Semantics of C++
Concurrency

David Hauzar
hauzar@d3s.mff.cuni.cz
Mathematizing C++ Concurrency

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, Tjark Weber

University of Cambridge

POPL 2011

Formally described the current draft of concurrent behavior in C/C++

Created a tool that makes it possible to explore possible behavior for given C/C++ fragment
Why should we be interested?

- C/C++ is the most wide-spread programming language
- GMC tool
  - Explicit model checker of C/C++ languages
  - GAUK, GACR proposals
1. CONCURRENT BEHAVIOR OF C++
How to guarantee that the receiver will receive data from the sender?

```c
// sender
data = ...;
p = &data;
```  

```c
// receiver
r1 = p
r2 = *r1;  // data
```

- The design criteria
  - Implementability
  - Useful abstractions
  - Precise definition
Defining behavior: single-threaded programs

- Concepts
  - *Happens-before* (hb)
    - *Sequenced-before* (sb)
  - *Reads-from* (rf)
    - *Visible side effect* (vse)

- Undefined behavior
  - *Indeterminate reads*
  - *Unsequenced races*

```c
int main() {
    int x = 2;
    int y = 0;
    y = (x == (x=3));
    return 0; }
```
Defining behavior: multi-threaded programs

- Concepts
  - Additional-synchronizes-with (asw)

- Undefined behavior
  - Data race (dr)

```c++
int main() {
    int x = 2;
    int y;
    {{
        x = 3;
        y = (x==3);
    }};
    return 0;
}
```
Mutexes

- Lock and unlock memory actions
- Sc relation
  - Totally orders lock and unlock actions
- Synchronizes-with edges from every unlock to every lock that is ordered after it in sc relation
Sequentially consistent atomics

- SC read action, SC write action
- SC atomic operations are totally ordered by sc
- Interleaving with each other in a global time-line

```cpp
atomic_int x = 0;
atomic_int y = 0;
x.store(1, seq_cst);
y.load(seq_cst);

atomic_int x = 0;
atomic_int y = 0;
x.store(1, seq_cst);
y.load(seq_cst);
```

Diagram:
- \( W_{SC} \) for write operation
- \( R_{SC} \) for read operation
- Interleaving with sc operations in a global timeline
Sequentially consistent atomics – initial example

```cpp
atomic data;
atomic flag;

// sender
data = ...;
p.store(&data, mo_seq_cst);
flag.store(1, mo_seq_cst);

// receiver
while (!flag.load(mo_seq_cst))
    r1 = p.load(mo_seq_cst);
    r2 = *r1;
```

\[ W_{na}\text{data} = 15 \]
\[ W_{SC}p = &data \]
\[ W_{SC}\text{flag} = 1 \]
\[ R_{SC}\text{flag} = 1 \]
\[ R_{SC}p = data \]
\[ R_{na}r1 = 15 \]
Release/Acquire atomics

- Release action (write), acquire action (read)
- The acquire action cannot see any values of data that precede release action in modification order
- Concepts
  - modification-order
  - release-sequence
  - synchronizes-with
Release/Acquire atomics – initial example

```c++
// sender
x = ...
y.store(1, release);

// receiver
while (0 == y.load(acquire));
r = x;
```
Release/Consume atomics

- Efficient implementation of algorithms that use pointer reassignment for commits of their data
- Release action (write), consume action (read)
- Main concepts
  - Data-dependency (dd)
  - Carries-a-dependency-to (cad)
  - Dependency-order-before (dob)

```c++
int main() {
    int data; atomic_address p;
    {{{
        data=1;
        p.store(&data, mo_release); }
    }};
    printf("%d\n", *(p.load(mo_consume)));
    return 0; }
```
There must be a dependency between the reads!

```cpp
// sender
x = ... 
y.store(1, release);

// receiver
while (0 == y.load(mo_consume()))
    r = x;
```

```
b:W_{na} x=1  d:R_{CON} y=1
   sb   dr   sb
  c:W_{REL} y=1  e:R_{na} x=0
```
Relaxed atomics

- relaxed reads and writes are fast

```cpp
// sender
x = ...  // receiver
y.store(1, release);
while (0 == y.load(acquire));
r = x;

// sender
x = ...  // receiver
y.store(1, release);
while (0 == y.load(acquire))
    { while (0 == y.load(relaxed)); }
r = x;
```
Refinements to the standard (1)

- Fixed
  - Happens before relation
  - Definition of SC atomics
  - Coherence requirements
- Not fixed
  - Overlapping execution and thin-air reads
Refinements to the standard (2)

- **Coherence requirements**
  - Candidate execution is required to be free of the following execution fragments:

  ![Diagram of coherence requirements]

  - The draft standard originally enforced **CoRR** and **CoWW**
Refinements to the standard (3)

- Thin-air reads
  - Need to find solution that is not overly constraining

```cpp
int main() {
    int r1, r2;
    atomic_int x = 0;
    atomic_int y = 0;
    {{{ { r1 = x.load(mo_relaxed));
        y.store(r1,mo_relaxed); }
    }|| { r2 = y.load(mo_relaxed));
        x.store(r2,mo_relaxed); }
}}
    return 0; }
```
2. CPPMEM TOOL
CPPMEM Tool (1)

- **Input**
  - Program in a fragment of C/C++

- **Output**
  - The set of its executions allowed by the memory model
The semantics of the program $p$ is set of allowed executions $X$

- $X_{\text{opsem}}$ part
  - Can be decided by the syntactic structure of the source code
    - sequenced-before, additional-synchronized-with, data-dependency

- $X_{\text{witness}}$ part
  - Existentially quantified in the definition of cpp_memory_model
    - read-from, sc, modification-order

Computation of the set of allowed executions

1. Compute $X_{\text{opsem}}$ part from operational semantics
2. Enumerate possible $X_{\text{witness}}$ for each of those
3. Calculate the derived relations and predicates of the model, check whether it is consistent
Summary

- Formal model of the specification
  - www.cl.cam.ac.uk/users/pes20/cpp
  - Clearer than prose specification
  - Fixed problems in the specification
- The prototype x86 implementation
- Formal model is building stone for:
  - Optimized compilation
  - Static analysis
  - Dynamic analysis
  - Verification
  - …