Mixed size non-atomics in C/C++11

Kyndylan Nienhuis and Mark Batty

November 7, 2016

This note describes our extension of the C/C++11 axiomatic concurrency model [2, 4, 3, 1] to cover mixed-size nonatomic accesses.

1 The proposed model

To distinguish between character and whole object accesses we add *footprints* to reads and writes. In the following example the footprint of the first write is the whole object, while the footprint of the second write is only the second byte.

```
int x = 42;
*((char *)(&x + 1)) = 0;
printf("%i", x);
```

This example also shows that reads can now read from multiple writes. To make explicit which part the read is reading from which write, we also add footprints to *rf*-edges. In the example the footprint of the *rf*-edge from the first write to the read is the whole object minus the second byte, while the footprint of the *rf*-edge between the second write and the read is just the second byte.

We leave the type *footprint* abstract, so that users of the proposed model can implement it as they wish, and we only manipulate footprints using the following functions (which also have to be implemented by users of the model). One possibility would be to implement a footprint as a set of addresses, where an address points to one byte in the memory, and to implement the functions below by their corresponding functions on sets.

- val *footprint_empty* : FOOTPRINT
- val *footprint_is_empty* : FOOTPRINT $\rightarrow$ BOOL
- val *footprint_inclusion* : FOOTPRINT $\rightarrow$ FOOTPRINT $\rightarrow$ BOOL
- val *footprint_difference* : FOOTPRINT $\rightarrow$ FOOTPRINT $\rightarrow$ FOOTPRINT
- val *footprint_intersection* : FOOTPRINT $\rightarrow$ FOOTPRINT $\rightarrow$ FOOTPRINT
- val *footprint_bigunion* : SET FOOTPRINT $\rightarrow$ FOOTPRINT

The possibility of multiple *rf*-edges to a read means that the value of the read can no longer simply be the value of the write the read reads from. To determine this value we use the following function whose implementation is also left to users of the model. The parameter is a set of tuples (v, f_1, f_2) where v is the value of a write, f_1 the footprint of the write and f_2 the footprint of the *rf*-edge from that write. The return value can be *nothing*, for example in the case that the set is empty.

- val *combine_cvalues* : SET (CVALUE ∗ FOOTPRINT ∗ FOOTPRINT) → MAYBE CVALUE

Below we discuss how the proposed model differs from the original axiomatic model. For each definition *[name]* that we changed we use the name *[name]-fp* for the new definition, where *fp* stands for footprint.

1.1 Visible side effects

In the original model the visible side effects to a read r are all the writes w to the same location with $(w, r) \in hb$ and such that there is no write to the same location that is *hb*-between w and r. This is formally defined below.

```
val visible_side_effect_set : SET ACTION → SET (ACTION ∗ ACTION) →
SET (ACTION ∗ ACTION)
let visible_side_effect_set actions₁ hb =
  { (a, b) | ∀ (a, b) ∈ hb |
    is_write a ∧ is_read b ∧ (loc_of a = loc_of b) ∧
    ¬ ( ∃ c ∈ actions₁. ¬ (c ∈ {a, b}) ∧
      is_write c ∧ (loc_of c = loc_of b) ∧
      (a, c) ∈ hb ∧ (c, b) ∈ hb) }
```

This definition does not suffice anymore: the read in the example of the beginning of this section can see the first write, although there is a write *hb*-between. Instead we define visible side effects as follows: if there is a part f of the footprint of a write w that is not overwritten by writes *hb*-between w and r, than (w, f, r) is a visible side effect. This is formally defined below.

```
val visible_side_effect_set_fp : SET ACTION → SET (ACTION ∗ ACTION) →
SET (ACTION ∗ FOOTPRINT ∗ ACTION)
let visible_side_effect_set_fp actions₁ hb =
 let x = (fun (a, b) →
    let overwriting_footprint =
      footprint_bigunion {footprint_of c
                        | ∀ c ∈ actions₁
                        | ¬ (c ∈ {a, b}) ∧ is_write c ∧
                          (a,  c) ∈ hb ∧ (c,  b) ∈ hb} in
    let remaining_footprint_of_a =
      footprint_difference (footprint_of a) overwriting_footprint in
    let footprint_between_a_and_b =
      footprint_intersection (footprint_of b) remaining_footprint_of_a in
```

```
    (a, footprint_between_a_and_b, b)) in
  { (a, f, b) | ∀ (a, f, b) ∈ Set.map x hb |
    is_write a ∧ is_read b ∧ ¬ (footprint_is_empty f)}
```

