Shape Analysis with Plate Notation

Samir Jindel
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Logan C. Brooks
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

***Abstract*—Shape analysis is the project of extracting information about the run-time shape of data structures in a program through static analysis. In this project we evaluate a new approach to shape analysis that uses an idea from graphical models to represent the object graph of a program compactly at compile time while retaining the relevant information about it.**

Introduction

Motivation

Shape analysis has many potential applications; one is *compile-time garbage collection* (CTGC): approximating the lifetime of objects at compile-time to remove the burden of garbage collection from a run-time system. CTGC systems range from identifying the lifetimes of almost all objects (cite as in Rust), to only objects that have a very short lifetime and can be allocated within a stack frame (cite as in escape analysis).

Related Work

Most previous approaches at compile-time garbage collection have focused either on functional languages [1] or languages that encode lifetime information in their type system though ownership types [2] [3], affine types [4], or similar systems. CTGC is easier for functional languages because they generally disallow heap mutation, making lifetime analysis at compile time significantly easier than in imperative languages. However, approaches designed for these languages are useless for popular imperative languages like Java, C++, or Go. Languages with lifetime information encoded in their type system are also easier to analyze because their type systems invalidate programs that whose object graphs can not be modeled in the type system, but this restricts their applicability to existing code in languages without these type systems. Some other approaches target existing languages, but have serious flaws, such as being unable to identify the lifetimes of objects within recursive structures [5].

Our Approach

Unlike [5], we aim not to insert allocation and deallocation statements directly into the transformed code, but rather to identify which structures in the program are cyclic, so that the rest may be handled with a simple reference-counting system and not a garbage collector.

To this end, we have designed and implemented a whole-program shape analysis system that models the object graph of the program at every point in the program in a compressed fashion that allows us to represent acyclic data structures correctly that requires only modest annotations from the programmer. The information provided by the analysis is more than enough to implement a CTGC using it, although we have not yet implemented it.

Our technical contribution is the shape analysis algorithm and its C++ implementation.

Plate Notation

A graphical model represents a system of random variables by a graph that indicates the relationships between the variables. Plate notation is a notation used within graphical models to indicate that parts of that graph that appear multiple times repeat together. Our approach is similar in spirit to the use of plate notation in graphical models, but the exact semantics of plates are very different in our system. We use a graph to represent the possible configurations of the heap of the program at runtime – each node corresponds to a set of objects in the heap and edges between the nodes correspond to points-to relationships between these objects. Since the object graph of a program can be arbitrarily large, we can not represent it precisely at compile time, so we use plate notation to encode which parts of the graph recur together within a single data structure.

Representing the Object Graph

Our compile-time representation of the object graph of the program, which we term the compressed object graph (COG), has two types of nodes: concrete nodes and plate nodes. A concrete node represents one or more exact objects in the program's heap, whereas a plate node represents a type of subgraph that has been elided in the representation of the object graph.

Each concrete node is labeled with its type and Boolean values indicating whether it *singular*, i.e., whether it corresponds to only one object in the heap or possibly several and a Boolean indicating whether it is *fixed*, i.e., whether it corresponds to a stack variable or an object that is only referenced indirectly by stack variables. Each plate node is labeled with a description of its plate, as described in the next section.

There is only one type of edge in the COG, but it is labeled with the field of the pointer in the struct of its head to which it corresponds and a Boolean flag indicating whether it represents at most one edge in the runtime object graph or several.

We define a *plate* as a COG with an identified root node. A plate corresponds to the internal structure of some pattern

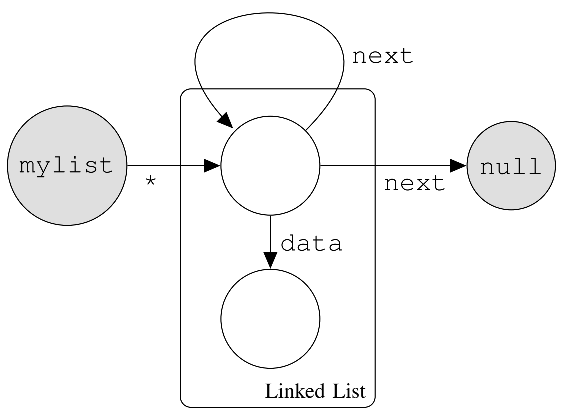

Fig. 1. COG for a non-circular linked list, in which each linked list node l has a non-null pointer to an external data node d, which l owns (i.e., there are no other references to d besides the incoming edge from l)

