Static Program Analysis / Instructor: Mooly Sagiv, Assistant: Noam Rinetzky
Lecture #1, 04-Mar-2004: Static Analysis Overview
Notes by: Yotam Shtossel

Overview

What is static analysis?

Static analysis is a special-purpose theorem prover, designed to prove useful facts about a program. As the name 'static' suggests, these facts hold on every execution of a program. It supplies much more than simple syntactic properties of the program, but is short of full program verification.

*an assumption (later to be discarded) – all of the code is known in advance.

Examples of static properties of special interest

Constant propagation - A variable in the program which is constant on each run, and thus can be 'discarded' as such by the compiler. The detection of constant expressions can improve the effectiveness of other compiler optimizations. For example, the compiler can detect loops which are executed constant number of times and unroll the loop body, thereby, removing costly branches.

```
int p(int x) {   return (x *x)  ; }
void main()
{
  int z;
  if  (getc())
     z = p(3) + 1;          // ➔ Z=10
    else z = p(-2) + 6;     // ➔ Z=10
  printf (z);               // ➔ printf(10);
}
```

The above program can be replaced by a program which performs:

```
getc();
printf(10);
```

to produce the same results with less time/space consumption.

A **virtual function call**, which in fact is a call to a member function of a single concrete class, i.e. the method's **address is unique** in some specific application. Notice that the address of this function may not be unique when this class is used by other clients.

Thus, the compiler (or at least the linker) can benefit from the fact that all the code is available.

Finding 'Live variables':

A variable is **live** at a program location

- If its R-value can be used before set.
- There exists a definition-free execution path from the label to a use of x.

An example:

```
{
    /* c  (it is used before being assigned to) */
    L0:  a := 0
    /* ac (…) */
    L1: b := a + 1
    /* bc */
        c := c + b
    /* bc */
        a := b * 2
    /* ac – 'a' is live because we can branch to L1, and this is not known in static time */
        if c < N goto L1
    /* c  */
        return c
}
```

It is easy to see from the example that 'a' and 'b' are never live together, and thus a compiler may **allocate** the same **register** to them. Also, the 'live variables' information can be useful for **Garbage Collection** (Garbage Collection (GC) usually scans the heap starting for the stack. It is possible to allow GC to reclaim more space by only scanning from live variable. In other words, the content of dead variables is not relevant and thus the GC can ignore pointer access paths starting from dead variables). Once again, this optimization is effective when the compiler detects small set of live variables)

Memory Leakage:

In the following program, a logic error caused the *"head = n;"* and *"rev = head;"* lines to be replaced. This causes the reversed list to be half of the original list, with the other half leaked. A static analysis would alert to the leakage of the *'head'* pointer (since it is assigned to without being stored anywhere) and thus the bug can be prevented.

```
List* reverse(List *head)
{
   List *rev, *n;
   rev = NULL;
   while (head != NULL)
```

```
    {
        n = head →next;
        head → next = rev;
        head = n; //error – advanced the head another time without "saving it" – memory leak
        rev = head;
    }
    return rev;
}
```

Reaching definitions:

Reaching definitions determine for each basic block B and each program variable *x* what statement of the program could be the last statement defining *x* along some path from the start node to B.

A statement *defines x* if it assigns a value to *x*, for example, an assignment or read of *x*, or a procedure call passed *x* (not by value) or a procedure call that can access *x*.

Formally, we say that a definition, *def* - def: x := , *reaches a point P* in the program if there is some path from the start node to *p* along which *def* occurs and, subsequent to *def*, there are no other definitions of *x*. Figure 1 shows this pictorially.

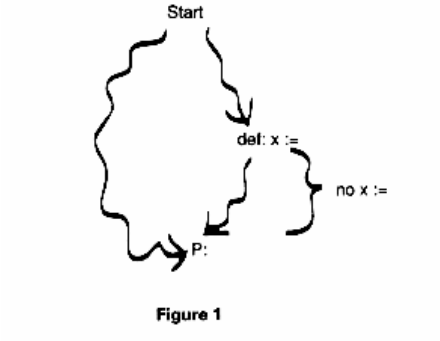


Figure 1

These are part of the Dataflow analysis (a summary of the creation/destruction of values in a program) which is used to identify legal optimization opportunities.

