Static Analysis with Soot

SMA HS 2018
Manuel Leuenberger
PhD student at SCG
leuenberger@inf.unibe.ch
Soot is a static analysis framework

• originally an optimization framework (used in compilers)
• understands JVM languages (Java, Android, etc.)
• whole-program analysis (call graph construction)
• dataflow analysis (nullness, array boundary checks)
...but first some theory
Dataflow analysis

- dataflow analysis is a form of abstract interpretation, i.e. reason about some properties of the program state at a certain block

- different types:
  - forward (reaching definitions)
  - backward (liveness)
  - branched (nullness)
Reaching definitions

```c
int gcd(int a, int b) {
    int c = a;
    int d = b;
    if (c == 0) {
        return d;
    }
    while (d != 0) {
        if (c > d) {
            c = c - d;
        } else {
            d = d - c;
        }
    }
    return c;
}
```

which definitions reach here?
Reaching definitions

• Which definitions reach a block?

• used in compiler optimizations
  • constant folding
  • common subexpression elimination
  • use-def/def-use chains
Potential optimizations

```c
int gcd(int a, int b) {
    int c = a;
    int d = b;
    if (c == 0) {
        return d;
    }
    while (d != 0) {
        if (c > d) {
            c = c - d;
        } else {
            d = d - c;
        }
    }
    return c;
}
```

$c = a$, parameter $a$ not changed  $\rightarrow a == 0$
Potential optimizations

```c
int gcd(int a, int b) {
    int c = a;
    int d = b;
    if (a == 0) {
        return d;
    }
    while (d != 0) {
        if (c > d) {
            c = c - d;
        } else {
            d = d - c;
        }
    }
    return c;
}
```
Potential optimizations

```c
int gcd(int a, int b) {
    int c = a;
    int d = b;
    if (a == 0) {
        return b;
    }
    while (d != 0) {
        if (c > d) {
            c = c - d;
        } else {
            d = d - c;
        }
    }
    return c;
}
```

c, d not used, a, b unchanged → allocate later
Potential optimizations

```c
int gcd(int a, int b) {
    if (a == 0) {
        return b;
    }
    int c = a;
    int d = b;
    while (d != 0) {
        if (c > d) {
            c = c - d;
        } else {
            d = d - c;
        }
    }
    return c;
}
```

a, b only used in definition → reuse registers
Potential optimizations

```c
int gcd(int a, int b) {
    if (a == 0) {
        return b;
    }
    while (b != 0) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```
Control-flow graphs

```c
int gcd(int a, int b) {
    int c = a;
    int d = b;
    if (c == 0) {
        return d;
    }
    while (d != 0) {
        if (c > d) {
            c = c - d;
        } else {
            d = d - c;
        }
    }
    return c;
}
```
int gcd(int a, int b)

{ c = a; 
d = b;

if (c == 0)
    return d;

while (d != 0)
{	if (c > d)
            return c;
        
c = c - d;
        d = d - c;
}

return d;
}
int gcd(int a, int b)
{
c = a

d = b

if (c == 0)
    return d

while (d != 0)
{
    if (c > d)
        return c
    c = c - d
    d = d - c
}

return c;
}
int gcd(int a, int b) {
    int c = a, d = b;
    if (c == 0) return d;
    while (d != 0) {
        if (c > d) return c;
        c = c - d;
        d = d - c;
    }
    return c;
}
int gcd(int a, int b)
{
    int c = a, d = b;
    if (c == 0)
        return d;
    while (d != 0)
    {
        if (c > d)
            return c;
        c = c - d;
        d = d - c;
    }
    return c;
}
How to merge?