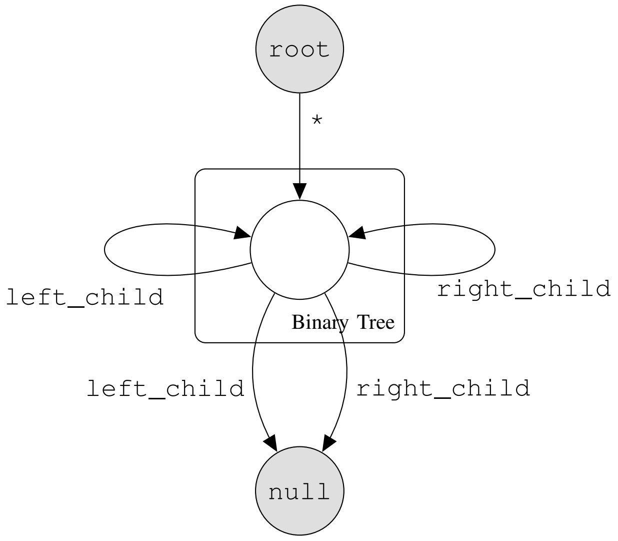

Fig. 2. COG for a binary tree with non-pointer-type data included within each node

that appears arbitrarily often within a program's object graph. Any edge in a COG with a plate node as a tail represents an edge from the head of the edge to the root node of the plate corresponding to the plate node. However, whereas multiple edges with a single concrete node as a tail represent multiple pointers to the *same* object, every edge into a plate node points to a *distinct* copy of the COG represented by the plate. This gives a natural representation to recursive structures by plate nodes in a cycle.

Figure 1 demonstrates how a linked-list of arbitrary length with a particular format is represented within a COG. Here, stack-allocated and global variables are represented using gray-filled nodes, while heap-allocated objects are represented using white-filled nodes. The abstract node `null` represents an "object" of any type located at memory address `0x0`. Plates are drawn using rounded rectangles. In this diagram, linked list nodes have two fields:

- a `next` field, which points either to another linked list node or to null, with no possibility of a cycle; and

$$
\begin{aligned}
\text{inst} ::= {} & x = y \\
& \mid x = y.f \\
& \mid x.f = y \\
& \mid x = \text{new } \tau \\
& \mid y = f(x_1, \cdots, x_n)
\end{aligned}
$$

Fig. 3. Instructions that manipulate the COG

- a `data` field, which is non-null and points to an external data object which is uniquely paired with (i.e., owned by) the linked list node.

Our implementation caches occurrences of the same plate type across multiple program points, so the area within the box will be represented by a single node within the analysis; information about nodes within the box will exist in a separate COG in a global store of plate types. Similarly, Figure 2 demonstrates how a tree is represented.

Each distinct subgraph (up to isomorphism) associated with a plate type is stored in a global index of plate types, and every instance of a subgraph associated with a plate type in a COG at any program point is replace with a corresponding plate node. Since there may be multiple edges leaving a plate from different nodes within the plate, the possible fields leaving a plate node are indexed by (n, f) pairs, where n is a node within the subgraph of the plate type and f is a field leaving n.

There are two important functions that act on the COG alone: CONTRACT, described in Algorithm 1, lowers the size of a COG by identifying copies of plate types that have been registered in the global store and replacing these copies with corresponding plate nodes. It also collapses multiple copies of plate nodes that have identical sets of outgoing edges to concrete nodes, which bounds the maximum number of plate nodes in a COG by 2^C, where C is the number of concrete nodes in the COG.

FOLLOWEDGE, described in Algorithm 2, implements the semantics of plate expansion; it "unfolds" a plate node into a separate set of concrete nodes – this expansion is performed when the program assigns multiple pointers to point to a node which was formerly a plate node.

Last, MERGE, merges the points-to information from one COG into another by copying all edges from the first into the second. Each node in a COG is given a unique label that identifies the same node between COGs in multiple program points, nodes with the same label in the two COGs must represent the program object (if they represent sets, the same node in the new COG will represent the union of their sets).

