Data Structure Analysis: A Fast and Scalable Context-Sensitive Heap Analysis

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu

ABSTRACT

Our recent work has focused on developing compiler analyses and transformations which operate at the level of entire logical data structures, rather than individual load/store operations or data elements. This paper describes the scalable heap analysis algorithm that is the foundation of such transformations, which we call Data Structure Analysis. Data Structure Analysis is fully context-sensitive (in the sense that it names memory objects by entire acyclic call paths), is field-sensitive, builds an explicit model of the heap, and is robust enough to handle the full generality of C.

Despite these aggressive features, the algorithm is both extremely fast (requiring 2-7 seconds for C programs in the range of 100K lines of code) and is scalable in practice. It has three features we believe are novel: (a) it incrementally builds a precise program call graph during the analysis; (b) it distinguishes complete and incomplete information in a manner that simplifies analysis of libraries or other portions of programs; and (c) it uses speculative field-senstivity in type-unsafe programs in order to preserve efficiency and scalability. Finally, it shows that the key to achieving scalability in a fully context-sensitive algorithm is through the use of a unification-based approach, a combination that has been used before but whose importance has not been clearly articulated.

1. INTRODUCTION

Extensive research on alias analysis for programs with complex pointer-based data structures has been successful guiding traditional low-level memory optimizations. These transformations rely on disambiguating pairs of memory references and on identifying local and interprocedural side-effects of statements. In contrast, there has been much less success with transformations that apply to entire *instances* of *data structures* such as a lists, heaps, or graphs. Many reasons exist for this disparity, including the possibility of non-type-safe memory accesses in common programming languages (e.g., C and C++), and the potentially high cost of an analysis that can distinguish different instances of a logical data structure.

We have recently developed examples of such transformations, which illustrate both the value and the difficulties of the problem. *Automatic* pool allocation is an automatic (and unusual) transformation that converts a subset of heap-based data structures in a program from using standard heap allocation to using pool-based allocation [13]. It segregates *disjoint instances of logical data structures into separate pools*, e.g., distinct trees created within a single function (and proven disjoint) or distinct graphs created in successive calls to a function. It also internally (and optionally) segregates objects by type. A second example is a static program safety analysis [5] which ensures pointer and heap safety for a large class of type-safe C programs. It does so runtime checks or garbage collection, by building on pool allocation and using the analysis presented in this paper.

Enabling such analyses and transformations requires some powerful analysis capabilities:

- **Full Context-Sensitivity**: Identifying disjoint instances of data structures requires the analysis algorithm to distinguish between heap objects created via different call paths in a program (i.e., naming objects by entire acyclic call paths). Even many partially context-sensitive algorithms do not attempt to distinguish heap objects by call paths [6, 24, 7, 23, 3], which makes them unable to detect this key property. On the other hand, naïve cloning can easily lead to an explosion in the size of the heap representation (because the number of call paths may be exponential in the size of the program), and can make recursion difficult to handle.

- **Field-Sensitivity**: Identifying the internal connectivity pattern of a data structure requires distinguishing the points-to properties of different structure fields. Such "field-sensitivity" is often supported by analyses targeting languages which are type-safe, but is difficult to support efficiently (if at all) in non-type-safe languages (e.g., see [21, 17]).

- **Explicit Heap Model:** Analyzing heap data structures requires constructing an explicit heap model, including objects not directly necessary for identifying aliases. Some common alias analysis algorithms (e.g., Steensgaard's [22] and Andersen's [1] algorithms) build an explicit heap representation, but do not provide any context-sensitivity. Other, more powerful analyses only record alias pairs to determine pointer aliasing properties [6, 2, 10]. Retaining both capabilities is challenging.

Practical alias and pointer analysis algorithms have not attempted to provide the combination of properties described above, because of the potential cost. In contrast, "shape analysis" algorithms are powerful enough to provide this information and more (e.g., enough to identify a particular structure as a "linked-list" or "binary tree" [9, 19]). Shape analysis, however, has so far not proven practical for use in commercial optimizing compilers.

