Compile-Time Construction of Plate-Based Object Graphs

Samir Jindel Logan Brooks

Abstract

Suboptimal memory management continues to cause errors and inefficiency in both languages with manual deallocation and those with automated systems such as a garbage collector. Ideally, a compiler would be able to

- identify mistakes in manual approaches, and
- when possible, replace garbage collection with less costly schemes, such as reference counting or explicit `malloc`-`free` pairs.

We present a system that enables some such checks and optimizations by building a graph representing the structure of objects on the heap. Since the object graph can be arbitrarily large, we use a simplified version of plate models (from Bayesian network literature) to represent and reason about recursive structures.

Representing object graphs with plates

Compressed object graphs (COGs) consist of

- concrete nodes, representing a single or multiple location in memory with type and other information;
- edges, representing possible points-to information for node fields; and
- plate nodes, representing a copy of a template COG.

In the figures, white-filled nodes are heap-allocated nodes. Gray-filled nodes are nodes for stack and global variables. Node `null` represents the abstract "object" of any type at `0x0`. Edge labels correspond to field names. Bounded boxes represent plate node boundaries. An edge that crosses the boundary of a plate, pointing into that plate, should be viewed as pointing to a node in a new copy of the contents of the plate.

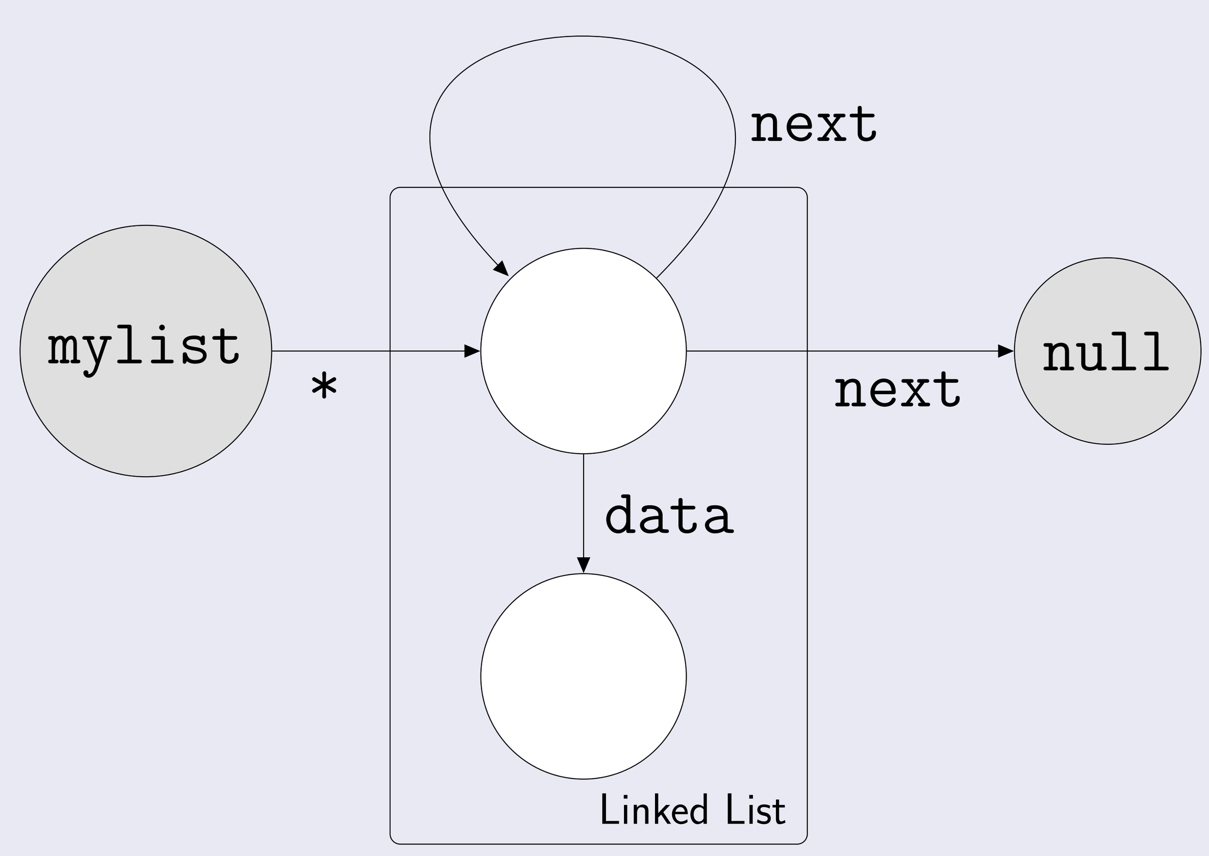


Figure : Plate representation of a pointer, `mylist`, to a linked list; each linked list node points to a unique, "owned", non-null data element.