1.2 Well formed rf

In the original *well-formed-rf* predicate (which is displayed below) we change loc_of a = loc_of b by the requirement that the footprint f of the *rf*-edge is included in both the footprints of a and b, and that f is non-empty. The last conjunct of the original predicate requires there is at most one *rf*-edge to each read. We now only require that for atomics (and phrase it in a different way) and for non-atomics we require that there is at most one *rf*-edge between every pair (a, b) of write and read, and that the footprints of all the *rf*-edges to the same read are disjoint. Finally, we change value_written_by a to a computation that combines the values of all the writes that a read reads from. To improve readability, we moved this requirement to a separate definition $well\_formed\_rf\_fp_2$, named the remainder of the predicate $well\_formed\_rf\_fp_1$, and defined *well_formed_rf_fp* as the conjunction of these two predicates.

```
val well_formed_rf : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST →
BOOL
let well_formed_rf (Xo, Xw, _) =
  ∀ (a, b) ∈ Xw.rf.
    a ∈ Xo.actions ∧ b ∈ Xo.actions ∧
    loc_of a = loc_of b ∧
    is_write a ∧ is_read b ∧
    value_read_by b = value_written_by a ∧
    ∀ a′ ∈ Xo.actions. (a′, b) ∈ Xw.rf ⟶ a = a′

val well_formed_rf_fp1 : CANDIDATE_EXECUTION_FP → BOOL
let well_formed_rf_fp1 (Xo, Xw, _) =
  (∀ (w, f, r) ∈ Xw.rf_fp.
     w ∈ Xo.actions ∧ r ∈ Xo.actions ∧
     is_write w ∧ is_read r ∧
     ¬ (footprint_is_empty f) ∧
     footprint_inclusion f (footprint_of w) ∧
     footprint_inclusion f (footprint_of r) ∧
     let writes_of_r = {(w′, f′) | ∀ (w′, f′, r′) ∈ Xw.rf_fp | r = r′} in
     (is_at_atomic_location Xo.lk r ⟶ writes_of_r = {(w, f)}) ∧
     (is_at_non_atomic_location Xo.lk r ⟶
        (∀ (w′, f′) ∈ writes_of_r.
           (w = w′ ⟶ f = f′) ∧
           (w ≠ w′ ⟶ footprint_is_empty (footprint_intersection f f′)))))

val well_formed_rf_fp2 : CANDIDATE_EXECUTION_FP → BOOL
let well_formed_rf_fp2 (Xo, Xw, _) =
```

```
(∀ r ∈ Xo.actions.
   let writes_of_r = {(w, f) | ∀ (w, f, r′) ∈ Xw.rf_fp | r = r′} in
   (¬ (null writes_of_r)) ⟶
   value_read_by r =
   combine_cvalues (Set.setMapMaybe (fun (w, f) → match value_written_by w with
                                    | Just v → Just (v, footprint_of w, f)
                                       | Nothing → Nothing
                                       end)
                          writes_of_r))

val well_formed_rf_fp : CANDIDATE_EXECUTION_FP → BOOL
let well_formed_rf_fp ex =
   well_formed_rf_fp1 ex ∧ well_formed_rf_fp2 ex
```

