A classic tool: slicing
• Of interest by itself
• And for the underlying representations
  – Originally, data flow
  – Later, program dependence graphs

Slicing, dicing, chopping
• Program slicing is an approach to selecting semantically related statements from a program [Weiser]
  • In particular, a slice of a program with respect to a program point is a projection of the program that includes only the parts of the program that might affect the values of the variables used at that point
    – The slice consists of a set of statements that are usually not contiguous

Basic ideas
• If you need to perform a software engineering task, selecting a slice will reduce the size of the code base that you need to consider
  • Debugging was the first task considered
    – Weiser even performed some basic user studies
  • Claims have been made about how slicing might aid program understanding, maintenance, testing, differencing, specialization, reuse and merging

Example
```
read(n)
i := 1;
sum := 0;
product := 1;
while i <= n do begin
  sum := sum + i;
  product := product * i;
i := i + 1;
end;
write(sum);
write(product);
```

Weiser’s approach
• For Weiser, a slice was a reduced, executable program obtained by removing statements from a program
  – The new program had to share parts of the behavior of the original
• Weiser computed slices using a dataflow algorithm, given a program point (criterion)
  – Using data flow and control dependences, iteratively add sets of relevant statements until a fixpoint is reached
Ottenstein & Ottenstein

• Build a program dependence graph (PDG) representing a program
• Select node(s) that identify the slicing criterion
• The slice for that criterion is the reachable nodes in the PDG

Real PDGs are a bit more complicated

• Vertices in the graph represent (a) assignment states and (b) predicates in the program
• Edges represent control and data flow dependences
• Control dependences always start at a predicate (or the entry node)
  – They are labeled with a boolean
  – Intuitively, node $w$ is control dependent on node $v$ if the predicate of node $v$ evaluates to the label on the edge from $v$ to $w$ – that is, what happens at $w$ controls whether or not $v$ executes
  – An assignment statement followed immediately by another assignment statement have no control dependence between them, since the second one always executes when the first one does.
• Data dependences represent the possible flow of values through the program
  – (Roughly) there is a data dependence (edge) from node $v$ to node $w$ if $v$ includes an assignment to some variable $x$, and then $w$ includes a use of (that specific) $x$.
  – These can be separated into (at least) loop-independent and loop-carried dependences, which roughly distinguish whether the relationship is across iterations of a loop or not
• Def-order dependences can also be used; these aren’t needed for all analyses, but ensure that only equivalent programs have isomorphic PDGs.

Procedures

• What happens when you have procedures and still want to slice?
• Weiser extended his dataflow algorithm to interprocedural slicing
• The PDG approach also extends to procedures
  – But interprocedural PDGs are a bit hairy (Horwitz, Reps, Binkley used SDGs)
  – Representing conventional parameter passing is not straightforward

The next slide...

• Shows a fuzzy version of the SDG for a version of the product/sum program
  – Procedures Add and Multiply are defined
  – They are invoked to compute the sum, the product and to increment $i$ in the loop
Context

- A big issue in interprocedural slicing is whether context is considered.
- In Weiser’s algorithm, every call to a procedure could be considered as returning to any call site.
  - This in general significantly increases the size of a slice.

Reps et al.

- Reps and colleagues have a number of results for handling contextual information for slices.
- These algorithms generally work to respect the call-return structure of the original program.
  - This information is usually captured as summary edges for call nodes.
- www.cs.wisc.edu/~reps/talks/PLDI00.tutorial.ppt
  - General graph reachability for program analysis tutorial.

Chopping

- Given source S and target T, what program points transmit effects from S to T?
- Very roughly, intersect forward slice from S with backward slice from T.
- Dicing: “dynamic chopping.”

Technical issues

- How to slice in the face of unstructured control flow?
- Must slices be executable?
- What about slicing in the face of pointers?
- What about those pesky preprocessor statements?

Size of slices

- Most optimistic study [Binkley & Harmon 2003]:
  - A large-scale study of 43 C programs totaling just over 1 million lines of code.
  - Included the forward and backward static slice on every executable statement -- 2,353,598 slices constructed and analyzed.
  - Average slice size being just under 30% of the original program.
  - Ignoring calling-context led to a 50% increase in average slice size.

