Chapter 10
Interprocedural Analysis and Optimization
Sections 10.2.6 through 10.7

Tim Harvey

April 20, 1995

Comp 515

Overview

- kill analysis
- call graph construction
- interprocedural optimization
- whole program compilation

Kill Analysis

kill(p) – the set of all variables that must be modified on every path through procedure p.

kill is a flow-sensitive problem; instead, we can calculate *nkill(p)* for a procedure using the following equation:

$$nkill(b) = \bigcup_{c \,\in\, succ(b)} thru(b,c) \;\cap\; nkill(c)$$

...where $thru(b,c)$ is the set of variables which do not get killed on some path through b to c.

This equation can be solved iteratively.

Kill Analysis

But what about the case of kill analysis for the whole program?

We will require two levels of computation:

- kill analysis on the control flow graph
- kill analysis on the call graph

We first compute the kill set for the procedure, then we propagate this information around the call graph.

Each time we change a procedure's kill set, we have to propagate the information around the call graph and recompute other procedures' kill sets – expensive.

To alleviate some of the expense, we will compute the *reduced control-flow graph* for each procedure, where the nodes in the graph are call sites.

Building the Reduced Control-Flow Graph

Algorithm for constructing the reduced control-flow graph.

remove all back edges from the control-flow graph;
let $b0$ denote the procedure entry node;
mark $b0$ processed;
$\mathit{worklist} := \emptyset$;
for each $s \in \mathit{successors}(b0)$ **do**
    $\mathit{worklist} := \mathit{worklist} \cup \{(b0\ , s)\}$ **od**;

while $\mathit{worklist} \neq \emptyset$ **do**
    take an arbitrary element $(b0\ , s)$ from the worklist,
        such that all predecessors of s have already been
        processed or merged into b;
    if s is a call site **then**
        for each $t \in \mathit{successors}(s)$ **do**
            $\mathit{worklist} := \mathit{worklist} \cup \{(s,\ t)\}$ **od**;
        mark s as processed;
    else if s is the exit node **then** do nothing
    else /* s is a normal node */
        merge s into b;
        for each $t \in \mathit{successors}(s)$ **do**
            if $\mathit{THRU}[b,\ t]$ is undefined **then**
                $\mathit{THRU}[b,\ t] := \mathit{THRU}[b,\ s] \cup \mathit{THRU}[s,\ t]$
            else
                $\mathit{THRU}[b,\ t] :=$
                    $\mathit{THRU}[b,\ t] \cup (\mathit{THRU}[b,\ s] \cup \mathit{THRU}[s,\ t]\ )$
            fi
        od
    fi
od

Computing *nkill*(p)

```
for each b ∈ GTHRU(p) in reverse-topological order do
    if b is the exit node then nkill[b] := Ω
    else if b is a call site then
        nkill[b] := ∅
        for each successor s of b in GTHRU(p) do
            nkill[b] := nkill[b] ∪ [ nkill[s] ∩ THRU(b, s) ]
        od;
        nkill[b] := nkill[b] ∩ nkill[q]
            where q is in the procedure called at b
    else/* b is the entry node for GTHRU(p) */
        nkill[b] := ∅
        for each successor s of b in GTHRU(p) do
            nkill[b] := nkill[b] ∪ [ nkill[s] ∩ THRU(b, s) ]
        od;
        nkill[p] := nkill[b]
    fi
od
```

Observations for the interprocedural computation:

- $O(N^2 V)$ if the call graph is nonreducible
- $O((N+E)dV)$ if the call graph is reducible
- takes longer for reference formals, due to the register-shift problem
- sets are conservative, because aliasing is ignored – adds exponential time to compute

Symbolic Analysis

In constant propagation, we start without an idea of what value a variable has and then seek to prove it has some particular value.

In symbolic analysis, one motivation is to start with a set of values a variable must *not* have and then seek to prove it never takes on those values.

Example:

```
SUBROUTINE S(A,N,E)
    REAL A(N+E)
    INTEGER N, E
    DO I = 1, N
        A(I+E) = A(I) + B
    ENDDO
END
```

Symbolic Analysis

In general, the goal of *interprocedural symbolic analysis* is to prove facts about the values of variables that may hold on entry to a given subroutine or return from a given call site.

