they denote the same memory location at a program point

p = new Object

q = p => here, *p and *q alias

- Aliases arise with:
  - pointers
  - call by reference
  - array indexing
  - C union

Aliasing examples

- Pointers

```
int *p, i;
p = &i; // *p and i alias
```

- Call by reference

```
void proc(int& a, int &b) { ... }

proc(x, x); // a and b alias in proc
proc(x, glob); // b and glob alias in proc
```

```
int i, j, a[N];
i = j; // a[i] and a[j] alias
```

Aliasing examples

- Why important?

`*p = a + b;`
`y = a + b;` // if *p aliases a or b, 2nd a+b is NOT redundant

- If not done, other analyses must be very conservative

Trivial alias analysis

- Assume nothing must alias
- Assume all pointer derefs may alias
- Assume variables whose address is taken (and globals) may alias all pointer derefs

```
p = &a; //*q and a may alias

a = 3; b = 4;
*q = 5; //*q and b do not alias, so b is 4. a may be 3 or 5.
```

Type-based alias analysis

- If strongly typed language, we can use type information too
  - Two variables with incompatible types cannot alias

  Possibly:
  - String vs. List – cannot alias
  - String vs. Object – may alias

Points-to analysis

```
if (c)
      p = &x;
else
      p = &y;
```

- Represent alias analysis results as points-to relation
  - p -> {x,y}
  - p can contain the address of x and y
  - “p points to x or y”

An Interprocedural Example – context analysis


```
int id(int x) {return x;}

void p() {a=2; b=id(a);…}
void q() {c=3; d=id(c);…}
```

- If we distinguish p calling id from q calling id, then we can discover b=2 and d=3.
- Otherwise, we think b, d = {2, 3}.

General Complexities of Various Types of Analyses

- Insensitive: proportional to size of program (number of variables).
- Flow-Sensitive: size of program, squared (points times variables).
- Context-Sensitive: worst-case, may be exponential in program size (acyclic paths through the code).

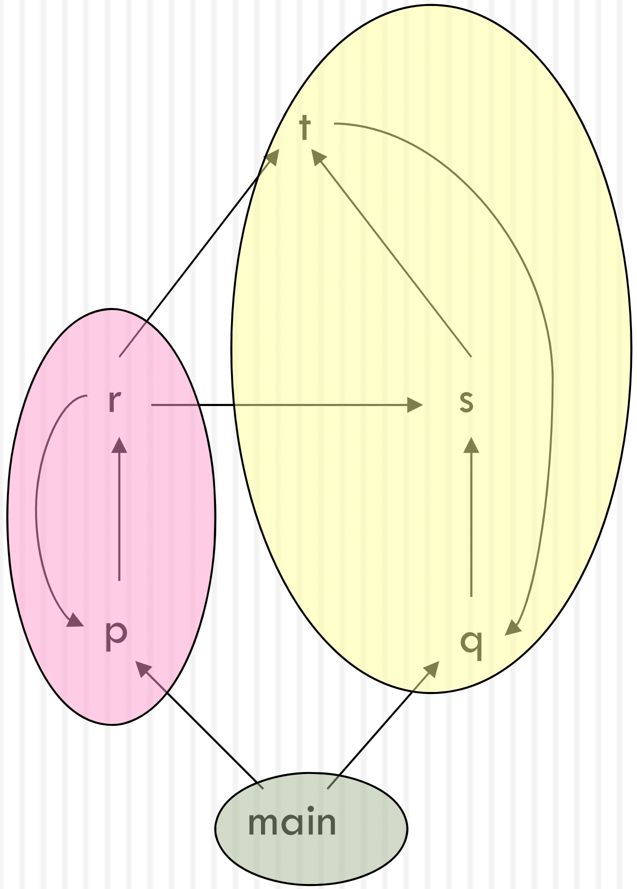
Context-Sensitivity

- We care how we got there (to the function, variable, …) – **context sensitivity.**

Context Sensitivity, contd

- Can distinguish paths to a given point.
- Loops and recursive calls lead to an infinite number of contexts…often strong components of the calling graph are collapsed to a single group.
- “Context” becomes the sequence of groups on the calling stack (in the case of procedures…).

Example: Calling Graph



Contexts:

Green
Green, pink
Green, yellow
Green, pink, yellow

Calling Graphs – other examples

On the board

Example of Implementation of interprocedural analyse with the use of logic programming

Logical Representation:

- There has been recent success with a logical formulation, involving predicates.