INTRAPROCEDURAL ANALYSIS

Though C or C++ programs do not necessarily fit the following format, we assume for simplicity in the presentation of our analysis that the instructions relevant to our analysis have one of the forms in Figures 1, 2, or 4, or some other programmer-specified pattern. It should be possible to automatically identify important structures by using information

Algorithm 1 CONTRACT(COG G)

```
Remove nodes in G unreachable from fixed nodes
for every plate type P in the plate registry do
    if the COG of P is isomorphic to a subgraph S of G via map $\mu$ then
        if any nodes in S are fixed
        or $\mu(root(P))$ has more than one incoming edge
        or $\mu(root(P))$ does not dominate S
         then
            continue
        end if
        Let $h$ be the head of the single edge $e$ to $\mu(root(P))$
        Replace $S$ with a single plate node $p$ of plate type $P$
        Add an edge from $h$ to $p$ with field $field(e)$
        Copy outgoing edges from $S$ to $p$.
        Restart contraction
    end if
end for

for every subset $S$ of concrete nodes in $G$ do
    if $S \neq \emptyset$
    and plate nodes $p$ and $q$ have outgoing edges to exactly the concrete nodes in $S$ then
        Merge $p$ and $q$
    end if
end for
for every plate node $p$ with no outgoing edges to concrete nodes do
    Merge $p$ with a neighbor plate node of the same plate type
end for
```

Algorithm 2 FOLLOWEDGE(COG g, Edge e)

```
if $head(e)$ is a concrete node then
    continue
end if
Let $p$ be the plate node at the tail of $e$
Create a copy of $COG(plate_type(p))$ in $g$
Create an edge from $head(e)$ to $p$ with field $field(e)$
for each edge $e'$ leaving $p$ with field $(n, f)$ do
    Create an edge from copy of $n$ to $tail(e)$ with field $f$
end for
Remove $p$ from $g$
```

collected from a custom runtime, or incrementally build up complex plate structures during compile-time analysis. Our implementation makes some assumptions about the use of pointers within the LLVM program under analysis so that the operations we describe on these instructions accurately reflect the implementation.

Each object held by a stack variable is given a unique fixed node in the COG – stack variables which are pointers may then point to objects via the special field $\star$. The procedures for updating a COG per a program instruction are given in Algorithms 3, 4, 5, and 6.

Dataflow Analysis

Even though the COGs do not form a lattice under the MERGE operation described above, we can still apply the dataflow analysis framework, but the analysis is not guaranteed to converge. However, if all object types that may be part of a recursive structure are given plates, then the analysis should converge because objects of those types will be contracted into plates, and the number of plate nodes is bounded by the number of concrete nodes. We check equality between COGs in the dataflow analysis by graph isomorphism, so that iterations through of the dataflow analysis through a loop with the same structure but different exact nodes converges.

Algorithm 3 Process assignment $x = y$ in COG G

```
if $x$ is singular then
    Remove edges leaving $x$
end if
Let $s$ be the concrete node for variable $y$
for every edge $e$ leaving $s$ do
    FOLLOWEDGE(e, G)
    Copy edge $e$ with head $x$
end for
```

Algorithm 4 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
```

INTERPROCEDURAL ANALYSIS

Since most function calls pass data structures of similar shape to the function, we cache COGs of the parameters to functions. This is the only mechanism we have of summarizing

Algorithm 5 Process assignment $x.f = y$ in COG G

```
for every edge $e$ leaving $x$ with field $\star$ do
    Let $z = tail(e)$
    if $z$ is singular and $e$ is the only edge leaving $x$ then
        for every edge $e'$ leaving $z$ with $field(e') = f$ do
            Remove $e'$ from $G$
        end for
    end if
    Create an edge from $z$ to $y$ with field $f$
end for
```

Algorithm 6 Process assignment $x =$ NEW τ in COG G

```
Create a singular, concrete node $z$ of type $\tau$
if $x$ is singular then
    for every edge $e$ leaving $x$ do
        Remove $e$ from $G$
    end for
end if
Create an edge from $x$ to $z$ with field $\star$
```

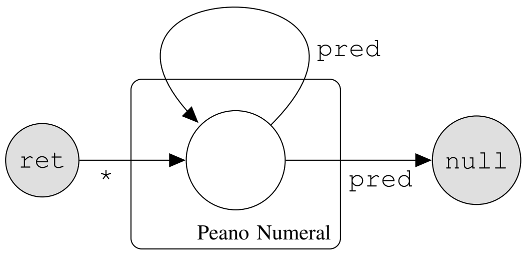

Fig. 4. COG for a Peano numeral, or linked list with non-pointer-type data contained within each node

functions; otherwise, our interprocedural analysis re-analyzes each function for different COGs that relate its parameters; the algorithm for processing function calls is given in Algorithm 7. Results of these analyses are cached, which speeds up analysis of multiple similar calls to the same function, and is a step towards enabling analysis of recursive function calls, provided that the appropriate plate models are provided, or that the object graph is trimmed (losing precision) to fit within a given size limit.

Algorithm 7 Process assignment $y = f(x_1, \cdots, x_n)$ in COG G