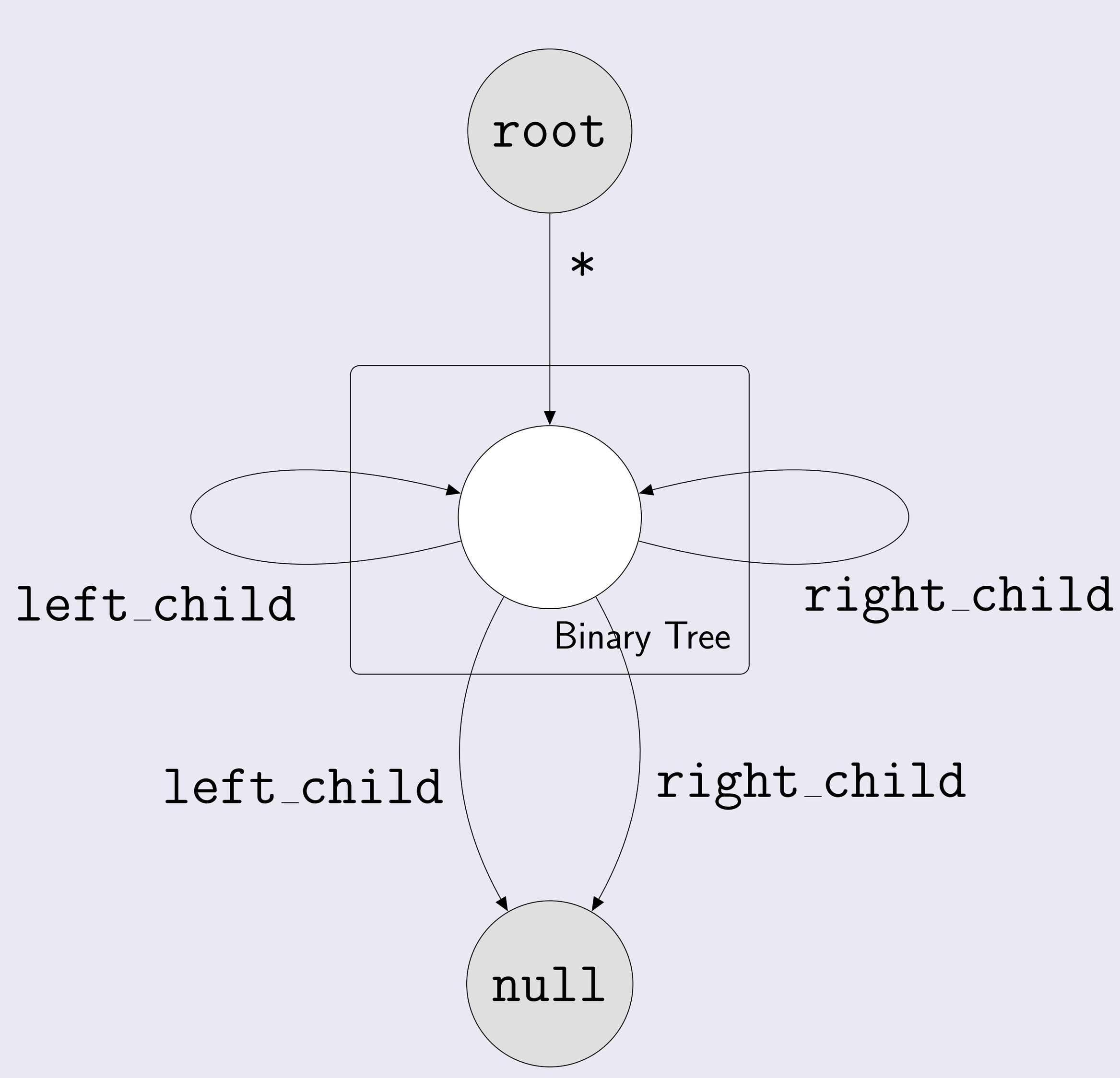


Figure : COG for a binary tree with non-pointer-type data included within each node

Checks, Optimizations, and Information from the Object Graph

Checks:

- Detect possible/definite dereferences of null/freed/uninitialized pointers: "compile-time `valgrind`".