Unreachable Code:

Code that cannot be reached due to false conditions or methods which are never invoked. This may indicate a bug in the original code and thus reported by some compilers. It can also happen for code which is automatically generated by code generators (e.g., Lexx, Yacc, and Bison generate unreachable code due to the fact that the code generation phase is local). Unreachable code can also result from other optimizations, e.g., constant propagation may detect that a branch is always taken and thus the else branch becomes unreachable.

Dead code:

Dead code is code which cannot affect the program results. For example, assignments to dead variables are dead. Dead code usually results from other optimizations.

Integer intervals:

Because the program's input changes every time it is run, we have to approximate the values of program variables (e.g., x, y, and the result of 'x < y') in a way that covers all executions of the code. One common way to represent the values that an integer variable (in the user program) could take is via intervals. For instance, in some simple user code like this:

```
{
    y = 9;
    if (DynamicSomething())
        x = 3;
    else
        x = 5;
    //static analysis: x = [3,5], y = [9,9]
    if (y > x)
        printf("Gotcha!");
}
```

Integer intervals tell us that "Gotcha" will always be printed.

Other Examples:

- Pointer variables never point into the same location (C makes this very hard to use).
- Points in the program in which it is safe to free an object.
- Statements that can be executed in parallel – 'modern' CPUs count heavily on parallelism (e.g. Pentium's U & V execution pipelines).
- An access to a variable which must be in cache – can't go to matrices multiplication without it…

Usage in compilers

Compilers have taken a generic scheme, which is the product of the experience of many language development projects over a period of decades. The source-program passes the building blocks of Scanner→Parser→Semantic Analysis→Code Generator→Static Analysis→Transformation (notice that the Static Analysis is performed after the automatic Code Generation so that compiler-made local descisions can be improved by global analysis

In recent years, focus of intellectual research has shifted to Program Analysis, for two main reasons:

1. Programming languages are moving to higher level of abstraction (functional, OO, garbage collection, concurrent).
2. Advanced Computer Architectures (pipeline, multiple functional units, Very Large Instruction Word - VLIW, prefetching) are more difficult to program for. If Moore's law predicts computing power to double every 18 months, Proebsting's law predicts compiler optimizations to do that same every 18 years… ☹

Bottom line – the gap between the program that a programmer writes and the code that actually runs on a machine is rapidly growing, and compiler optimizations are there to bridge this gap, particularly in the area of program optimization. The optimization must not change the meaning of the program, and to establish such facts is the domain of program analysis.

Our other clients

Software Productivity Tools

Compile time debugging

- Stronger type Checking for C
- Array bound violations
- Identify dangling pointers
- Generate test cases
- No runtime exceptions
- Memory leaks
- Uninitialized variables detection
- Prove pre- and post- conditions

while the goal is full program verification which is well beyond the capability of any known method.

Competition in the Computing Industry

A significant portion of the improvement in workstation performance is due not to the improvements in its underlying hardware but instead to improvements in compiler optimization and the program analysis on which optimization relies.

Static analysis a.k.a. Abstract Interpretation

Static analysis can be viewed as interpreting the program over an "abstract domain" (Cousot and Cousot 1977).

- Our program will always end
- Types are irrelevant

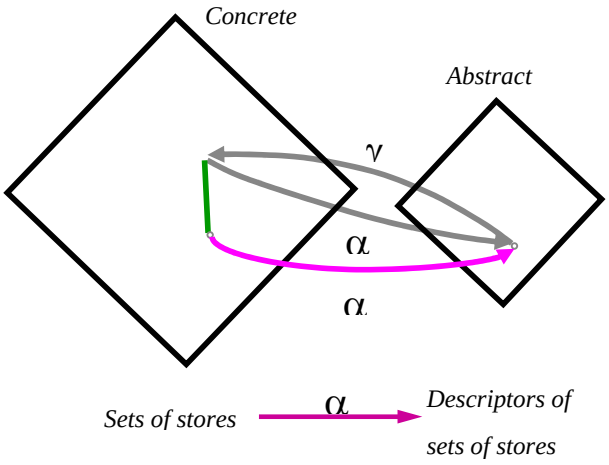
It is like executing the program over a larger set of execution paths (e.g. a Boolean will be considered to be false **and** true at the same time unless otherwise proved).