```
Mark all nodes in $G$ other than $x_1, \cdots, x_n$ as not fixed
Remove all nodes in $G$ not reachable from fixed nodes
for $i \in [n]$ do
  Let the $i$th parameter to $f$ be $z$
  Assign the node for $x_i$ in $G$ to $z$
end for
Process $f$ with COG $G$, with return value node $w$
Assign the node for $w$ in $G$ to $y$
```

Experimental Setup

We implemented the analysis within the LLVM compiler framework. [6] Our analysis currently targets C and C++ programs that do not use certain language constructs that defeat our analysis. Since it implemented for the LLVM, our implementation may be able to handle LLVM code generated by compilers for other high-level languages. Our analysis requires that the plates necessary to represent the data structures in the program be specified to the analysis.

Evaluation

Our analysis is able to handle several small C and C++ programs, such as the one in Figure 5, on which it correctly identifies linked-list structure shown in Figure 4.

Our analysis is currently unable to handle programs that use virtual functions, pointer arithmetic, recursion, or stack allocated objects. In addition, it is a whole-program analysis, so the entire program must be provided to the analysis within a single LLVM compilation unit.

The lack of support for virtual functions and stack allocated objects is a technical limitation that could be implemented with few changes to the existing system, but were not addressed due to time limitation. Pointer arithmetic defeats our analysis

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

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;
}

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

Fig. 5. C program constructing a linked list with data internal to the linked list nodes, in a cons-cell fashion

for the obvious reason; the inability of our system to handle recursion is its most significant issue.

It is unclear if our approach for finding fixpoints of loops though the dataflow framework could be extended to recursive functions. Certainly recursive functions with a single tail-call could be handled (as they are equivalent to loops), but it is not clear how we could find a fixpoint for recursive functions with multiple recursive calls or groups of mutually recursive functions.

Surprises

We were pleasantly surprised to discover that our system has a similarity to existing ownership type systems: edges that enter plates correspond to “ownership” in these systems, and edges that leave plates correspond to “borrowing”. However, unlike in these systems, not all pointers must be characterized in this way; our analysis can reason about constant-size data structures that do not follow this pattern in an ad hoc manner.

However, we were unpleasantly surprised to discover that determining what plate types a program needs was much more difficult than we expected, and we are not yet sure how to do this. Further, we were disappointed that our checking algorithm seemed rather brittle and ad hoc; while it was worked on the examples we have tried, it does not seem very general, and we are uncertain that it will work on larger programs.

Conclusion

We have designed and implemented a new algorithm for shape analysis that uses ideas from graphical models to represent recursive data structures in a program compactly. Given simple annotations that describe the internal structure of the recursive structures, the analysis is able to compactly model the object graph of the program at every program point, which provides sufficient information for compile-time garbage collection. Logan and I believe we both put approximately equal time into this project.

REFERENCES

[1] N. Mazur, P. Ross, G. Janssens, and M. Bruynooghe, "Practical aspects for a working compile time garbage collection system for mercury," in *Logic Programming*, ser. Lecture Notes in Computer Science, P. Codognet, Ed. Springer Berlin Heidelberg, 2001, vol. 2237, pp. 105–119. [Online]. Available: http://dx.doi.org/10.1007/3-540-45635-X_15

[2] W. Huang, W. Dietl, A. Milanova, and M. Ernst, "Inference and checking of object ownership," in *ECOOP 2012 Object-Oriented Programming*, ser. Lecture Notes in Computer Science, J. Noble, Ed. Springer Berlin Heidelberg, 2012, vol. 7313, pp. 181–206. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31057-7_9

[3] A. Poetzsch-Heffter, K. Geilmann, and J. Schfer, "Infering ownership types for encapsulated object-oriented program components," in *Program Analysis and Compilation, Theory and Practice*, ser. Lecture Notes in Computer Science, T. Reps, M. Sagiv, and J. Bauer, Eds. Springer Berlin Heidelberg, 2007, vol. 4444, pp. 120–144. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-71322-7_6

[4] Mozilla. (2014) The Rust programming language. [Online]. Available: http://www.rust-lang.org

[5] S. Cherem and R. Rugina, "Compile-time deallocation of individual objects," in *Proceedings of the 5th International Symposium on Memory Management*, ser. ISMM '06. New York, NY, USA: ACM, 2006, pp. 138–149. [Online]. Available: http://doi.acm.org/10.1145/1133956.1133975

[6] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673