# CSE584: Software Engineering

Lecture 9 (December 1, 1998)

David Notkin
Dept. of Computer Science & Engineering
University of Washington
www.cs.washington.edu/education/courses/584/CurrentQtr/

## This week

- Type inference (Lackwit)
- Program representations for tool support
- Inferring invariants



## 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



## Placement

- Lexical tools are very general, but are often imprecise because they have no knowledge of the underlying programming language
- Syntactic tools have some knowledge of the language, are harder to implement, but can give more precise answers
- Semantic tools have deeper knowledge of the language, but generally don't scale, don't work on real languages and are hard to implement



## Lackwit

- Semantic
- Scalable
- Real language (C)
- Static
- Can work on incomplete programs
  - Make assumptions about missing code, or supply stubs
- 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_veh` field of `map_manager_global`?



## 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

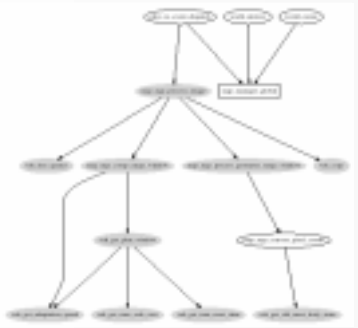
## Query commonalities

- 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



## An example

- Query about the `cur_veh` 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)`
- $(\mathbf{num}^{\alpha}\ \mathbf{ref}^{\beta}, \mathbf{num}^{\alpha}\ \mathbf{ref}^{\gamma}) \rightarrow^{\phi} \mathbf{num}^{\alpha}\ \mathbf{ref}^{\beta}$
- 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**$^{\beta}$, $b$) $\rightarrow^{\phi}$ $a$ **ref**$^{\beta}$
- 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**$^{\beta}$, $d$) $\rightarrow^{\delta}$ $c$ **ref**$^{\beta}$
  - ($e$ **ref**$^{\alpha}$, $f$) $\rightarrow^{\chi}$ $e$ **ref**$^{\alpha}$

$$\forall a.\forall \beta.\forall b.\forall \phi.(a\ \mathbf{ref}^{\beta}, b) \rightarrow^{\phi} a\ \mathbf{ref}^{\beta}$$



## 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



- `*from1` is not compatible with either `*from2` or `*to2`
  - But it is with `copy:*from, copy:*to, copy5:*from + copy5:*to`



## 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 onjects
  - 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
  - Although I didn't show the numbers, they are now better than reported in the ICSE paper



## 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);
```

```
read(n)
i := 1;

product := 1;
while i <= n do begin

    product := product * i;
    i := i + 1;
end;