Abstract Interpretation is conservative – it guarantees sound results but can also be very imprecise. Its output for the constant propagation problem could be:

- Every identified constant is indeed a constant (sound).
- But not every constant is identified as such (not complete).

Formally: A programming language has a standard interpretation μ mapping programs e to values μ(e). An abstract interpretation consists of another interpretation α that maps programs e to abstract values α(e). The abstract values are abstractions of the standard values and this is formalized by a function γ that maps every abstract value δ to the set of (standard) values γ(δ) that δ represents. If a (standard) value ν is in γ(δ), then δ is said to be an abstraction of ν.

Visually:



Abstract Interpretation examples

Casting out nines:

The concrete domain is integers and the abstract domain would be integers mod 9. The latter can be calculated by summing the digits (recursively) for an intermediate result which exceeds 8.

"123 * 457 + 76543 = 132654?"

– Left: 123*457 + 76543 ➔ 6 * 7 + 7 = 6 + 7 ➔ 4

- Right: 21 → 3
- Report an error since the values don't match

Soundness of the comparison between the expressions using the integers and the expressions using the integers mod 9 is guaranteed because of the following mathematic rules:

(10a + b) mod 9 = (a + b) mod 9
(a+b) mod 9 = (a mod 9) + (b mod 9)
(a*b) mod 9 = (a mod 9) * (b mod 9)

Positive/Negative:

- The concrete language is built from integers, multiplication and addition. We'll define an abstract semantics which computes only the sign of the result.
- Abstraction α : Exp → {'+','-','0'}

  α(i) =

  if (i > 0) return '+'

  else if (i < 0) return '-'

  else return '0' // i == 0
- Concretization γ: {'+','-','0'} → 2^{Int}

  γ(+) = { i | i > 0}

  γ(-) = { i | i < 0}

  γ(0) = {0}
- The abstract semantics arithmetic will be defined as follows
  - Multiplication

Mul	+	0	-
+	+	0	-
0	0	0	0
-	-	0	+

  - Addition

Add	+	0	-
+	+	+	?
0	+	0	-
-	?	-	-

- How can this abstract interpretation assist us?

{

int x = rand() – RAND_MAX/2; //x can be positive or negative

y = x*x + 2; //y is positive here

```
    if (y<0)
    {
        //Dead code
    }
}
```

Asserting that y is positive, in the upper example, enabled us to remove dead code from the program.

Odd/Even:

- Determine if an integer variable is even or odd at a given program point.
- Abstraction $\alpha(X)$ =

  if $X = \varnothing$ return $\perp$

  else if for all z in X (z%2 == 0) return E

  else if for all z in X (z%2 == 0) return O

  else return ?
- Concretization $\gamma(a)$ =

  if $a = \perp$ return $\varnothing$

  else if a = E return Even

  else if a= O return Odd

  else return Natural
- e.g. $\alpha(\{-2,0\}) = E$

  $\gamma(E) = \{x \mid x \in Even\}$

  $\gamma(\alpha(\{-2,1,5\})) = \gamma(?)$ = All concrete states ➔ loss of information
- Example code:

```
{
    /*x = ? */
    while (x != 1) do { /* x = ? */
        if (x % 2 == 0)
/* x = E */        { x = x/2; }        /* x = ? */
        else
/* x = O */        { x = 3*x + 1; } /* x = E */
    }
}
```

- How do we know to tell that X will turn from odd to even? This is by performing abstract interpretation and then using the abstract domain's "arithmetic" which states that Odd*Odd + Odd = Even (i.e. 3*x + 1 ➔ Even when x is odd). While the arithmetic 'rules'

over the concrete domain (integers) are infinite, they are finite and small for the abstract domain (we have 3 states – '?', 'O' and 'E' and two operators – '+' and '*' on them) and can be stored in a small table.

Soundness of Abstract Interpretation

Let us denote SETin and SETout as the sets of concrete states before and after a statement S respectively. The abstract semantics performed for S on α(SETin) produces a descriptor of a set of states which contains SETout, $\gamma(AbsSem\_S(\alpha(SETin))) \supseteq SETout$, i.e. the abstract semantics is also 'true' in the operational semantics and this is the basis for the soundness.