- Example: Reach(d,x,i) = “definition d of variable x can reach point i.”

Datalog --- (1) – Prolog like description (see Dragon book)

Atom = Reach(d,x,i)

Predicate

Arguments:
variables or constants

Literal = Atom or NOT Atom

Rule = Atom :- Literal & … & Literal

Make this
atom true
(the *head*).

The *body* :
For each assignment of values
to variables that makes all these
true …

Example: Datalog Rules


```
Reach(d,x,j) :- Reach(d,x,i) &
                StatementAt(i,s) &
                NOT Assign(s,x) &
                Follows(i,j)
Reach(s,x,j) :- StatementAt(i,s) &
                Assign(s,x) &
                Follows(i,j)
```

EDB Vs. IDB Predicates

- Some predicates come from the program, and their tuples are computed by inspection.
  - Called *EDB*, or *extensional database* predicates.

- Others are defined by the rules only.
  - Called *IDB*, or *intensional database* predicates.

Iterative Algorithm for Datalog

- Start with the EDB predicates = "whatever the code dictates," and with all IDB predicates empty.

- Repeatedly examine the bodies of the rules, and see what new IDB facts can be discovered from the EDB and existing IDB facts.

Using for Pointer Analysis

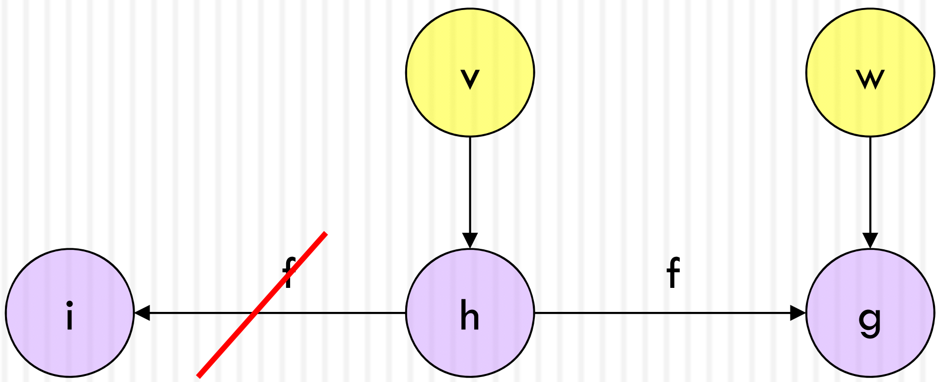
- Flow/context insensitive analysis.
  - Local variables, which point to:
  - Heap objects, which may have fields that are references to other heap objects.

Representing Heap Objects

- A heap object is named by the statement in which it is created.
- Note many run-time objects may have the same name.
- Example: `h: T v = new T;` says variable v can point to (one of) the heap object(s) created by statement h.

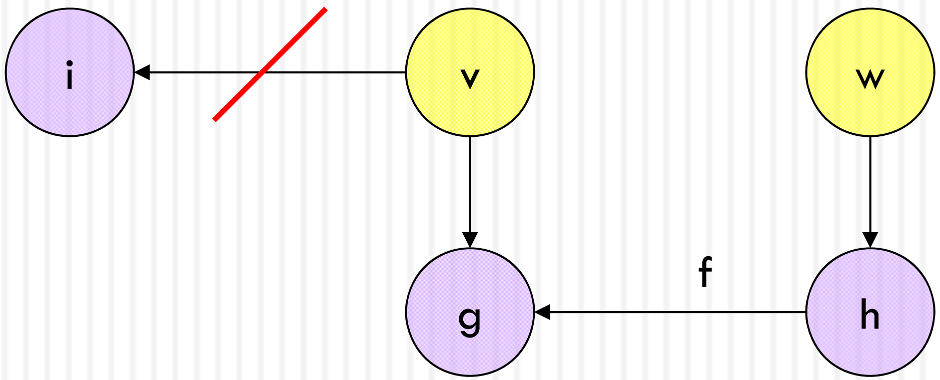
Other Relevant Statements

- `v.f = w` makes the f field of the heap object h pointed to by v point to what variable w points to.



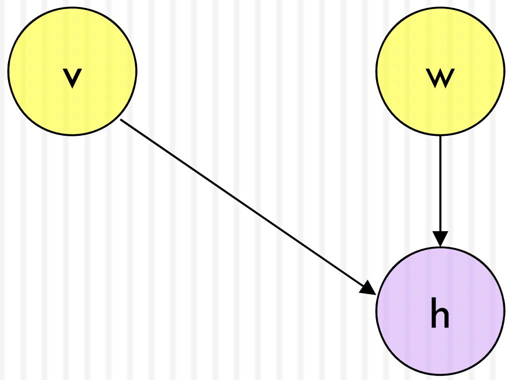
Other Statements --- (2)

- `v = w.f` makes v point to what the f field of the heap object h pointed to by w points to.



