GARBAGE COLLECTION IN AN UNCOOPERATIVE ENVIRONMENT

Hans-Jürgen Böhm, Mark Weiser

STATE OF THE ART

- conventional storage management
  - conservative: all garbage should eventually be reclaimed
    - reference counting
    - requires program cooperation for every pointer assignment
- garbage collector determines accessibility
  - locate references, distinguish them from data

APPROACHES

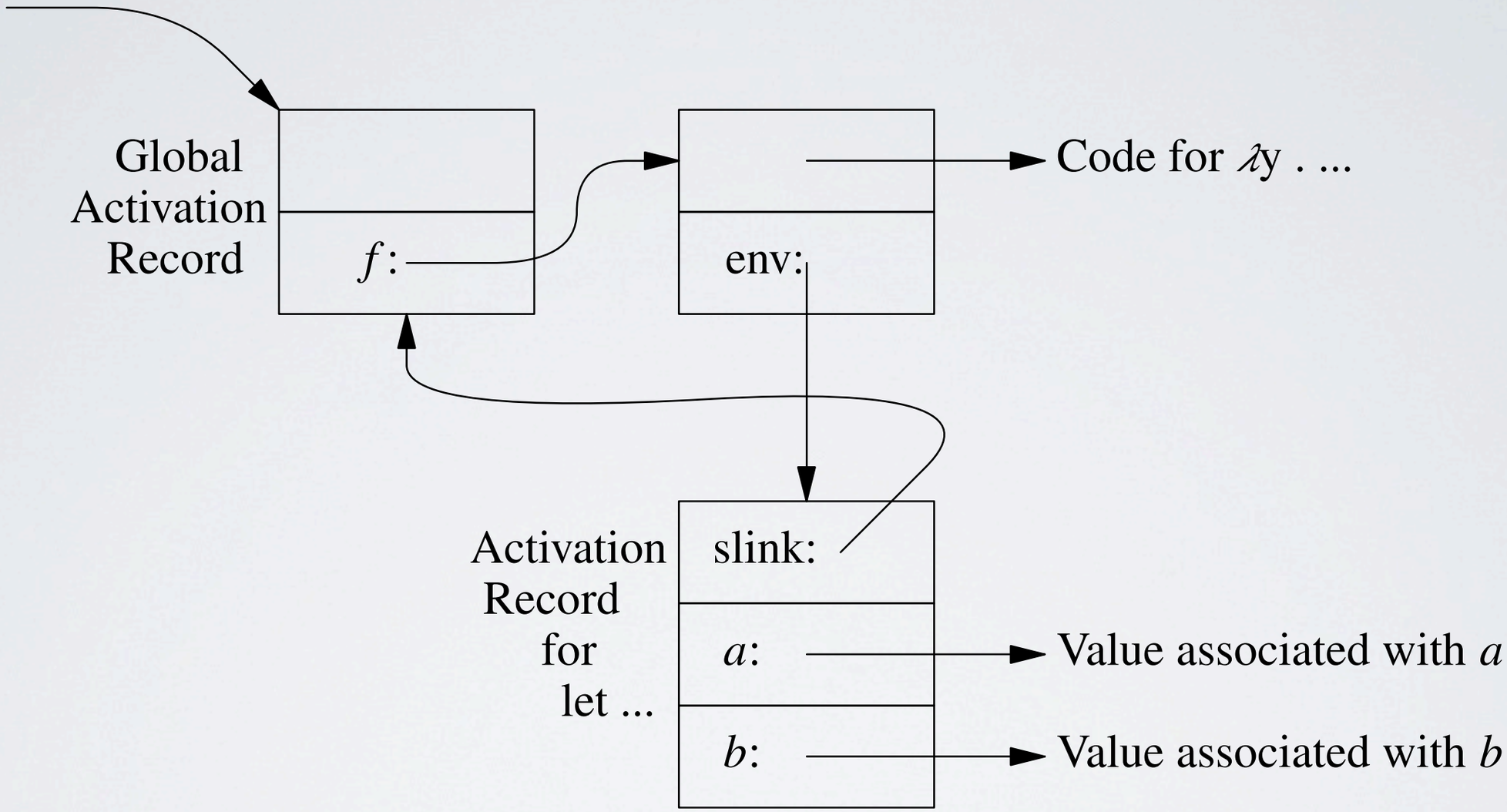
- each data item contains information to identify pointers
  - shadow values, fat data items, slow
- partition memory and registers
  - does not work directly with the stack
- tailor-made traversal routine
  - does not work with polymorphic functions

LIMITATION

- we get: reclaim objects not accessible **at runtime**
- we want: reclaim objects not accessible **by the program**
- compiler may fail to clear references
- activation records (stack frames) may keep unaccessible objects alive

```
let
    f = let
        a = …
        b = …
    in
        if … a … b … then (λy.y)
        else (λy.y+1)
in
    A: …
```

```
int main(void) {
    int f(int y) {
        int a = …
        int b = …
        if (a … b) then return y
        else return y+1;
    }
A: …
}
```


Global
Activation
Record
f:
env:
Code for λy . ...
Activation
Record
for
let ...
slink:
a:
b:
Value associated with a
Value associated with b

REALITY-CHECK

Accept the fact that any garbage collector may fail to reclaim memory that can never be accessed.