For the same reason, the abstract semantics can't achieve better knowledge than the operational semantics, and this can be crucial for the (im)precision of the analysis. Using the Odd/Even domain again,

- SETin = {16, 32} ➔ α(SETin) = E
- The statement is x = x/2
- SETout = {8, 16}, by simple operational semantics
- ➔ α(SETout) = **E**
- This result could not be achieved by the abstract semantics since it is not true that every Even divided by 2 produces an Even again, and thus the analysis would have given us **'?'** in this case. This is of course **sound** but 'E' ⊂ '?' and our analysis is clearly **incomplete**.

Another example for the loss of information, by the abstraction of integer operations to the Positive/Negative domain:

- SETin = {X=1, Y=-2} ➔ α(SETin): {α(X) = '+', α(Y) = '-'}
- The statement is Y = X + Y
- SETout = {X=1, Y=1}, by operational semantics (addition over integers)
- ➔ α(SETout) = {α(X) = '+', α(Y) = **'-'**}
- However, by first performing abstraction of SETin and then using the abstract semantics we'd get
  - γ(AbsSem_S(α(SETin))) ➔ {X = '+', Y = **'?'**}
- And that is because the sign of the sum of two unknown integers, one positive and the other negative, is not known!

Some static optimizations can cause violations of the soundness:

- Loop invariant code motion – if the moved code was capable of causing an exception, it will be raised in a different context…
- Dead code elimination - Remove run time errors and infinite loops)
- Overflow ((x+y)+z) != (x + (y+z)) because of floating point precision

For many uses, the client of the analysis would allow this sort of violations to occur.

Quality checking tools may decide to ignore certain kinds of error. The claim 'sound' becomes with respect to different concrete semantics.

Challenges in Abstract Interpretation

- Finding appropriate program semantics (runtime)
- Designing abstract representations
  - What to forget
  - What to remember and how to keep it short…
  - Handling loops
  - Handling procedures
- Scalability
  - Large programs
  - Missing source code
- Precise enough

Runtime vs. Abstract Interpretation (software quality tools)

	Runtime (e.g. Purify)	Abstract
Effectiveness	Missed Errors (input and timing dependant)	False alarms (the above imprecision)
		Locate rare errors (analyses all execution paths)
Cost	Proportional to program's execution	Proportional to program's size

Undecidability

- Issues:
  - It is undecidable if a program point is reachable in some execution – If we could, we'd ask for a point just after the program's end and thus solve the program stop problem…
  - Some static analysis problems are undecidable even if the program conditions are ignored – examples can be found in open math problems etc.
- The Constant Propagation Example:

```
while (getc()) {
```

```
    if (getc()) x_1 = x_1 + 1;
        if (getc()) x_2 = x_2 + 1;
        ...
    if  (getc()) x_n = x_n + 1;
    }
  y = truncate (1/ (1 + p2(x_1, x_2, ..., x_n))
  /* Is y=0 here? */  ➔ Static analysis doesn't have the answer…
```

- Coping with undecidability
  - Loop free programs

    (-) A normal program without loops? Come on…

    (+) Useful for certain program, such as the code that runs in a hardware chip which consists of 'IF' clauses only and no loops (FPGA/Altera?) and which is very important to debug.
  - Simple static properties
  - Interactive solution
    - The user will help solve ambiguities by answering queries
    - Problem – the user will probably not know what hit him on such queries
  - Conservative estimations (aspire to be "good enough")
    - Every enabled transformation cannot change the meaning of the code but some transformations are not enabled
    - Non optimal code
    - Every potential error is caught but some “false alarms” may be issued

Precision/Optimality issues

As mentioned before, the soundness obtained by being conservative comes at the price of precision.