Optimizations:

- Reference-count acyclical recursive structures.
- Compile-time garbage collection on the compressed object graph can be used to generate custom destructors.

Ownership information:

- All incoming pointers to a simplified plate must point to the same "generative node".
- The generative node of a plate owns all other nodes within the plate.

Special Node and Edge Types

Labeled node: node directly accessible in program (stack/args/globals)
Singular node: node with a single possible location in memory
Generative node of a plate: node to which all incoming edges to the plate must point; owns all other nodes in a plate
Fixed node: nodes, such as labeled nodes for the current function and its call-ancestors, that should not be collapsed into a plate node
Singular edge: represents a single possible edge in the full object graph

Some Operations on Compressed Object Graphs

Plate recognition: identification of unlabeled subgraphs that are isomorphic to a known plate pattern; candidate generative node must dominate rest of subgraph
Plate contraction: combination of two matching plate nodes with identical outgoing edges into a single plate node
Plate expansion: generation of fresh unlabeled nodes according to plate node; potentially generates new plate nodes
Edge following: plate expansion to ensure that an edge points to a nonplate node
Graph merge: union of node and edge sets of a set of graphs
Node contraction: combination of two nodes into a nonsingular node, copying incoming and outgoing edges (possibly losing precision)
Graph pruning: removal of heap nodes unreachable from labeled nodes

Relating Two Compressed Object Graphs

Isomorphism: bijection f between all nodes in graph A and all those in graph B preserving

- node labels, singularity, fixation, edge direction, as well as
- edge direction, field labeling, and singularity.

Subisomorphism: isomorphism between a graph A (e.g., a plate template) and a subgraph B' of another graph B

One Object Graph Construction Routine

Algorithm 1 Process assignment $x = y.f$ in COG G

if x is singular **then**
  Remove edges leaving x
end if
for every edge e leaving y with field $\star$ **do**
  FollowEdge(e, G)
  Let s be the concrete node for variable $tail(e)$
  for every edge e' leaving y **do**
    FollowEdge(e', G)
    Copy create an edge from x to $tail(e')$ with field $\star$
  end for
end for

Module Analysis

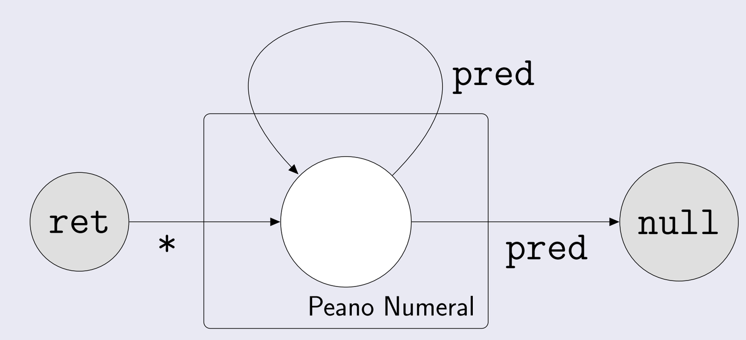
Intraprocedural analysis: For intraprocedural analysis, we use the usual dataflow algorithm. However, COGs do not follow a lattice structure, so the normal approach to proving termination is unsuccessful. Provided that we have a suitable database of plate templates or contract nodes to stay within a size limit, the algorithm should still terminate, though.

Interprocedural analysis: When calling a function, we copy the current object graph, replace the current labeling with labels of the current function arguments and globals, and prune. On exit, we perform a similar graph restriction on the return value. The inputs and outputs are cached for efficiency and as a step towards more complicated interprocedural analysis handling recursive calls.

Example Code and Output

```c
typedef struct Peano {
  struct Peano *pred;
  float data;
} Peano;

int main() {
  Peano *asdf = build();
  return 0;
}
```




```c
Peano *build() {
  Peano *result = (Peano*)malloc(
    sizeof(Peano));
  result->pred = NULL;

  unsigned i = 0;
  do {
    Peano *temp = (Peano*)malloc(
      sizeof(Peano));
    temp->pred = result;
    result = temp;
  } while(++i < 100);

  return result;
}
```

Figure : C program constructing a linked list with data internal to the linked list nodes, in a cons-cell fashion, and corresponding COG returned from `build`