Three kinds:

- symbolic expression analysis – builds a representation of the variable based on the values of variables at that point in the program

$$x \Leftarrow (y + z) - 2 * w$$

- predicate analysis – describes linear relationships between pairs of variables

$$x + y = c$$

- range analysis – computes the range of values that a single variable may have

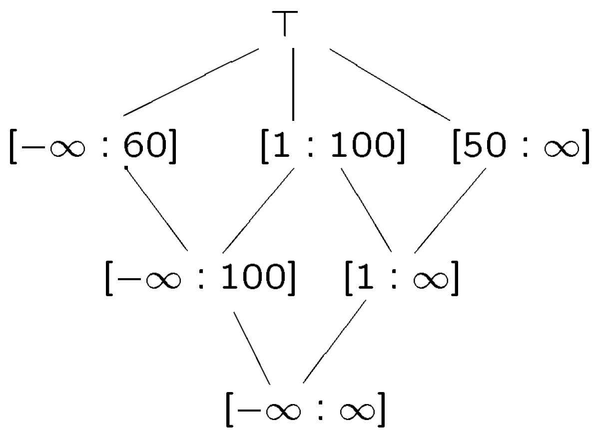
$$x \in [-\infty : -1, 14, 100 : \infty]$$

Range analysis

Example (K should be in [0 : 1]):

```
SUBROUTINE S(A,N,K)
    REAL A(0:N)
    INTEGER N, K
    DO I = 2, N
        A(I) = A(I) + A(K)
    ENDDO
END
```

We compute ranges using a lattice similar to the constant propagation lattice and a meet rule that picks the lower of two lower-bounds and the higher of the two upper-bounds



Symbolic Predicate Analysis

We can represent variables as linear functions of another variable:

$$x - y = c$$

More generally:

$$c_1 x - c_2 y = c_0$$

This analysis is complex; we can end up having to solve systems of equations.

Given:

$$d_1 y + d_2 z = d_0$$
$$e_1 x + e_2 z = e_0$$

...we can derive the following identity:

$$z = \frac{e_0 - e_1 x}{e_2}$$

...and from this, we can associate x and y:

$$d_1 y + d_2 (\frac{e_0 - e_1 x}{e_2}) = c_0$$

"The complexity of the analysis is purely a function of ambition..."
– **Ken Kennedy**

Array section analysis

Motivation:

```
SUBROUTINE S
    DIMENSION A(100,100)
    ...
    DO I = 1, N
        CALL SOURCE (A, I)  ! ASSIGNS TO A
        CALL SINK (A, I)    ! USES A
    ENDDO
END
```

To parallelize this loop, what must be true about the interaction between the calls $SOURCE$ and $SINK$?

Array section analysis

The loop carries a true dependence iff the set of values written to in `SOURCE` do not exactly equal those read in `SINK`.

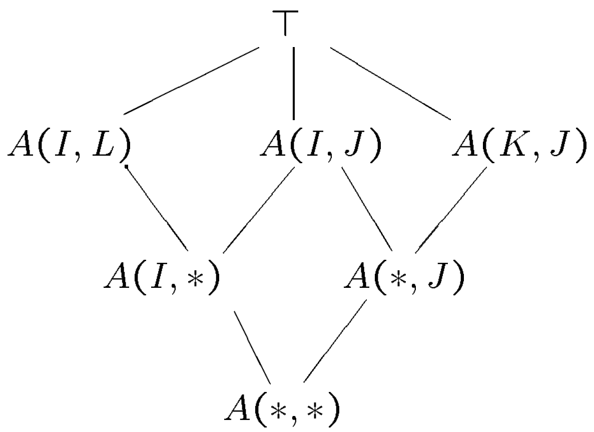
So, we'll represent arrays interprocedurally using a lattice.