```c
int gcd(int a, int b)
{
    int c = a, d = b;
    while (d != 0)
    {
        if (c >= d)
            return c;
        c = c - d;
        d = d - c;
    }
    return d;
}
```
```
int gcd(int a, int b)
{
    int c = a, d = b;
    if (c == 0)
        return d;
    while (d != 0)
    {
        if (c > d)
            return c;
        c = c - d;
        d = d - c;
    }
    return c;
}
```
int gcd(int a, int b)

c = a

d = b

if (c == 0)
    return d

while (d != 0)
    if (c > d)
        return c
    c = c - d
    d = d - c

return d
\begin{align*}
\text{REACH}_{\text{in}}[S] &= \bigcup_{p \in \text{pred}[S]} \text{REACH}_{\text{out}}[p] \\
c &= c - d \\
\text{REACH}_{\text{out}}[S] &= \text{GEN}[S] \cup (\text{REACH}_{\text{in}}[S] - \text{KILL}[S])
\end{align*}

Dataflow equation
Dataflow equation solver

• iterative round-robin

• start with initial (empty) set for each block input

• compute output of each block whenever its input (=output of predecessors) change

• repeat until no outputs change

```c
int gcd(int a, int b) {
    int c = a, d = b;
    while (c != 0) {
        if (c > d) return c;
        c = c - d;
        d = d - c;
    }
    return d;
}
```
Ensure convergence

• a block output needs to reach a fix-point...

• ...or the solver never terminates

• “unify towards top of lattice”
\[
\text{REACH}_{\text{in}}[S] = \bigcup_{p \in \text{pred}[S]} \text{REACH}_{\text{out}}[p]
\]

lattice for each element in REACH, e.g., \(d = d - c\):

- lattice top: “reaches block”
- lattice bottom: “does not reach block”

\[
\begin{align*}
\{ & \} \cup \{ & \} = \{ & \}
\end{align*}
\]

REACH(out, p1) \hspace{1cm} REACH(out, p2) \hspace{1cm} REACH(in, S)
Enough theory! Where is the code?
Soon...
Java is complex

c = a > 5 ? b : d;
int gcd(int a, int b) {
    int c = a;
    int d = b;
    if (c == 0) {
        return d;
    }
    while (d != 0) {
        if (c > d) {
            c = c - d;
        } else {
            d = d - c;
        }
    }
    return c;
}
int gcd(int, int) {
    ch.unibe.scg.sma.soot.TestClass this;
    int a, b, c, d;
    this := @this: ch.unibe.scg.sma.soot.TestClass;
    a := @parameter0: int;
    b := @parameter1: int;
    c = a;
    d = b;
    if a != 0 goto label3;
    return b;
    label1:
    if c <= d goto label2;
    c = c - d;
    goto label3;
    label2:
    d = d - c;
    label3:
    if d != 0 goto label1;
    return c;
}
A simple IR is crucial!

- small instruction set, i.e. just a few statement types
- statements do one thing only
- flat structure
- one scope per method, no nesting
- many special cases in Java are common cases in Jimple
int fib(int)
{
    X this;
    int n, $i0, $i1, $i2, $i3, $i4;
    this := @this: TestClass;
    n := @parameter0: int;
    if n > 1 goto label1;
    return n;

label1:
    $i0 = n - 2;
    $i1 = virtualinvoke this.<X: int fib(int)>($i0);
    $i2 = n - 1;
    $i3 = virtualinvoke this.<X: int fib(int)>($i2);
    $i4 = $i1 + $i3;
    return $i4;
}
Reaching Definitions

• intra-procedural: analyze each method independently

• only look at definitions of locals

• forward flow: knowledge flows along control-flow

\[
REACH_{\text{in}}[S] = \bigcup_{p \in \text{pred}[S]} REACH_{\text{out}}[p]
\]

\[
REACH_{\text{out}}[S] = GEN[S] \cup (REACH_{\text{in}}[S] - KILL[S])
\]
Code!
Other applications

• Nullness analysis
  • Which values can be null?
  • What is checked for null?

• Protocols
  • How are objects initialized?
  • Which methods do I need to call before doing X?

• Inter-procedural analysis
  • Sinks and sources: Can confidential information leak?