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 → BOOL`
- `val footprint_is_empty : FOOTPRINT → BOOL`
- `val footprint_inclusion : FOOTPRINT → FOOTPRINT → BOOL`
- `val footprint_difference : FOOTPRINT → FOOTPRINT → FOOTPRINT`
- `val footprint_intersection : FOOTPRINT → FOOTPRINT → FOOTPRINT`
- `val footprint_bigunion : SET FOOTPRINT → 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 \textit{nothing}, for example in the case that the set is empty.

- \texttt{val combine_cvalues} : \texttt{SET (CVALUE * FOOTPRINT * FOOTPRINT) \rightarrow MAYBE CVALUE}

Below we discuss how the proposed model differs from the original axiomatic model. For each definition \texttt{name} that we changed we use the name \texttt{name}-fp for the new definition, where \texttt{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.

\[
\text{val \ visible_side_effect_set : SET ACTION \rightarrow SET (ACTION * ACTION) \rightarrow (ACTION * ACTION)}
\]

\[
\text{let \ visible_side_effect_set actions\_1 hb =}
\{
(a, b) \mid \forall (a, b) \in hb
\quad \text{is\_write } a \land \text{is\_read } b \land \text{loc\_of } a = \text{loc\_of } b \land
\quad \neg (\exists c \in \text{actions\_1}. \neg (c \in \{a, b\}) \land
\quad \text{is\_write } c \land \text{loc\_of } c = \text{loc\_of } b) \land
\quad (a, c) \in hb \land (c, b) \in hb
\}
\]

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 \(r\), then \((w, f, r)\) is a visible side effect. This is formally defined below.

\[
\text{val \ visible_side_effect_set\_fp : SET ACTION \rightarrow SET (ACTION * ACTION) \rightarrow (ACTION * ACTION)}
\]

\[
\text{let \ visible_side_effect_set\_fp actions\_1 hb =}
\text{let \ x = \text{(fun } (a, b) \rightarrow }
\text{let \ \textit{overwriting\_footprint} = }
\text{footprint\_bigunion \{\text{footprint\_of } c
\mid \forall c \in \text{actions\_1}
\quad \neg (c \in \{a, b\}) \land \text{is\_write } c \land
\quad (a, c) \in hb \land (c, b) \in hb\}}
\text{in}
\text{let \ \textit{remaining\_footprint\_of\_a} = }
\text{footprint\_difference (footprint\_of \ a) \ \textit{overwriting\_footprint} in}
\text{let \ \textit{footprint\_between\_a\_and\_b} = }
\text{footprint\_intersection (footprint\_of b) \ \textit{remaining\_footprint\_of\_a} in}
\]
(a, footprint_between_a_and_b, b)) in
\{ (a, f, b) | \forall (a, f, b) \in \text{Set.map } x hb |
is_write a \land is_read b \land \neg (\text{footprint_is_empty } f)\}

### 1.2 Well formed rf

In the original well-formed-rf predicate (which is displayed below) we change
\text{loc_of } a = \text{loc_of } b \text{ by the requirement that the footprint } f \text{ of the rf-edge is}
included in both the footprints of } a \text{ and } b, \text{ and that } f\text{ 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 dif-
f erent way) and for non-atomics we require that there is at most one rf-edge
between every pair } (a, b) \text{ 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
\text{well_formed_rf_fp}_2, \text{ 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 \rightarrow BOOL**

let well_formed_rf (Xo, Xw, _) =
\forall (a, b) \in Xw.rf.
\ a \in Xo.actions \land b \in Xo.actions \land
\text{loc_of } a = \text{loc_of } b \land
\text{is_write } a \land \text{is_read } b \land
\text{value_read_by } b = \text{value_written_by } a \land
\forall a' \in Xo.actions. (a', b) \in Xw.rf \implies a = a'

**val well_formed_rf_fp1 : CANDIDATE_EXECUTION_FP \rightarrow BOOL**

let well_formed_rf_fp1 (Xo, Xw, _) =
\forall (w, f, r) \in Xw.rf_fp.
\ w \in Xo.actions \land r \in Xo.actions \land
\text{is_write } w \land \text{is_read } r \land
\neg (\text{footprint_is_empty } f) \land
\text{footprint_inclusion } f \text{ (footprint_of } w) \land
\text{footprint_inclusion } f \text{ (footprint_of } r) \land
let writes_of_r = \{(w', f') | \forall (w', f', r') \in Xw.rf_fp | r = r'\} \in
\text{is_at_atomic_location } Xo.lk r \implies \text{writes_of_r} = \{(w, f)\}\land
\text{is_at_non_atomic_location } Xo.lk r \implies
\forall (w', f') \in \text{writes_of_r}.
\ (w = w' \implies f = f') \land
\ (w \neq w' \implies \text{footprint_is_empty } (\text{footprint_intersection } f f'))

**val well_formed_rf_fp2 : CANDIDATE_EXECUTION_FP \rightarrow BOOL**

let well_formed_rf_fp2 (Xo, Xw, _) =
(∀ r ∈ X₀.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_vvalues(Set.setMapMaybe (fun (w, f) →
| 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 (X₀, Xw, _ :: (“vse”, vse) :: _) =
∀ (w, r) ∈ Xw.rf. is_at_non_atomic_location X₀.lk r →
(w, r) ∈ vse

val consistent_non_atomic_rf_fp : CANDIDATE_EXECUTION_FP → BOOL
let consistent_non_atomic_rf_fp (X₀, Xw, rel₁) =
∀ (w, f, r) ∈ Xw.rf fp. is_at_non_atomic_location X₀.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 (X₀, Xw, _ :: (“vse”, vse) :: _) =
∀ r ∈ Xo.actions.
is_load r →
(∃ w ∈ Xo.actions. (w, r) ∈ vse) =
(∃ w′ ∈ Xo.actions. (w′, r) ∈ Xw.rf)

val det_read fp : CANDIDATE_EXECUTION_FP → BOOL
let det_read_fp (Xo, Xw, rel₁) =
∀ r ∈ Xo.actions.
is_load r →
footprint_bigunion {f | ∀ (w, f, r′) ∈ rel₁.vse.fp | r = r′} =
footprint_bigunion {f | ∀ (w, f, r′) ∈ Xw.rf.fp | 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 | ∀ b ∈ Xo.actions | is_read b ∧ ¬(∃ a ∈ Xo.actions. (a, b) ∈ Xw.rf)}

val indeterminate_reads fp : CANDIDATE_EXECUTION_FP → SET ACTION
let indeterminate_reads fp (Xo, Xw, rel₁) =
{ b | ∀ b ∈ Xo.actions |
is_read b ∧
let footprint_of_writes =
footprint_bigunion {f | ∀ (w, f, r′) ∈ Xw.rf.fp | r′ = b} in
¬(footprint_inclusion (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 f₁ f₂ =
¬(footprint_is_empty (footprint_intersection f₁ f₂))

References