In this work, we present an analysis algorithm called **Data Structure Analysis**, which has been the key foundation for our work on transformations that apply to disjoint instances of logical data structures mentioned above. The algorithm aims to lie somewhere between traditional pointer analyses and more powerful shape analysis algorithms. It provides the three required capabilities listed above, it supports the full generality of C programs, including type-unsafe code, incomplete programs, function pointers, recursion, and `setjmp/longjmp`. We believe it is efficient and scalable enough for use in commercial compilers.

There are three key novel aspects of this algorithm, plus a fundamentally important property that has been used but not articulated before:

(i) The algorithm incrementally discovers an accurate call-graph for the program on-the-fly, using the call graph for parts of the analysis itself. The algorithm is completely non-iterative, visiting each instruction and each call edge only once during the analysis.

(ii) The algorithm explicitly distinguishes between complete and incomplete information, enabling it to be conservative even at intermediate stages of analysis, and allowing it to analyze portions of programs safely.

(iii) The algorithm provides speculative field sensitivity, by assuming that memory objects in the program are type-safe until shown otherwise. This allows the algorithm to be completely field-sensitivity for objects accessed in a type-safe manner (the common case).

(iv) Finally, the property that we believe is fundamental to achieving a scalable "fully context-sensitive" algorithm is the use of a unification-based approach. With this combination, it is extremely unlikely for the analysis representation to grow large, despite using a context-sensitive, field-sensitive representation.[1] This is discussed in Section 3.7.

We show that the worst case time and memory complexity are $\Theta(n\alpha(n) + k\alpha(k)e)$, and $\Theta(fk)$, where n, k, e, and f denote the number of instructions, the maximum size of a data structure graph for a single procedure, the number of edges in the call graph, and the total number of functions. In practice, k is very small, typically on the order of a hundred nodes or less, even in large programs.

We evaluate the algorithm on **35** C programs, showing that the algorithm is extremely efficient in practice (in both performance and memory consumption). This includes programs that contain complex heap structures, recursion, and function pointers. For example, it requires less than **8** seconds of analysis time and about **16MB** of memory to analyze `povray31`, a program consisting of over 130,000 lines of code. Overall, we believe the broader implication of our work may be to show that a fully context sensitive analysis as described here can be practical for significant, large, real-world programs.

The three closest previous algorithms to ours are those by Fähndrich et al. [7], Liang and Harrold [17], and Ruf [18]. All three are context-sensitive, flow-insensitive, and appear roughly comparable to ours in terms of analysis time. As discussed in Section 5, however, none of these three provide the full generality of our work, including the incremental call graph construction (while handling both function pointers and recursion), support for incomplete programs, and support for type-unsafe programs with partial field sensitivity.

```
typedef struct list { struct list *Next;
                      int Data; } list;
int Global = 10;
void do_all(list *L, void (*FP)(int*)) {
  do { FP(&L->Data);
       L = L->Next;
  } while(L);
}
void addG(int *X) { (*X) += Global; }
void addGToList(list *L) { do_all(L, addG); }
list *makeList(int Num) {
  list *New = malloc(sizeof(list));
  New->Next = Num ? makeList(Num-1) : 0;
  New->Data = Num; return New;
}
int main() {    /* X & Y lists are disjoint */
  list *X = makeList(10);
  list *Y = makeList(100);
  addGToList(X);
  Global = 20;
  addGToList(Y);
}
```

Figure 1: C code for running example

2. THE DATA STRUCTURE GRAPH

Data Structure Analysis computes a graph we call the Data Structure Graph (DS graph) for each function in a program, summarizing the memory objects accessible within the function along with their connectivity patterns. Each DS graph node represents a (potentially infinite) set of memory objects and distinct nodes represent disjoint sets of objects, i.e., the graph is a finite, static partitioning of the memory objects. All dynamic objects which may be pointed to by a single pointer variable or field are represented as a single node in the graph.