write(product);
```

This example (and other material) due in part to Frank Tip



## Weiser's approach

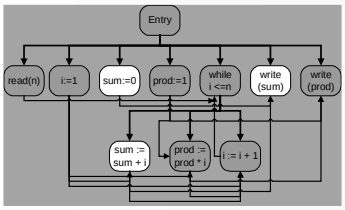
- For Weiser, a slice was a reduced, executable program obtained by removing statements from a program
  - The new program had to share parts of the behavior of the original
- Weiser computed slices using a dataflow algorithm, given a program point (criterion)
  - Using data flow and control dependences, 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

## PDG for the example




- Thick lines are control dependences
- Thin lines are (data) flow dependences



## Procedures

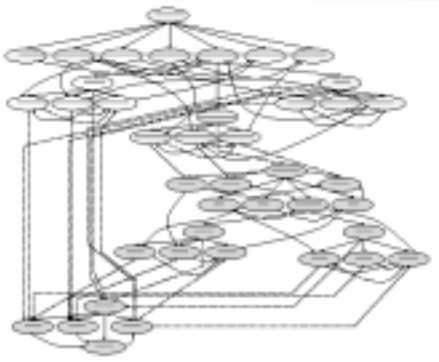
- What happens when you have procedures and still want to slice?
- Weiser extended his dataflow algorithm to interprocedural slicing
- The PDG approach also extends to procedures
  - But interprocedural PDGs are a bit hairy (Horwitz, Reps, Binkley used SDGs)
  - Representing conventional parameter passing 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
- This size issue is one of the ones that Griswold has addressed in a couple of ways







## 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 may significantly increase 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

## 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?



## Dynamic slicing

- These algorithms have all been static
  - They work for all possible inputs
- There is also work in dynamic slicing, trying to find slices that satisfy some execution streams over sets of inputs
  - Korel and Laski characterize dynamic slices in terms of a *trajectory* that captures the execution history of a program in terms of a sequence of statements and control predicates



## (Potential) applications

- Debugging
- Program differencing
  - Semantic versions of diff
- Program integration
  - Merging versions together
- Testing
  - Slicing can be used to define more rigorous testing criterion than a conventional data flow testing criterion



## Recap

- Cool idea
  - Dicing and chopping are beyond the scope of this lecture
- Difficult on practical programs
  - *May* be coming closer to feasible after almost 20 years of research
- Little data on the size of slices
- Will it be more than a cool idea?
  - My guess? No (but I wouldn't bet the farm on it)



## Invariants

- Invariants play a central role in program development
  - Refining a specification into a correct program
  - Static verification of limited (but important) invariants such as type declarations
  - Run-time checking of invariants represented as `assert` statements
- A number of researchers firmly believe that the lack of stated invariants in programs is the root of almost all evil



## Where are they?

- Invariants are few and far between in most code
  - In Gnu Emacs, 33 out of 114KLOC lines match a grep on `"assert"` (some of them are actually asserts)
    - This isn't fully fair, since comments may also represent asserts (and perhaps other program statements, too)
- Invariants, like comments, are probably omitted in part because of the opportunity cost

# In any case, they aren't there





# Our approach: dynamically infer them

- (Well, at least some of them)
- Execute a program on a collection of inputs
- Extract the values that variables take on during these executions
- Infer invariants at program points based on these values



# Complement to static

- This approach (if it works) is, like all dynamic techniques
  - unable to produce sound information about all program executions
  - heavily dependent on the actual inputs selected
- Static techniques are always intellectually more attractive
  - But they are often difficult and often imprecise in practice
- The approaches surely complement each other



# Ex: a basic Gries program

*i,s := 0,0;*
*do i ≠ n →*
*i,s := i+1,s+b[i]*
*od*

```
15.1.1:::BEGIN  100 samples
  N = size(B)
  N in [7..13]                (7 values)
  B                           (100 values)
    All elements >= -100      (200 values)

15.1.1:::END    100 samples
  N = I = N_orig = size(B)
  B = B_orig
  S = sum(B)
  N in [7..13]                (7 values)
  B                           (100 values)
    All elements >= -100      (200 values)
```

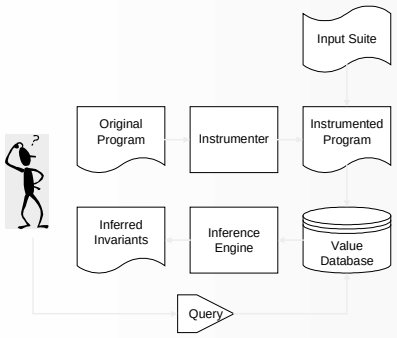


# And the loop invariants