Other Statements --- (3)

- `v = w` makes v point to whatever w points to.
  - *Interprocedural Analysis* : Also models copying an actual parameter to the corresponding formal or return value to a variable.



EDB Relations

- The facts about the statements in the program and what they do to pointers are accumulated and placed in several EDB relations.

- Example: there would be an EDB relation Copy(To,From) whose tuples are the pairs (v,w) such that there is a copy statement `v=w`.

Convention for EDB

- Instead of using EDB relations for the various statement forms, we shall simply use the quoted statement itself to stand for an atom derived from the statement.
- Example: “v=w” stands for Copy(v,w).

IDB Relations

- Pts(V,H) will get the set of pairs (v,h) such that variable v can point to heap object h.

- Hpts(H1,F,H2) will get the set of triples (h,f,g) such that the field f of heap object h can point to heap object g.

Datalog Rules

1. Pts(V,H) :- "H: V = new T"
2. Pts(V,H) :- "V=W" & Pts(W,H)
3. Pts(V,H) :- "V=W.F" & Pts(W,G) & Hpts(G,F,H)
4. Hpts(H,F,G) :- "V.F=W" & Pts(V,H) & Pts(W,G)

Example


```
T p(T x) {
  h: T a = new T;
     a.f = x;
     return a;
}
void main() {
  g: T b = new T;
     b = p(b);
     b = b.f;
}
```

Apply Rules Recursively --- Round 1


```
T p(T x) {h: T a = new T;
      a.f = x; return a;}
 void main() {g: T b = new T;
      b = p(b); b = b.f;}
```

Pts(a,h)

Pts(b,g)

Apply Rules Recursively --- Round 2


```
T p(T x) {h: T a = new T;
      a.f = x; return a;}
void main() {g: T b = new T;
      b = p(b); b = b.f;}
```

$$\frac{\begin{array}{l}\text{Pts(a,h)}\\ \text{Pts(b,g)}\end{array}}{\begin{array}{l}\text{Pts(x,g)}\\ \text{Pts(b,h)}\end{array}}$$

Apply Rules Recursively --- Round 3


```
T p(T x) {h: T a = new T;
      a.f = x; return a;}
void main() {g: T b = new T;
      b = p(b); b = b.f;}
```

Pts(a,h)

Pts(b,g)

Pts(x,g)

Pts(b,h)

Pts(x,h)

Hpts(h,f,g)

Apply Rules Recursively --- Round 4


```
T p(T x) {h: T a = new T;
      a.f = x; return a;}
void main() {g: T b = new T;
      b = p(b); b = b.f;}
```

Pts(a,h)

Pts(b,g)

Pts(x,g)

Pts(b,h)

Pts(x,h)   Hpts(h,f,g)

Hpts(h,f,h)

Extension to Flow Sensitivity

- IDB predicates need additional arguments B, I.
  - B = block number.
  - I = position within block, 0, 1,…, n for n -statement block.
    - Position 0 is before first statement, position 1 is between 1st and 2nd statement, etc.

Example of Rules: Flow Sensitive Pointer Analysis

Pts(V,H,B,I+1) :- "B,I: H: V = new T"

I is local,
H is a global
index of
object-creating
statements.

Pts(V,G,B,I+1) :- "B,I: W = new T" & V != W & Pts(V,G,B,I)

Pts(V,G,B,I+1) :- "B,I: W.f = X" &
Pts(V,G,B,I)

Notice W=V OK

Pts(V,G,B,0) :- Pts(V,G,C,*n*) & "C is a predecessor block of B with *n* statements"

Handles all control-flow information within the flow graph. Hpts similar.

Adding Context Sensitivity

- Include a component C = context.
  - C doesn't change within a function.
  - Call and return can extend the context if the called function is not mutually recursive with the caller.

Example of Rules: Context Sensitive

Pts(V,H,B,I+1,C) :- "B,I: V=W" &
Pts(W,H,B,I,C)

Pts(X,H,B0,0,D) :- Pts(V,H,B,I,C) & "B,I: call P(…,V,…)" & "X is the corresponding actual to V in P" & "B0 is the entry of P" & "context D is C extended by P"