Dynamic slicing

- Conventional program slicing assumes nothing about the inputs.
- Dynamic slicing [Agrawal & Horgan 1990] [Korel & Laski 1990] is a variant that considers slicing with respect to a given test case (or suite) – increased precision for debugging is the intent.
Lackwit (O’Callahan & Jackson)

- Code-oriented tool that exploits type inference
- Answers queries about C programs
  - e.g., “locate all potential assignments to this field”
  - Accounts for aliasing, calls through function pointers, type casts
- Efficient
  - e.g., answers queries about a Linux kernel (157KLOC) in under 10 minutes on a PC

Lackwit

- Semantic
- Scalable
- Real language (C)
- Static
- Can work on incomplete programs
  - Make assumptions about missing code, or supply stubs

Lackwit analysis

- Approximate (may return false positives)
- Conservative (may not return false negatives) under some conditions
  - C’s type system has holes
  - Lackwit makes assumptions similar to those made by programmers (e.g., “no out-of-bounds memory accesses”)
  - Lackwit is unsound only for programs that don’t satisfy these assumptions

Query commonalities

- There are a huge number of names for storage locations
  - local and global variables; procedure parameters; for records, etc., the sub-components
- Values flow from location to location, which can be associated with many different names
- Archetypal query: Which other names identify locations to which a value could flow to or from a location with this given name?
  - Answers can be given textually or graphically

Sample queries

- Which integer variables contain file handles?
- Can pointer foo in function bar be passed to free()? If so, what paths in the call graph are involved?
- Field f of variable v has an incorrect value; where in the source might it have changed?
- Which functions modify the cur_vvh field of map_manager_global?

An example

- Query about the cur_vvh field of map_manager_global
- Shaded ovals are functions extracting fields from the global
- Unshaded ovals pass pointers to the structure but don’t manipulate it
- Edges between ovals are calls
- Rectangles are globals
- Edges to rectangles are variable accesses

Claim

- This graph shows which functions would have to be checked when changing the invariants of the current vehicle object
  - Requires semantics, since many of the relationships are induced by aliasing over pointers
Underlying technique

- Use type inference, allowing type information to be exploited to reduce information about values flowing to locations (and thus names)
- But what to do in programming languages without rich type systems?

Trivial example

- DollarAmt
  getSalary(EmployeeNum e)
- Relatively standard declaration
  - Allows us to determine that there is no way for the value of \( e \) to flow to the result of the function
  - Because they have different types
- int
getSalary(int e)
- Another, perhaps more common, way to declare the same function
  - This doesn’t allow the direct inference that \( e \)’s value doesn’t flow to the function return
  - Because they have the same type
- Demands type inference mechanism for precision

Lackwit’s type system

- Lackwit ignores the C type declarations
- Computes new types in a richer type system

- char*
  strcpy(char* dest, char* source)
- (num^ ref^, num^ ref^) \rightarrow num^ ref^
- Implies
  - Result may be aliased with dest (flow between pointers)
  - Values may flow between the characters of the parameters
  - No flow between source and dest arguments (no aliasing)

Incomplete type information

- void*
  return1st(void* x, void* y)
  {
    return x;
  }
- (a^ ref^, b) \rightarrow a^ ref^
- The type variable \( a \) indicates that the type of the contents of the pointer \( x \) is unconstrained
  - But it must be the same as the type of the contents of pointer \( y \)
- Increases the set of queries that Lackwit can answer with precision

Polymorphism

- char*
  ptr1;
  struct timeval* ptr2;
  char** ptr3;
  return1st(ptr1, ptr2); return1st(ptr2, ptr3)
- Both calls match the previous function declaration
- This is solved (basically) by giving return1st a richer type and instantiating it at every call site
  - (c^ ref^, d) \rightarrow c^ ref^
  - (b^ ref^, f) \rightarrow a^ ref^

Type stuff

- Modified form of Hindley-Milner algorithm “W”
- Efforts made to handle
  - Mutable types
  - Recursive types
  - Null pointers
  - Uninitialized data
  - Type casts
  - Declaration order
Morphin case study

- Robot control program of about 17KLOC
- Vehicle object contains two queue objects
  - Client was investigating combining these two queues into one
- Queried each queue object to discover operations performed and their contexts
- The two graphs each contained 171 nodes
  - But each graph had only five nodes highlighted as “accessor” nodes

Example

- These five matches helped identify code to be changed
- grep would have returned false matches and missed matches when parameters were passed to functions
- Context-sensitivity needed to distinguish the two queue objects
  - Because both are passed as arguments to the same queue functions

Recap

- Helps find relationships among variables in a C program
  - Exploits type inference to understand values flowing to locations and thus names
- Approximate, although safe under many (most?) conditions
- Reasonably efficient