Static Analysis of Concurrent Programs with Goblint

Kalmer Apinis, Vesal Vojdani
Concurrency and parallelism

• What is the difference?
Concurrency and parallelism

• What is the difference?

• Concurrency: Any task *can* make progress.
• Parallelism: Tasks execute simultaneously.

• Any parallel computation is concurrent, but . . .
• Typically, slightly different concerns: responsiveness versus speed-up.
Models of concurrency

• What are the different ways to structure concurrent/parallel computations?
  • Task decomposition.
  • Communication and co-ordination
Models of concurrency

- What are the different ways to structure concurrent/parallel computations?
  - Task decomposition.
  - Communication and co-ordination

- Interruptible tasks sharing memory (Linux)
- Tasks sending asynchronous messages (Actors)
- Data-parallelism on “shared” memory (CUDA)
- ...
Concurrency Issues

• What can possibly go wrong?
Concurrency Issues

- What can possibly go wrong?
- Data races.
- Deadlocks.
- Starvation.
• Time of Check to Time of Use

• Running as root, so we check requests:
• May user access “innocent.txt”? 
• If yes, read file “innocent.txt”.
TOCTOU

- Time of Check to Time of Use
- Running as root, so we check requests:
- May user access “innocent.txt”? 
- Concurrent attack:
  \[\text{ln } -s \text{ passwd.txt innocent.txt.}\]
- If yes, read file “innocent.txt”.
Outline

1. Data Flow Analysis
2. Lockset Algorithm
3. The Goblint
Outline

1. Data Flow Analysis
2. Lockset Algorithm
3. The Goblint
Example

```
int g = 3;

int fun2(int y){
g = 9;
return y/2;
}
```

```
int fun1(){
    int x = 7;
    if (g<5){
        x++;
        return fun2(x);
    }
}
```

- Goal: find out ranges of variables
### Example

<table>
<thead>
<tr>
<th>g = 3</th>
<th>fun1()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>int x = 7;</td>
</tr>
<tr>
<td>fun2(int y){</td>
<td>if (g&lt;5){</td>
</tr>
<tr>
<td>g = 9;</td>
<td>x++;</td>
</tr>
<tr>
<td>return y/2;</td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td>return fun2(x);</td>
</tr>
</tbody>
</table>

- Goal: find out ranges of variables
- ... when `fun1` is called in an multithreaded environment.
Control Flow Graph (1)

```c
int fun1(){
    int x = 7;
    if (g<5){
        x++;
    }
    return fun2(x);
}
```
Control Flow Graph (1)

```cpp
int fun1() {
    int x = 7;
    if (g < 5) {
        x++;  
    }
    return fun2(x);
}
```

⇒ if g ≥ 5

- x = 7;
- if g < 5
  - x++;
- if g ≥ 5
  - x++;
- return fun2(x);
Control Flow Graph (2)

```c
int fun2(int y){
    g = 9;
    return y/2;
}
```
int fun2(int y) {
    g = 9;
    return y / 2;
}

⇒

g = 9;

⇒

return y / 2;
fun1():

```
x = ?
x = 7;
if g < 5
    x = ?
    x = 7;
    x++;
    return fun2(x);
else
    g = 9;
    return y / 2;
```

fun2(y):

```
g = 3
```
fun1():

```python
x = 7;

if g < 5:
    x = 7;

if g >= 5:
    x += 1;

return fun2(x);
```

fun2(y):

```python
g = 3;
return y / 2;
```
fun1():

\[ x \text{?} \]

\[ x=7; \]

if \( g \geq 5 \)

\[ x \]

\[ x++ ; \]

return fun2(x);

fun2(y):

\[ g \]

\[ 3 \]
fun1():

x

x = 7;

if g ≥ 5

if g < 5

x = 7

x++;

return fun2(x);

fun2(y):

g 3

return y/2;
fun1():

```python
fun1():
    x = 7;
    if g >= 5:
        x = x + 1;
        return fun2(x);

fun2(y):
    if g < 5:
        x = y;
    else:
        g = 9;
    return y / 2;
```