1.3 Consistent non-atomic rf

The *consistent-non-atomic-rf* predicate requires that non-atomic reads only read from visible side effects. Both *rf*- and *vse*-edges now have footprints, but it would be wrong to require that the *rf*-edge is a visible side effect with the same footprint: in a racy program there could be distinct writes w and w' such that (w, f, r) and (w', f, r) are both visible side effects, and r could read only a part of w and read the rest from w'. This means that the *rf*-edges to r would not have f as footprint, so they are not included in *vse*. Instead we require that for every *rf*-edge there is a *vse*-edge whose footprint includes the footprint of the *rf*-edge.

```
val consistent_non_atomic_rf : PRE_EXECUTION ∗ EXECUTION_WITNESS ∗
RELATION_LIST → BOOL
let consistent_non_atomic_rf (Xo, Xw, _ :: (“vse”, vse) :: _) =
   ∀ (w, r) ∈ Xw.rf. is_at_non_atomic_location Xo.lk r ⟶
     (w, r) ∈ vse

val consistent_non_atomic_rf_fp : CANDIDATE_EXECUTION_FP → BOOL
let consistent_non_atomic_rf_fp (Xo, Xw, rel1) =
   ∀ (w, f, r) ∈ Xw.rf_fp. is_at_non_atomic_location Xo.lk r ⟶
     ∃ (w′, f′, r′) ∈ rel1.vse_fp. w = w′ ∧ r = r′ ∧ footprint_inclusion f f′
```

1.4 Determinate reads

The original *determinate-reads* predicate requires that a load r has an *rf*-edge to it if and only if there exists a visible side effect to r. In our proposed model we instead require that the union of the footprints of the *rf*-edges to r equals the union of the footprints of the *vse*-edges to r.

```
val det_read : PRE_EXECUTION ∗ EXECUTION_WITNESS ∗ RELATION_LIST →
BOOL
let det_read (Xo, Xw, _ :: (“vse”, vse) :: _) =
```

$\forall\ r \in Xo.actions.$
  $\text{is\_load}\ r \longrightarrow$
  $(\exists\ w \in Xo.actions.\ (w,\ r) \in vse) =$
  $(\exists\ w' \in Xo.actions.\ (w',\ r) \in Xw.rf)$

val *det_read_fp* : CANDIDATE_EXECUTION_FP → BOOL
let *det_read_fp* $(Xo,\ Xw,\ rel_1)$ =
  $\forall\ r \in Xo.actions.$
    $\text{is\_load}\ r \longrightarrow$
    $\text{footprint\_bigunion}\ \{f \mid \forall\ (w,\ f,\ r') \in rel_1.vse\_fp \mid r = r'\} =$
    $\text{footprint\_bigunion}\ \{f \mid \forall\ (w,\ f,\ r') \in Xw.rf\_fp \mid r = r'\}$

1.5 Indeterminate reads

The original function *indeterminate-reads* returns the set of reads that have no *rf*-edge to them. In our proposed model this function returns the set of reads r whose footprint is not covered by the footprints of the *rf*-edges to r.

val *indeterminate_reads* : CANDIDATE_EXECUTION → SET ACTION
let *indeterminate_reads* $(Xo,\ Xw,\ \_)$ =
  $\{b \mid \forall\ b \in Xo.actions \mid \text{is\_read}\ b \wedge \neg\ (\exists\ a \in Xo.actions.\ (a,\ b) \in Xw.rf)\}$

val *indeterminate_reads_fp* : CANDIDATE_EXECUTION_FP → SET ACTION
let *indeterminate_reads_fp* $(Xo,\ Xw,\ rel_1)$ =
  $\{b \mid \forall\ b \in Xo.actions \mid$
    $\text{is\_read}\ b \wedge$
    let *footprint_of_writes* =
      $\text{footprint\_bigunion}\ \{f \mid \forall\ (w,\ f,\ r') \in Xw.rf\_fp \mid r' = b\}$ in
    $\neg\ (\text{footprint\_inclusion}\ (\text{footprint\_of}\ b)\ (footprint\_of\_writes))\}$

1.6 Races

Both in *unsequenced-races* and in *data-races* we change loc_of a = loc_of b by footprint_overlap (footprint_of a) (footprint_of b), which is defined as follows.

val *footprint_overlap* : FOOTPRINT → FOOTPRINT → BOOL
let *footprint_overlap* $f_1\ f_2$ =
  $\neg\ (\text{footprint\_is\_empty}\ (\text{footprint\_intersection}\ f_1\ f_2))$

References

[1] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *Proc. POPL*, 2011.

[2] Hans-J Boehm and Sarita V Adve. Foundations of the C++ concurrency

memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008.

[3] WG14. ISO/IEC 14882:2011.

[4] WG14. ISO/IEC 9899:2011.