GO2 TEAM MEETING #1

JAN 09, 2020

Referencing StaticRaceChecker2:

The tool first concatenates all directory paths. For each path, it loads the source code and create the ***SSA representation*** for all path-specific packages and their dependencies.

```
// Create SSA packages for well-typed packages and their dependencies.
prog, pkgs := ssautil.AllPackages(initial, 0)

// Build SSA code for the whole program.
prog.Build()
```

lines 1245-1248

It then proceeds to retrieve information per ***instruction*** (derived from each function in the SSA program).

ie. Each `RecordSet` consists of 4 structs (`RecordField`): `Field` `Array` `Basic` and `Map` Each containing information regarding SSA Instruction, SSA Value, Field index, whether or not it is a Write Operation, and whether or not it is an Atomic Operation.

```
for i, recordSet := range [][]RecordField{r.RecordsetField, r.RecordSetArray, r.RecordsetBasic, r.RecordsetMap}
```

line 277

$\texttt{Pairs}_{\texttt{candidate}}$ are generated according to the following conditions

- at least one RecordSet is a ***Write Operation***
- RecordSets do ***not*** share the same Basic Block
- at least one RecordSet is ***not*** an Atomic Operation
- at least one RecordSet is in an anonymous function // ***(DEBATABLE)***
- both RecordSets originate from the ***same*** Field path

```
(pi.isWrite || pj.isWrite) &&
//      reflect.DeepEqual(pi.value.Type(), pj.value.Type()) &&
pi.Field == pj.Field &&
pi.ins.Block() != pj.ins.Block() &&
(!pi.isAtomic || !pj.isAtomic) &&
(isInAnnoFunc(pi.ins) || isInAnnoFunc(pj.ins)) &&
r.FastSame(pi.value, pj.value) { //&&
```

Call graph created with Go's `golang.org/x/tools/go/callgraph` package validates whether both elements of a $\texttt{Pair}_{\texttt{candidate}}$ are reachable.

Taking two nodes from the call graph, if both share a common parent, start with the common parent, otherwise begin at the root. ***
Then create a `map` to record succession along the call chain and flag each node as **reachable**, **reachable and spawned by goroutine** or **unreachable**.

$\texttt{Pair}_{\texttt{concur}}$ only occurs if both nodes are **reachable**, and at least one of them is **reachable and spawned by goroutine**.

```
for i := 0; i < len(queue); i++ {
    now := queue[i]
    state := seen[now]
    for _, e := range now.Out {
        newState := state
        if _, isGo := e.Site.(*ssa.Go); isGo && !BlackFuncList[e.Site.Common().Value.String()] {
            newState = 2 // called by go
        }
        if seen[e.Callee] < newState {
            seen[e.Callee] = newState
            queue = append(queue, e.Callee)
        }
    }
}

return seen[node1]+seen[node2] >= 3 // Go + reachable
```

For each (a, b) in $Pair_{concur}$, check

$$a <_{HB} b \ or \ b <_{HB} a$$

Recall HB is defined as $(PO \cup SO)^{+}$.

Program Order (PO): calculate the must-happen-before and must-happen-after sets of a and b.

$$B_a \triangleq \{ins \mid ins <_{MHB} a\}$$
$$A_a \triangleq \{ins \mid ins <_{MHA} a\}$$

Similarly for B_b, A_b.

According to the program order:

- The must-happen-before set includes all synchronization operations and Go instructions that occur before the targeted instruction.
- The must-happen-after set includes all synchronization operations and Go instructions that occur after the targeted instruction.

*** It appears to us that the program omits atomic instructions (e.g. `atomicLoadInt`)

Sync Order (SO): Collected via the following:

- Go instruction
- Channel `send`, `recv`, `close`
- Locks `Mutex`, `RWMutex`
- WaitGroup `Done`, `Wait`

Consider each pair of instructions (i_1, i_2) in $B_a \times A_b \cup A_a \times B_b$, if i_1 ***can*** happen before i_2, according to the Go memory model, then a race is ***NOT*** reported.

Declared Limitations by Author

- Sync Ops in loops and branches are not taken into design consideration
- Synchronization between channel ops is modeled naively.
  - assumes empty channels, with buffer size of at most 1.
- Ignorant to transitivity of *Happens-Before*.

Observed false negetives not mentioned by the author:

Case 1

```
func main() {
        fmt.Println(getNumber())
}
func getNumber() int {
        var i int
        go writeI(&i)
        return i
}
func writeI(i *int) {
        *i = 5
}
```

Case 2

```
type S struct {
    i int
}
func (s *S) read() int {
    return s.i
}
func (s *S) write(i int) {
    s.i = i
}
func main() {
    s := &S{ i: 1 }
    go func() {
        s.write(12)
    }()
    s.read()
}
```

Case 3

```
var i int
func main() {
        go func() {
                i = 1
        }()
        i = 2
}
```