`g` has the value `3`. 
fun1():

if g ≥ 5

\[
x = 7; \\
x = 7 \\
x = x + 1;
\]

return fun2(x);

fun2(y):

\[
g = 3; \\
return \frac{y}{2};
\]

\[
x = x + 1;
\]
fun1():

```
if g >= 5
    x = 7;
else
    x = ?

if g < 5
    x = 7;
else
    x = 8;
```

```
return fun2(x);
```
fun1():

\[
\begin{align*}
\text{if } g & \geq 5 \rightarrow x = 8 & \text{\quad if } g < 5 \rightarrow x = 7; \\
\text{x=7;} & \text{\quad } x = 7 \\
\text{\quad } x = 7 & \text{\quad } g < 5 \\
\text{\quad } x = 8 & \\
\text{return fun2(x)} \\
\end{align*}
\]

fun2(y):

\[
\begin{align*}
\text{g = 9;} & \\
\text{y = 8} & \\
\text{\quad return y/2;} \\
\text{\quad return y/2;} & \text{\quad g = 3} \\
\end{align*}
\]
fun1():

\[
x \quad ?
\]

\[
x = 7;
\]

\[
x \quad 7
\]

if \( g < 5 \)

\[
x \quad 7
\]

if \( g \geq 5 \)

\[
x \quad 8
\]

\[x++;
\]

return fun2(x);

fun2(y):

\[
y \quad 8
\]

\[g = 9;
\]

return \( y/2 \);

\[
g \quad 3
\]
fun1():

- $x$ is set to 7;
- If $g < 5$, then:
  - $x$ remains 7;
  - $x$ is incremented to 8;
- If $g \geq 5$, then:
  - $x$ remains 8;
- Return $\text{fun2}(x)$;

fun2(y):

- $y$ is 8;
- Set $g = 9$;
- Return $y / 2$;
fun1():

\[
\begin{align*}
\text{x} & \leftarrow \text{?} \\
\text{x} & = 7; \\
\text{x} & \leftarrow \text{?} \\
\text{x} & = 8; \\
\text{return} & \text{ fun2(x);} \\
\end{align*}
\]

fun2(y):

\[
\begin{align*}
\text{y} & \leftarrow 8 \\
\text{g} & = 9; \\
\text{return} & \text{ y/2;} \\
\end{align*}
\]
fun1():

\[ x \] ?

\[ x = 7; \]

\[ x \begin{array}{c} 7 \end{array} \]

if \( g < 5 \)

\[ x \begin{array}{c} 7 \end{array} \]

\[ x++; \]

if \( g \geq 5 \)

\[ x \begin{array}{c} 8 \end{array} \]

return \( \text{fun2}(x) \);

\[ R \]

fun2(y): 

\[ y \begin{array}{c} 8 \end{array} \]

g=9;

\[ y \begin{array}{c} 8 \end{array} \]

return \( y/2; \)

\[ g \begin{array}{c} 3..9 \end{array} \]
fun1():

```
x
x = 7;
if g < 5
  x = 7
if g >= 5
  x = 8
return fun2(x);
```

fun2(y):

```
y = 8
if g >= 5
  g = 9
return y/2;
```

```python
g = 3..9
```
fun1():

```
x  ?
```
x = 7;

```
  x  7
```
if \( g < 5 \)

```
  x  7
```
x++;

```
  x  8
```
return fun2(x);

fun2(y):  

```
y  8
```
g = 9;

```
y  8
```
return \( \frac{y}{2} \);

```
R  4
```
g \( 3..9 \)
fun1():

if g ≥ 5
    x = 7
    x++;
return fun2(x);

fun2(y):

g = 9;
return y/2;

if g < 5
    x = 7
    if g ≥ 5
        x = 7..8
        x++;
return fun2(x);
fun1():

\[ x \begin{cases} ? \\ 7 \\ 7..8 \end{cases} \]

if \( g \geq 5 \)

\[ x=7; \]

if \( g < 5 \)

\[ x=7; \]

return fun2(x);

fun2(y):

\[ y \begin{cases} 8 \\ 7..8 \end{cases} \]

\[ g=9; \]

return \( y/2; \)

\[ g \begin{cases} 3..9 \end{cases} \]
fun1():

```
x  ?
```

x = 7;

```
x  7
```

if g < 5

```
x  7
```

if g ≥ 5

```
x  7..8
```

x++; return fun2(x); return fun2(x);

