I. Overview
   A. We’ve been doing optimization within a routine
      1. Making some assumptions about calls
      2. Making assumptions about the parameters to the routine
      3. We should be able to do better if we take routines into account
   B. What are the problems
      1. Separate compilation
         a. You don’t have all the procedures that are called
         b. You won’t have library code in any case
      2. Determining what is called
         a. C: use of function pointers
            i. Track what the pointer can point to
         b. Java/C++: virtual methods
            i. Any inherited method of the called type
               ➢ Can vary with time (independent compilation)
            ii. Method may be determinable from argument type
         c. Java :: Use of reflection
            i. C++: look up names in dynamic symbol table and jump
   C. What do we want from interprocedural optimization
      1. Think of the CALL instruction
         a. What do we determine from instructions
         b. We would like to be able to handle the call accordingly
      2. MODIFYS relation
      3. Flow analysis functions
         a. LIVEUSE, LIVEKILL
         b. ANTLOC, KILL
         c. Value numberings (constant propagation)
      4. What assumptions do we currently make re parameters
         a. What if these are constant for a call
   D. Note that this doesn’t have to be particularly accurate
1. Simply identify routines with no external side effects or memory stores
   a. All getters
   b. Many other routines are of this form (C++ const methods)
   c. Probably 50% of the calls
2. Then the calls to these have no side effects (might affect USE)

II. Finding the call graph
   A. First problem is determining what is called at each call site
      1. Why is this difficult
         a. Calls through a pointer
         b. Virtual calls (which are calls through a pointer)
         c. Reflection
      2. How might you handle this
         a. Assume all routines on call through a pointer
            i. Track potential values for a pointer (there are often only one or two)
         b. Assume all methods that can inherit for virtual calls
            i. What’s wrong with this (when is it an over approximation)
            ii. What about Object.toString()
         c. Can you do better
   B. Virtual methods
      1. Handle by using type analysis to determine viable types
      2. Determine the type of each memory reference
         a. Assumes a strongly typed language
         b. Can be done statically or through flow analysis
      3. Statically: use the declared types rather than the called type
         a. This is a bit general, but better than nothing
         b. Collections classes make a mess
      4. Dynamically: determine what objects can flow to here
         a. We’ll get to a way of doing this later
   C. Function pointers
      a. Restrict statically
         i. Based on types and number of arguments
      b. Associate a set of functions with each function pointer
         i. Similar to constant propagation
            ➢ Except you track sets of constants for each temporary
         ii. Why might this be appropriate
Most uses of function pointers have a small set of routines that can be called

c. Initially each constant has a value of BOTTOM
d. When assigning a constant address, add it to the set
e. When assigning a computed address
   i. Either set the set to TOP
   ii. Or if it is a table lookup into a constant table ...

D. This can be done either flow-sensitive or flow-insensitive
   1. Flow sensitive :: see what values flow to what uses
   2. Flow insensitive :: accumulate over the whole routine ignoring flow
      a. Looking at each instruction once, not using flow analysis
      b. This is standard technique to get approximations for interprocedural analysis

E. What is the remaining problem
   1. What if the function pointer is passed into the routine
   2. Need to take all routines into account

F. There use a work list algorithm to find a fixed point
   1. Add a routine to the work list whenever the parameters that may be function pointers change
   2. Add a call site to the work list whenever a routine changes something

III. Inlining routines

A. We’ve already mentioned this
   1. What are you trying to gain
      a. Eliminating the call and return instruction (useful if tiny)
      b. Allow better optimization
         i. Use of getter-setter methods can be optimized
         ii. Bigger routines yield more chances for optimization
      c. Specialize the method based on incoming parameters
   2. Can be done at either AST level or IR level

B. When to do it
   1. If the routine is short
      a. Call time can be relatively significant
   2. If there are constant parameters
      a. That might affect control flow
      b. That might enable optimization of the routine body