The lattice should have the following properties:

- accuracy – within the usual tradeoffs
- efficient meet operation
- efficient dependence test
- finite descending chain property to ensure halting in the presence of recursion
- it should be possible to deal with parameter transformations that occur at call sites

Array section analysis

An example lattice:



This is a good lattice, because:

- the depth is $k+1$, where k is the number of subscripts
- the meet operation is $O(k)$
- intersection, used for dependence testing, is also $O(k)$

This is a weak lattice, because:

- limited accuracy – the array $A(10, 10)$, where the array is actually $A(100, 100)$ is represented as $A(*, *)$

Call graph construction

In a language which allows procedure parameters, construction of the call graph is non-trivial:

```
SUBROUTINE SUB1(X,Y,P)
    INTEGER X, Y
    CALL P(X,Y)
    RETURN
END
```

Obviously, we need some way to determine all of the possible values of P.

Unfortunately, this is not sufficient. (I.e., the naive solution is overly conservative.)

Call graph construction

To ensure the least-conservative call graph, we need to take "control flow" into account:

```
SUBROUTINE SUB2(X,P,Q)
    INTEGER X
    CALL P(X,Q)
    RETURN
END
```

Given the two following two calls:

```
CALL SUB2(X,P1,Q1)
CALL SUB2(X,P2,Q2)
```

Then the call to P can be described by the set

$$[P1\ \ P2]$$

The function parameter to P can be described by

$$[Q1\ Q2]$$

The closure of these sets is:

$$[(P1, Q1)\ (P1, Q2)\ (P2, Q1)\ (P2, Q2)]$$

The actual pairs consist of just:

$$[(P1, Q1)\ (P2, Q2)]$$

Call graph construction

Algorithm for computing procedure parameter tuples

for each procedure *p* in the program **do**
    PROCPARMS[*p*] := $\emptyset$ **od**;
W := $\emptyset$;
for each call site *s* in the program **do**
    if the call site passes procedure names to all
        procedure parms of the call procedure **do**
        let $t = \langle N_1, N_2, \ldots, N_k \rangle$ be the tuple of procedure
        names passed in order of the parameters to which
        they are passed;
        W := *W* $\cup$ $\{\langle t, p \rangle\}$, where *p* is the procedure called
    fi
od;
while *W* $\neq \emptyset$ **do**
    choose and remove $\langle t = \langle N_1, N_2, \ldots, N_k \rangle, p \rangle$ from *W*
    PROCPARMS[*p*] := *PROCPARMS*[*p*] $\cup$ $\{t\}$;
    let $\langle P_1, P_2, \ldots, P_k \rangle$ be the set of
        procedure parameters to which the elements of the
        tuple $t = \langle N_1, N_2, \ldots, N_k \rangle$ are mapped;
    for each call site *s* in *p* where P_i is passed
        as a procedure parameter **do**
        let $u = \langle M_1, M_2, \ldots M_k \rangle$ be the set of procedure names
            passed to the procedure *q* called at *s*, where each
            M_i is either the procedure name in the ith
            position or N_j if P_j is passed in the ith position;
        if $u \notin$ *PROCPARMS*[*q*] **then** *W* := *W* $\cup$ $\{\langle u, \text{tq} \rangle\}$ **fi**
    od
od

Interprocedural optimization

"All the analysis in the world won't improve code."
Keith Cooper

Three kinds:*

- inline substitution – insertion of a called routine into the body of the caller
- procedure cloning – creation of copies of the same procedure to allow each to be specialized
- hybrid optimizations – transformations involving more than one procedure

*This is not an exhaustive list!!!

Inline substitution

Copy a called procedure into the body of the caller, substituting actual parameters for formal parameters.

Example:

```
PROGRAM MAIN
    REAL A(100)
    CALL INPUT(A,N)
    DO I = 1, N
        CALL PROCESS(A,I)
    ENDDO
    CALL REPORT(A,N)
END
SUBROUTINE PROCESS(X,K)
    REAL X(*)
    X(K) = X(K) + K
    RETURN
END
```

Inlining produces:

```
PROGRAM MAIN
    REAL A(100)
    CALL INPUT(A,N)
    DO I = 1, N
        A(I) = A(I) + I
    ENDDO
    CALL REPORT(A,N)
END
```

Inline substitution

Advantages:

- procedure call overhead is eliminated
- procedure body code can be tailored for the environment
- optimizations can now be performed that were impossible

Disadvantages:

- exponential code growth
- different code shape can confuse compilers
- requires whole program recompilation when the inlined procedure is changed
- information, such as structure of arrays, is lost

Procedure Cloning

Make separate copies of a procedure and specialize them to specific call sites.