```
15.1.1:::LOOP   1107 samples
  N = size(B)
  S = sum(B[0..I-1])
  N in [7..13]                      (7 values)
  B                                 (100 values)
    All elements in [-100..100]     (200 values)
  I in [0..13]                      (14 values)
  sum(B) in [-556..539]             (96 values)
  B[0] nonzero in [-99..96]         (79 values)
  B[-1] in [-88..99]                (80 values)
  B[0..I-1]                         (985 values)
    All elements in [-100..100]     (200 values)
  I <= N                            (77 values)
 Negative invariants:
  N != B[-1]                        (99 values)
  B[0] != B[-1]                     (100 values)
```



# Basic structure



## Instrumentation

- A mere matter of programming
  - Not!
  - Ask Jake!
- The C/C++ instrumenter is semi-automatic as now
  - Built using the EDG framework



## Captured data

- The instrumenter ensures that for each instrumented program point the values of all selected variables are written out
  - Locals, globals, parameters, return values can be selected
- Question: should variables with unchanged values be written out each time they are encountered?
  - Changes the number of samples and confidence in inferred invariants



## User control

- The user should be able to select
  - the program points to instrument and
  - the variables to instrument at those points
- This is important for two reasons
  - To manage the performance of the inference engine
  - To allow the user to focus on the parts of the program that are of interest for the given task



## Invariants

- Given the value database, the engine checks for a variety of invariants
- The list of invariants was developed by hypothesizing some basic invariants, studying the Gries programs to find generally useful invariants
  - Undoubtedly, this list will evolve



## Invariants

- For any variable, determine if it is a constant or takes on a small number of values
- For any numeric variable, determine if it is in a given range
  - Or is (or is not) equal to a mod b (where a and b are constants)
- For multiple numeric variables, look for linear relationships, standard library function relationships, comparisons, etc.



## Sequence invariants

- For a given sequence, determine if all its elements satisfy some invariant (for instance, are all less than some constant)
- For multiple sequences, check for subsequence and lexicographic relationships
- For a sequence and a scalar variable, check for membership

## Checking

- As each new value for a variable is read, check each possible invariant
  - Stop checking if an invariant fails to hold
- The listed invariants are cheap to check
  - But there are a lot of them!



## Statistics

- Statistical analysis is used to decide if properties are likely to be invariants as opposed to coincidental properties
  - Negative invariants: relationships that might be expected to occur but were never observed in the input
  - Inferring ends of ranges



## Negative invariant example

- Reported values for variable x fall into a range of size r that includes 0
- For a given value of x, the probability it is not 0 is 1-1/r (assuming uniform distribution)
- With v values, the probability x is never 0 is $(1-1/r)^v$
- If this is less than a user-defined confidence level, then the invariant $x \neq 0$ is reported



## Gries program, new input

```
15.1.1:::LOOP   986 samples
  N = size(B)
  S = sum(B[0..I-1])
  B                                (96 values)
    All elements in [-6005..7680]  (784 values)
  N in [0..35]                     (24 values)
  I >= 0                           (36 values)
  sum(B) in [-15006..21144]        (95 values)
  B[0..I-1]                        (887 values)
    All elements in [-6005..7680]  (784 values)
  I <= N                           (363 values)
```



## Derived variables

- To reduce the complexity of computing invariants, derived variables (actually, values) are used
- A derived variable might represent, for instance, the length of an array
- This allows a broader variety of invariants to be found without modifying the engine deeply each time



## Scenario

- Take a small sized C program (about 500 lines) and modify it to explore the use of dynamically computed invariants
- The program and thousands of test cases existed beforehand
- It does regular expression matching, but didn't provide the + operator

## Informal walkthrough…

- …of the change and the use of the invariants



## Discussion

- Mixed static analysis with the use of dynamic invariants
- Invariants provided a suitable basis for the programmer's own, more complex inferences
  - Since the invariants are in terms of source code entities, the programmer could do other analysis to better understand these issues
- Invariants acted as a succinct abstraction of a mass of supporting data



## Question

- Does anyone know of any product or approach in which information from test suite executions is stored and later queried?



## Scalability

- When this idea was first proposed, I thought that the biggest issue and problem would be performance
- Now that seems to be less of a problem (although still material)
  - Better performance of the engine that I expected (although memory is still a big issue)
  - Focused instrumentation and limited variable choice is the ultimate way to reduce costs
    - But will it hurt the utility of the approach?



## Possible uses

- Easing evolution
- Test case coverage
- Program understanding (especially with value database queries)
- Use to form a program spectrum
- Use to help insert assert statements in programs
- To aid compiler optimizations



## Other uses?

- And general comments?

## Recap

- Invariants are "good" but infrequently written down by programmers
- Instead, use execution information to infer (likely) program invariants
- But will this approach
  - Scale?
  - Actually help with any software engineering tasks?



## Tools

- Static vs. dynamic
  - Complementary
- Finding bugs vs. improving performance
- Program representations
  - Affect precision and performance
- Partial specifications
  - You get some benefit for small cost
- Inference to reduce programmer effort
  - Type, dynamic



## Next (and final) week

- Testing and quality assurance issues