fun2(y):

```
y  8
```

g = 9;

```
y  8
```

return y/2;

```
y  7..8
```

```
R  4
```

g = 3..9
fun1():

```
x | ?

x=7;
```

```
x | 7
```

```
if g<5
```

```
x | 7
```

```
x | 7..8
```

```
if g\geq5
```

```
x | 7..8
```

```
x | 7..8
```

```
return fun2(x);
```

```
R | 4
```

fun2(y):

```
y | 8
```

```
y | 7..8
```

```
g=9;
```

```
y | 8
```

```
y | 7..8
```

```
return y/2;
```

```
R | 4
```

```
R
```

```
g | 3..9
```
fun1():

```
x = ?
```

```
x = 7
```

```
x = 7..8
```

```
x = 7; if g<5
```

```
x = ++;
```

```
return fun2(x);
```

fun2(y):

```
y = 8
```

```
y = 7..8
```

```
g = 9;
```

```
return y/2;
```

```
R = 4
```

```
R = 3..4
```

```
g = 3..9
```
fun1():

\[
\begin{array}{c}
x \ ? \\
x = 7; \\
x \ 7 \\
\text{if } g < 5 \\
x \ 7 \ldots 8 \\
\text{if } g \geq 5 \\
x \ 7 \ldots 8 \\
\text{return } \text{fun}2(x); \\
\end{array}
\]

fun2(y):

\[
\begin{array}{c}
y \ 8 \\
g = 9; \\
y \ 7 \ldots 8 \\
\text{return } y / 2; \\
R \ 4 \\
R \ 3 \ldots 4 \\
g \ 3 \ldots 9
\end{array}
\]
Data Flow Analysis

- In the previous example, we used Interval Analysis.
Data Flow Analysis

• In the previous example, we used Interval Analysis.

• To describe an analysis we must define
  • a set of potential invariants and
  • a transfer function for each statement.
Outline

1. Data Flow Analysis
2. Lockset Algorithm
3. The Goblint
The Lockset Analysis

• For each program point
  • Compute set of locks that is definitely held.
  • lock(l) adds the lock that l must point to.
  • unlock(l) removes locks that l may point to.

• For each expression e in the program
  • Check if e may point to a shared variable.
  • Record the access with set of locks held.

• Shared var has no common lock \(\Rightarrow\) race!
Simple Example (no race)

\[ T_1 : \text{lock}(&l_1); \]
\[ v = v + 1; \]
\[ \text{unlock}(&l_1); \]

\[ T_2 : \text{lock}(&l_1); \]
\[ v = v + 1; \]
\[ \text{unlock}(&l_1); \]

- List of accesses:
  \[ \langle v, \{l_1\}, \text{write}, \text{file.c : 2}\rangle \]
  \[ \langle v, \{l_1\}, \text{write}, \text{file.c : 5}\rangle \]

- \( v \) is protected by \( \{l_1\} \).
Simple Example (race!)

\[ T_1: \text{lock}(&l_1); \]
\[ v = v + 1; \]
\[ \text{unlock}(&l_1); \]

\[ T_2: \text{lock}(&l_2); \]
\[ v = v + 1; \]
\[ \text{unlock}(&l_2); \]

- List of accesses:
  \[ \langle v, \{l_1\}, \text{write, file.c : 2} \rangle \]
  \[ \langle v, \{l_2\}, \text{write, file.c : 5} \rangle \]

- No common lock!
Outline

1. Data Flow Analysis

2. Lockset Algorithm

3. The Goblint
Goblint

- Implements a special Data Flow Analysis, that can handle multi-threaded C.
Goblint

- Implements a special Data Flow Analysis, that can handle multi-threaded C.
- Started out as a Data Race analyzer using constant propagation and lock sets.
Goblint

- Implements a special Data Flow Analysis, that can handle multi-threaded C.
- Started out as a Data Race analyzer using constant propagation and lock sets.
- Has been expanded to a general framework.
• Implements a special Data Flow Analysis, that can handle multi-threaded C.
• Started out as a Data Race analyzer using constant propagation and lock sets.
• Has been expanded to a general framework.
• Is now used to implement state-of-the-art analyses on multi-threaded code.