Example:

```
PROCEDURE UPDATE(A,N,IS)
    REAL A(N)
    INTEGER N, IS
    DO I = 2, N
        A(I*IS+1) = A(I) + 100
    ENDDO
END
```

When $IS = 0$:

```
PROCEDURE UPDATE(A,N,IS)
    REAL A(N)
    INTEGER N, IS
    A(1) = A(N) + 100
END
```

When $IS \neq 0$:

```
PROCEDURE UPDATE(A,N,IS)
    REAL A(N)
    INTEGER N, IS
    A(IS+1:N*IS+1) = A(1:N) + 100
END
```

Hybrid interprocedural optimizations

Hybrid, because these optimizations involve changing code in more than one procedure.

Example (loop embedding):

```
PROGRAM MAIN
    REAL A(100)
    CALL INPUT(A,N)
    DO I = 1, N
        CALL PROCESS(A,I)
    ENDDO
    CALL REPORT(A,N)
END
SUBROUTINE PROCESS(X,K)
    REAL X(*)
    X(K) = X(K) + K
    RETURN
END
```

We can move the loop *into* the subroutine as follows:

```
SUBROUTINE PROCESS(X,K,N)
    REAL X(*)
    DO K = 1, N
        X(K) = X(K) + K
    ENDDO
END
```

Whole-program compilation

Without extra knowledge, recompilation of a single routine will force a recompilation of the entire program – users won't stand for this in a compiler.

Goal: keep track of which facts were relied on by the optimizer and watch for those facts to change.

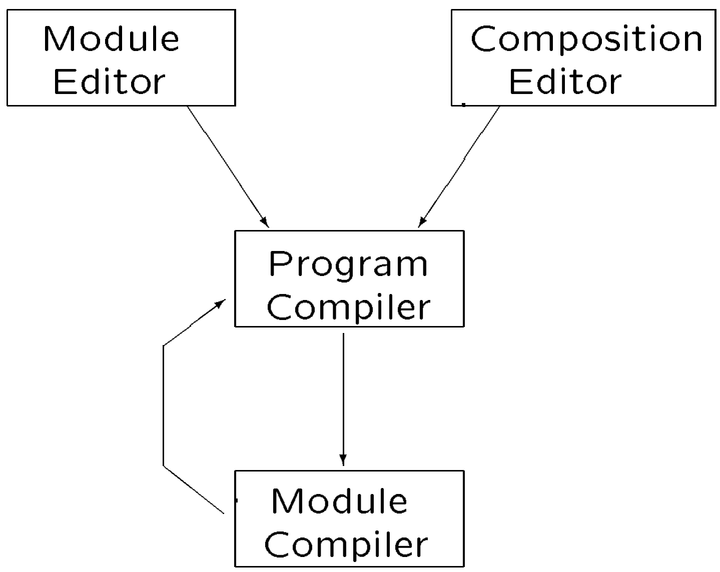
This implies that we need to store the information for later use.

We also need some way to represent the information about the facts used.

Whole-program compilation

This requires a special compilation *system*:

- a compiler to build the intermediate representation
- a compiler for interprocedural information
- a compiler for optimization
- a program manager that keeps track of changes and stores intermediate representations



Separate compilation

The compilation system is only half the story.

When the module compiler runs, it needs some way to communicate its decisions to enable recompilation analysis.

For example:

```
SUBROUTINE P(S,X,Y)
    INTEGER X, Y
    X = 5    Y = 10
    CALL S(X)
    RETURN X+Y
END
```

Assuming that the the MOD set for S does not include X, the constant propagator can remove the addition in the $RETURN$ statement.

But what if the MOD set for S changes? We need the $MUSTNOTMOD$ set for P, which will include X.

When S is recompiled, we will compare its new MOD set against P's $MUSTNOTMOD$ set.

If the intersection is nonempty, we need to recompile P as a result of recompiling S.