C. Optimization with inlining
   1. Need to be intelligent about replacing parameters
2. Need to be intelligent about return value
3. Need to do dead code elimination
   a. Including branch prediction and unreachable blocks
4. Handling things like synchronized methods
5. Handling things like recursion, function pointers, virtual methods
D. Can optimize via specialization as well
   1. Suppose a routine has different control flow based on a parameter
      a. And you often can determine that parameter is a constant
      b. Generate two (or more) versions of the routine
         i. One for the general case
         ii. One for each constant case
      c. Change the calls where the parameters are known constants
IV. Interprocedural Data Flow Analysis
   A. Suppose we want to compute the effect of CALL instruction
      1. For modifies, live, kill, anticipated, etc.
   B. First approximation
      1. Do a flow-insensitive analysis for each routine
         a. Unioning the set of modifies for each non-call instruction
            i. Taking TOP into account
         b. This is a minimum set of items that are modified by the call
      2. Then you take each call and add to the sets all items from all the called
         routines
         a. This has to be done until a fixed point is reached
   3. Why is this a first approximation
      a. Doesn’t take parameters into account
      b. Doesn’t take aliasing into account
   4. Why flow insensitive
      a. Flow sensitive taking routines into account is NP-hard if done fully
      b. Flow insensitive generally gives enough information
      c. Compromise between the two is often used as well
   C. Parameter analysis
      1. Suppose the language supports call-by-reference
         a. Then you have to associate formal parameters and their temporaries
            with actual parameters and their temporaries
         b. Note that such calls might be reflected as pointer arithmetic in the IR
            i. But you can specialize these cases as they are more common than
               general pointers
2. If the passed values are temporaries
   a. Then it is sufficient to record whether the procedure modifies the corresponding temporary internally
   i. This is kept as associated information with the procedure
   b. The call then marks all such temporaries as written
   c. Note you can just assume these are written without doing analysis
   d. But what happens with parameters?
3. Here you build a binding graph
   a. Nodes are the formal parameters of all routines in the program
   b. Edges represent a binding of a callers formal parameters to a callees parameters
   i. If at some call site in p, there is a call of q that binds formal parameter x of p to formal parameter y of q, then there is an edge x -> y in the binding graph
   c. This is then used to provide a better approximation

D. Alias Analysis
1. Reference parameters are common but not the only problem
2. In general, you have to deal with arbitrary aliases
   a. How much depends on the language
   i. C/C++: arbitrary pointers (including pointer to local storage and registers)
   b. Aliases are generally created by
   i. Parameter passing
   ii. Access to non-local variables
   iii. Pointer arithmetic
3. Build a relations
   a. ALIAS: (var,routine) -> set of variables
   i. Can be done for a particular point in the program rather than by routine
   b. Also PTS(X) = set of items tags can point to
   c. Do this incrementally
   d. Similar to handling reference parameters, but generalized
   e. Note this is useful in general (inside a routine)
   i. Parameters might be aliases
   ii. Pointer arithmetic might create aliases
   iii. Needs to be integrated into the modifies relationship
   iv. Might be needed in the uses relationship
4. Methodology
   a. Build an the program’s binding graph that takes formal parameters into account
      i. Might need to augment this if routines can be nested
   b. Solve a forward data flow problem for each formal parameter fp the set A(fp) of variables v (not including formal parameters) that fp may be aliased to by a call chain that binds v to vp
   c. Do a forward data flow analysis to determine formal parameters that might be aliased to one another
   d. Combine the information

5. To handle pointers, more needs to be done
   a. Have abstract memory locations
      i. Essentially value numbers corresponding to memory
   b. Determine for each temporary the set of abstract memory locations
      i. That the temporary can overlap with at each program point
      ii. That the temporary can point to at each program point
   c. This gives alias information
   d. However, this needs to be generalized to be interprocedural
      i. Handle calls using flow-insensitive analysis for each routine
      ii. Using the union of all call sites