DESIGN GOALS

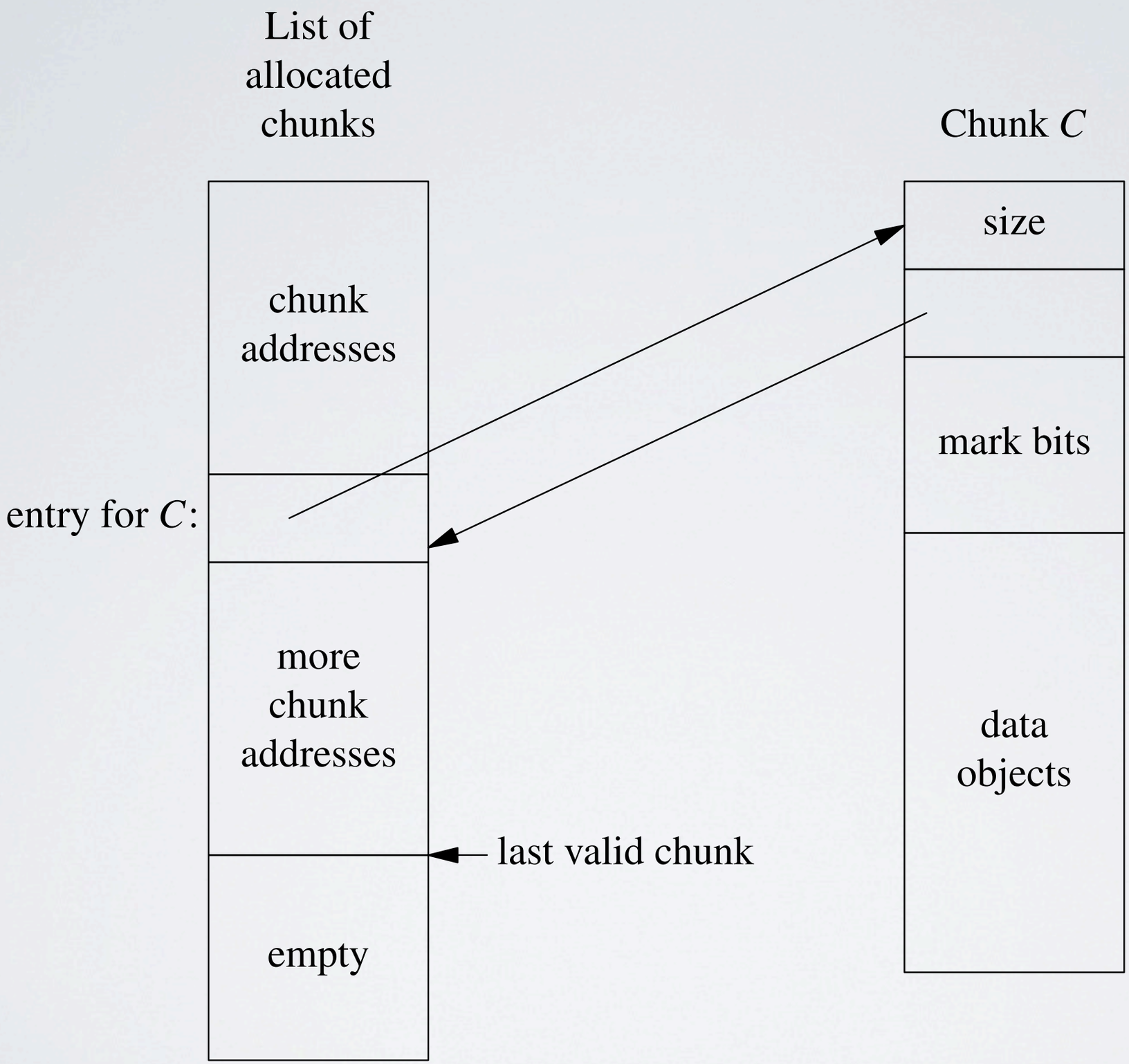
- execution on native hardware
- pay for garbage collection only when needed
- stay compatible with underlying OS and libraries
- no extra memory for tagging
- do not complicate compilers
- support C

APPROACH

- mark-sweep collector
  - first pass: traverse and mark all accessible data
  - second pass: reclaim unmarked objects
- accessible data is never moved

PROBLEMS

- no tags: any data item is potentially a pointer
  - Which data items are valid pointers?
    - must never set mark-bit on anything but valid objects
- misdetection of integers as pointers possible
  - no impact on correctness
  - should be minimized


List of allocated chunks
chunk addresses
entry for C:
more chunk addresses
last valid chunk
empty
Chunk C
size
mark bits
data objects

ASSUMPTION

We assume that for every accessible object there is an accessible pointer to the beginning of the object.

OPTIMIZATION

- lay out memory so small integers are never valid heap pointers
  - automatically on UNIX
- separate object types onto different heaps
  - **atomic:** contains no references (i.e. strings)
  - **composite:** may contain embedded references
  - no need to traverse and clear atomics

EVALUATION

- works
- collection times sufficiently short
- reclamation leaks sufficiently small
- free tool for debugging memory leaks

EVALUATION

- unmodified C programs can use garbage collection
- used for two code generators for the Russell compiler
  - to fix a storage allocation bug
  - to improve performance

DISCUSSION

- one down, two to go?
  - parallelism, robustness
- orthogonality
  - language, framework, runtime, memory management