A case against semantic dependencies

Ori Lahav, Tel Aviv University
Based on joint discussions with Minki Cho, Chung-Kil Hur, Sung-Hwan Lee, Ben Simner
Imagine a new model for C/C++ that works in three steps:

- **Step 1:** “calculate” a set $S$ of **candidate program execution graphs**
- **Step 2:** given $S$, derive **semantic dependency** ($sdep$) for each graph
- **Step 3:** apply the **consistency predicate** from the C/C++ standard

**Definition 1.** An execution $G$ is called **RC11-consistent** if it is complete and the following hold:

- $hb; eco^?$ is irreflexive. (COHERENCE)
- $rmw \cap (rb; mo) = \emptyset$. (ATOMICITY)
- $psc$ is acyclic. (SC)
- $po \cup rf$ is acyclic. (NO-THIN-AIR)  $sdep \cup rf$ is acyclic.

\[
\begin{align*}
a &= Y (rlx) \\
X &= b (rlx) \\
\text{RW-reorder} &\quad X = b (rlx) \\
X &= a (rlx) \quad a = Y (rlx)
\end{align*}
\]
Imagine a new model for C/C++ that works in three steps:

- **Step 1**: “calculate” a set $S$ of candidate program execution graphs
- **Step 2**: given $S$, derive semantic dependency ($sdep$) for each graph
- **Step 3**: apply the consistency predicate from the C/C++ standard

I believe this approach can’t work. I argue via example that:

- **Step 2 cannot be thread-local**
- **Step 2 has to be aware of the consistency predicate in step 3**
Can “foo” be printed?

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a = X)</td>
<td></td>
</tr>
<tr>
<td>(Y = 1)</td>
<td></td>
</tr>
<tr>
<td>if (a == 1) {</td>
<td>r = Y</td>
</tr>
<tr>
<td>(Y = _1)</td>
<td>(X = r)</td>
</tr>
<tr>
<td>print (&quot;foo&quot;)</td>
<td></td>
</tr>
<tr>
<td>} else {</td>
<td></td>
</tr>
<tr>
<td>(Y = 1)</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
sdep approach

**Thread 1**

a = X

if (a == 1) {
    Y = a
    print ("foo")
} else {
    Y = 1
}

**Thread 2**

r = Y

X = r

X = r

---

```
a = X
if (a == 1) {
    Y = a
    print ("foo")
} else {
    Y = 1
}
```
### Main example

```python
a = X
if (a == 1) {
    Y = a
    print("foo")
} else {
    b = Z
    Y = b
}
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = X</td>
<td>r = Y</td>
<td>Z = 1</td>
</tr>
<tr>
<td>if (a == 1) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y = a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>print(&quot;foo&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>} else {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b = Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y = b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
a = X
if (a == 1) {
    Y = a
    print ("foo")
} else {
    b = Z
    Y = b
}
<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (a == 1) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y = a</td>
<td>r = Y</td>
<td>Z = 1</td>
</tr>
<tr>
<td>print (&quot;foo&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>} else {</td>
<td>X = r</td>
<td></td>
</tr>
<tr>
<td>b = Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y = b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Printing “foo” **has to be allowed**, assuming we allow compilers to:
  - Introduce redundant loads
  - Forward load across atomics:

\[
c = Z; a = X; b = Z \rightarrow c = Z; a = X; b = c
\]

*Both are performed by LLVM/GCC on non-atomics (Z can be easily made non-atomic)*
a = X
if (a == 1) {
  Y = a
  print ("foo")
} else {
  b = Z
  Y = b
}

if (c == 1) {
  a = X
  if (a == 1) {
    Y = a
    print ("foo")
  } else {
    b = c
    Y = b
  }
} else {
  ...
}
Print "foo" has to be allowed, assuming we allow compilers to:

- Introduce redundant loads
- Forward load across atomics:

\[
\begin{align*}
  c &= Z; a = X; b = Z \rightarrow c = Z; a = X; b = c
\end{align*}
\]

Both are performed by LLVM/GCC on non-atomics (Z can be easily made non-atomic)
Thread 1

\[ a = X \]

\[ \text{if} \ (a == 1) \ {\ }
   \begin{align*}
   Y &= a \\
   \text{print} \ (\"foo\")
   \end{align*}
\]

\[ \text{else} \ {\ }
   \begin{align*}
   b &= Z \\
   Y &= b
   \end{align*}
\]

Thread 2

\[ r = Y \]

\[ X = r \]

Thread 3

\[ Z = 0 \]

- With new Thread 3, printing “foo” **has to be disallowed** (thin-air!)
a = X
if (a == 1) {
    Y = a
    print ("foo")
} else {
    b = Z
    Y = b
}

r = Y
Z = 0/1

Step 2 (sdep calculation) cannot be thread local!
Step 2 (sdep calculation) depends on Step 3 (the consistency predicate)!

```
Thread 1
a = X
if (a == 1) {
    Y = a
    print ("foo")
} else {
    b = Z
    Y = b
}

Thread 2
r = Y
X = r

Thread 3
Z = your_favorite_litmus_text()
```

```
U = 1
f = V
V = f rel
g = V

return(d==1 && e==0 && f==1 && g==2)
```
### Substitution of Equivalents — “sanity condition” for weak memory models:

- If \( f() \) always returns \( 0 \) in a memory model \( M \), then \( f() \) and \( 0 \) should be equivalent in \( M \) (assuming \( f() \) uses a disjoint set of locations wrt rest of the program).

```python
Thread 1

a = X
if (a == 1) {
    Y = a
    print("foo")
} else {
    b = Z
    Y = b
}

Thread 2

r = Y
X = r

Thread 3

Z = your_favorite_litmus_test()

T Thread 1
T Thread 2
T Thread 3
```
Reasoning-aware $sdep$? :

- $sdep$ calculation has to take into account our reasoning principles.
  
  - No thin air values: $f()$ never returns 1 in some (possibly inconsistent) execution
    $\implies sdep$ must exist

  - A new (sound) program logic can prove that $f()$ never returns 1
    $\implies sdep$ must exist

- We have a memory model for reasoning (weaker than the “real” model).

- For reasoning to be potentially precise, $sdep$ needs to take into account the full consistency predicate.
The source of the problem

• Semantic dependencies are “dynamic” rather than “static”:

  - \textcolor{red}{\texttt{sdep}} \iff the model allows a thread to read some value at a certain program point.

  - Event-structure-based / pomset models / “Promising Semantics” capture such dynamic dependencies.

• The approach we discussed fails to do so.
A fresh look on the out-of-thin-air problem

- The discussion about the OOTA problem in C/C++ revolves around memory_order_relaxed
  - *Is it indeed expensive to forbid RW reordering of relaxed accesses?*
  - More provocatively: *do we really need relaxed writes?*

- A (more practical?) challenging problem arises with:
  - Strong accesses (SC) or mutexes that allow races
  - Weak accesses (non-atomic) that allow optimizations, including load introduction

see our PLDI’23 paper