Finalization should not be based on reachability

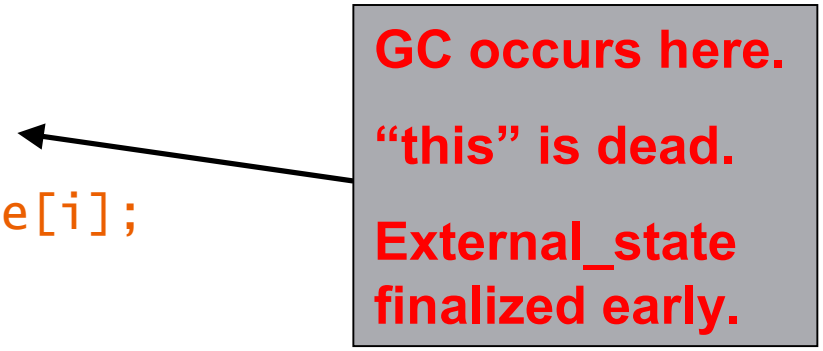
Hans-J. Boehm
HP Labs
(This benefited from discussions with Mike Spertus and others.)

The real problem with finalization

- "Live" objects may be unreachable and hence finalized.

  Last call on object, finalizer cleans up external_state:

```
foo()
{
        int i = this.my_index;
        …
        manipulate external_state[i];
}
```



No painless solutions

- Applies also to "weak references", if those can be used to detect reachability.
- Has nothing to do with object resurrection, etc.
- Eliminating finalization means that user may have to duplicate GC work, e.g. for
  - distributed GC
  - "native" object management
  - "WeakHashMap"-like field addition

Three "conventional" solutions

- The "C++ destructor" solution
  - Outlaw "dead" variable elimination.
  - Expensive (?), since (unlike C++ destructors) it seems to affect all pointer variables.
- Finalize only at "safe" points.
  - Used by some Scheme Guardian implementations?
  - Multithreaded version seems ugly, inconvenient.
  - We don't know how to fix it.
- Programmer inserts explicit "keepAlive()" calls after external state reference.
  - Java 6 solution. Ugly, but workable.

A different (?) approach

- Look at the last alternative differently.
  - And tweak it a bit.
- An object *A* may be finalized anytime after the last call to its keepAlive() method.
  - … not counting calls enabled by the call of *A*s finalize method.
- The implementation continues to be based on GC-determined reachability,
  - If it's unreachable I can't call keepAlive() on it.
  - In fact, keepAlive() only needs to extend live range.
    - No instructions generated (except possibly register spills).
  - But the programmer should not think in those terms.

Revised (roughly like Java 6)

- Last call on object, finalizer cleans up external_state:

```
foo()
{
        int i = this.my_index;
        …
        manipulate external_state[i];
        keepAlive();
}
```

GC occurs here.

Programmer: keepAlive() call possible.

GC: “this” is live.

No premature finalization.

Advantages

- Straightforward description.
- No implementation cost unless finalizers are used.
- No need to define “reachability” for a language like Java.
- Finalizer ordering is handled implicitly.
  - If A references B and needs it for finalization, A’s finalizer will call B’s method, which calls B.keepAlive().
- Can test by running finalizers asap?
  - Checkpoint at keepAlive calls.

Known issues:

- Existing code breaks.
  - But most of it was wrong anyway.
  - KeepAlive calls are hard to avoid.
- Unordered finalization breaks(?)
  - Good riddance.
- Doesn't handle common WeakHashMap uses.
  - WeakHashMap without removal detection(?)
    - "Applying" the map is the only interesting allowed operation.
    - If key is reclaimed, entry can be transparently removed.