Analogies with Numerical Analysis:

- Approximate the exact semantics
- More precision can be obtained at greater computational costs
  - But sometimes more precise can also be more efficient (more knowledge can be used later in the analysis to resolve ambiguities, such as an IF clause, and thus lead to examination of less execution path)
  - Abstract interpretation doesn't assist us in evaluating speed versus precision of a program analysis

Optimality Criteria:

- Precise (with respect to a subset of the programs)
- Precise under the assumption that all paths are executable (statically exact)

- Relatively optimal with respect to the chosen abstract domain
- Good enough (e.g. wasting registers in a machine that has plenty of them)

Static Analysis vs. Program Verification

Program verification:

- Mathematically prove the correctness of the program
- Requires formal specification (what is Word supposed to do?!)
- Example Hoare Logic {P} S {Q}
  - {x = 1} x++ ; {x = 2}
  - {x=1}
    {true} if (y >0) x = 1 else x = 2 {?}
  - {y=n} z = 1 while (y>0) {z = z * y-- ; } {?}
- Not feasible for a complex software (Word again)

Program Analysis	Program Verification
• Fully automatic • But can benefit from specification • Applicable to a programming language • Can be very imprecise • May yield false alarms • Identify interesting bugs • Establish non-trivial properties using effective	• Requires specification and loop invariants • Not decidable • Program specific • Relative complete • Must provide counter examples • Provide useful documentation

Origins of Abstract Interpretation

- [Naur 1965] The Gier Algol compiler "`A process which combines the operators and operands of the source text in the manner in which an actual evaluation would have to do it, but which operates on descriptions of the operands, not their value"
- [Reynolds 1969] Interesting analysis which includes infinite domains (context free grammars)
- [Syntzoff 1972] Well foudedness of programs and termination
- [Cousot and Cousot 1976,77,79] The general theory
- [Kamm and Ullman, Kildall 1977] Algorithmic foundations
- [Tarjan 1981] Reductions to semi-ring problems
- [Sharir and Pnueli 1981] Foundation of the interprocedural case
- [Allen, Kennedy, Cock, Jones, Muchnick and Scwartz]

Some Industrial Success Stories

Array bound checks for IBM PL.8 Compiler:

- out-of-bounds references can be critical
  - can be a result of bugs
  - can be a result of a malicious behavior (buffer overflow attacks etc.)
- Avoiding these errors is easy if we bound-check each reference by adding the check in the compiler level (as 1970s compilers could do)
- The problem is that code with checking runs slowly – obvious opportunity for optimization
- PL.8 philosophy: check everything & optimize checks
  - According to the compiler developers – the checked code runs 5 to 10 percent slower than the unchecked code
  - On upper/lower bound fault, a trap function will be called without buffer overrun taking place
  - The check block added is considered as an atomic operation
  - Optimizations include
    - moving the check block out of loops – check the endpoints
    - with known loop bounds, the checks become static (for i=1 to 100 a[i] = … ➔ Check in compile time)

Polyspace Technologies:

- Focuses on Run-time errors in embedded applications, using abstract interpretation
- Errors which are said to be found by their C++ developer edition
  - Read access to non-initialized data (variables and function return values),

  - De-referencing through null and out-of-bounds pointers,
  - Out-of-bounds array access,
  - Invalid arithmetic operations such as division by zero, sqrt(negative number),
  - Overflow / underflow on arithmetic operations for integers and floating point numbers,
  - llegal type conversions, e.g.: long to short, float to int,
  - Access conflicts for data shared between tasks,
  - Invalid dynamic_cast calls,
  - Throws of unauthorized exceptions,
  - Calls to virtual pure methods,
  - Negative size arrays,
  - Null receivers,
  - Null pointers to members,
  - Wrong type for receivers such as polymorphic type in non-virtual method,
  - Throws during catch parameter construction,
  - Non-terminating function calls and loops,
  - Unreachable code (dead code).
- (interesting) Published Clients
  - Railway company – Verified a signaling program for trains
    - 15,000 lines of code in Ada verified in a few hours
    - Requirement – zero defects (trains at 300 kp/h separated by 3 minutes from one another…)
    - Debugged:
      - Access to non-initialized variables
      - Division by zero
      - Out of bounds array access
  - EADS (European Aeronotic Defence and Space company) Launch vehicle
    - 100,000 lines of code, Ada again
    - Used as a "final touch" tool
    - Very concerned to errors leading to a software halt (which means the missile is on its own…)
    - "Reminder": The EADS Ariane 5 rocket crashed in 1996 (valued with cargo at $500 million) after a 64-bit float was converted to a 16-bit signed integer and overflowed, thus failing to convert…
  - Safety-critical units in oil refineries and chemical/petrochemical plants.
    - 70,000 lines of C and 140,000 lines of Ada in "hard real time"
    - Standard verification took up to 5 man-years.
  - Nuclear Energy Industry – Nuclear safety-software

AbsInt:

- PAG - The Program Analyzer Generator, which implements:
  - Cache behavior prediction
  - Stack usage analysis
  - Memory error detection (dereferencing null pointers)
  - Post pass code optimizations
  - Conditional constant propagation
  - Addressing mode optimization
  - Data dependency analysis on pointer structures
  - Optimizations for a video accelerator multiprocessor
  - Pipeline analysis
  - Escape analysis on Java
  - Value analysis on executables
  - Shape analysis
  - Strongly live variables analyses
  - Interval analysis
  - Classical bit-vector analyses
- Timing validation for real-time software
  - Verify that safety-critical applications always react fast enough
  - Avoids the need to perform Extensive timing testing
- Code Compaction - The size of compiled C code is becoming increasingly critical in embedded systems
  - Speed up the execution of the compiled code
  - Reduce size of program ➔ lower memory and hardware costs
- Stack usage analysis which is valid for all inputs and each task execution

'Prefix' by Intrinsa:

- Was used as early as 1997 by Sun Microsystems to aid development of hardware diagnostics tools for users of Sun's Solaris operating system, and by Netscape.
- Bought by Microsoft in 1999 in order to control bugs in Windows 2000. Although the tool found many bugs, it proved to be slow and processor-intensive. In response, Microsoft has developed an improved version (and has developed offspring tools like Prefast and SLAM).
- Aims to achieve a Purify 'look and feel' in a static analyzer.
- Results of running the tool on Mozilla web browser, Apache web server and a BoundsChecker demo program (GDI) using a P2 266Mhz with 96Mb RAM:

Table I. Performance on sample public domain software.

Program	Language	Number of files	Number of lines	PREfix parse time	PREfix simulation time
Mozilla	C++	603	540 613	2 h 28 min	8 h 27 min
Apache	C	69	48 393	6 min	9 min
GDI Demo	C	9	2655	1 s	15 s

Table II. Warnings reported in sample public domain software.

Warning	Mozilla	Apache	GDI
Using uninitialized memory	26.14%	45%	69%
Dereferencing uninitialized pointer	1.73%	0	0
Dereferencing NULL pointer	58.93%	50%	15%
Dereferencing invalid pointer	0	5%	0
Dereferencing pointer to freed memory	1.98%	0	0
Leaking memory	9.75%	0	0
Leaking a resource (such as a file)	0.09%	0	8%
Returning pointer to local stack variable	0.52%	0	0
Returning pointer to freed memory	0.09%	0	0
Resource in invalid state	0	0	8%
Illegal value passed to function	0.43%	0	0
Divide by zero	0.35%	0	0
Total number of warnings	1159	20	13

- Examination of the errors reported found that many of them are not on the mainstream code, which is usually tested and debugged more, and thus is complimentary to 'standard' testing.
- The parse time of the various applications was not much longer than their respective compile times.

Some Academic Success Stories

- Cousot PLDI 03
  - Validates floating point computations
    - High precision rate
    - Reasonable computational power and time
    - Main effort was to discover an appropriate abstraction
    - Uses 3 abstract domains – two specialized and one improved
- CSSV (Nurit Dor) PLDI 03 – C String Static Verifier
  - Prove the absence of buffer overruns by looking for the errors:
    - ANSI-C violations of string, such as an access out of bounds
    - Violations of pre/post-conditions of procedures
    - "Cleanness" – all accesses are before the null-termination byte etc.
  - Tested on actual EADS Airbus code with no bugs found
  - Scored a good bug/false alarm ratio on another commonly used string intensive application
- PLDI 02 Ramalingam et al., PLDI 04 Yahav & Ramalingam

- Conformance of client to component specifications

Complementary Approaches

- Finite state model checking
  - Limit integers to be 2^{32} (machine dependant)
- Unsound approaches
  - Compute underapproximation
- Better programming language design
  - Java/C# vs. C++
- Type checking
- Proof carrying code
  - Add assertions to the program, both for previously verified code and unverified code
- Just in time and dynamic compilation
  - More knowledge of the program is given – can achieve better analysis and faster
  - Usually ran in situations where the user is waiting for a response (e.g. JAVA programs running on client side in web browsing), and he doesn't like to wait!
- Profiling
- Runtime tests