Some assumptions about the input program representation are necessary for describing our graph representation; other details are described in Section 3.2 [12]. We assume that input programs have a simple type system with structural equivalence, having primitive integer and floating point types of predefined sizes, plus four derived types: pointers, structures (i.e., record types), arrays, and functions. We assume (as in the C language) that only explicit pointer types and integer types of the same size or larger can directly encode a pointer value, and call these *pointer-compatible* types (other values are handled very conservatively in the analysis). For any type τ, $fields(\tau)$ returns a set of field names for the fields of τ, which is a single degenerate field name if τ is a scalar type (field names are assumed to be unique to a type). An array type of known size k may be repre-

[1]We describe a graph-size-limiting heuristic to detect and avoid this rare case, but this has never been required in practice.

sented either as a structure with k fields or by a single field; an unknown-size array is always represented as the latter. Other assumptions about the input program representation are described in Section 3.2.

We also assume a load/store program representation in which virtual registers and memory locations are distinct, it is not possible to take the address of a virtual register, and virtual registers can only represent scalar variables (i.e., integer, floating point, or pointer). Structures, arrays, and functions are strictly memory objects and are accessed only through load, store, and call instructions. All arithmetic operations operate on virtual registers. Memory is partitioned into heap objects (allocated via a `malloc` instruction), stack objects (allocated via an explicit stack allocation instruction named `alloca`, similar to malloc), and global objects (global variables and functions).

The DS graph for a function is a finite directed graph represented as a tuple $DSG(F) = \langle N, E, E_V, C\rangle$, where:

- N is a set of nodes, called "DS nodes". DS nodes have several attributes described in Section 2.1 below.
- E is a set of edges in the graph. Formally, E is a function of type $\langle n_s, f_s\rangle \rightarrow \langle n_d, f_d\rangle$, where $n_s, n_d \in N$, $f_s \in fields(T(n_s))$ and $f_d \in fields(T(n_d))$, and $T(n)$ denotes type information computed for the objects of n as explained below. E is a function because a source field can have only a single outgoing edge. Note that the source and target of an edge are both *fields* of a DS node.
- E_V is a function of type $vars(f) \rightarrow \langle n, f\rangle$, where $vars(f)$ is the set of virtual registers in function f. Conceptually, $E_V(v)$ is an edge from register v to the target field $\langle n, f\rangle$ pointed to by v, if v is of pointer-compatible type.
- C is a set of "call nodes" in the graph, which represent unresolved call sites in the context of the current function. Each call node $c \in C$ is a $k+2$ tuple: $(r, f, a_1, \ldots, a_k)$, where every element of the tuple is a node-field pair $\langle n, f\rangle$. r and f respectively denote the value returned by the call (if it is pointer-compatible) and the function(s) being called. $a_1 \ldots a_k$ denote the pointer-compatible values passed as arguments to the call (other arguments are not represented). Conceptually, each tuple element can also be regarded as a points-to edge in the graph.

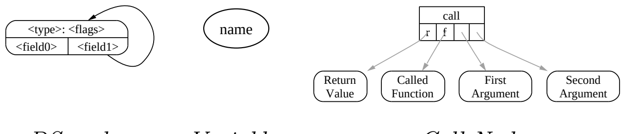


Figure 2: Graph Notation

To illustrate the DS graphs and the analysis algorithm, we use the code in Figure 1 as a running example. This example creates and traverses two disjoint linked lists, using iteration, recursion, function pointers, a pointer to a subobject, and a global variable reference. Despite the complexity of the example, Data Structure Analysis is able to prove that the two lists `X` and `Y` are disjoint (the final DS graph computed for `main` is shown in Figure 10).

To illustrate the DS graphs computed by various stages of our algorithm, we render DS graphs using the graphical notation shown in Figure 2. Figure 3 shows an example graph computed for the `do_all` and `addG` functions, before any interprocedural information is applied. The figure includes an example of a call node, which (in this case) calls the function pointed to by `FP`, passing the memory object pointed to by `L` as an argument, and ignores the return value of the call.

2.1 Graph Nodes and Fields

The DS nodes in a DS graph are responsible for representing information about a set of memory objects corresponding to that node. Each node n has three pieces of information associated with it:

- $T(n)$ identifies a language-specific type for the memory objects represented by n. Section 2.1.4 describes how this is computed for nodes representing multiple incompatible memory objects.
- $G(n)$ represents a (possibly empty) set of global objects, namely, all those represented by node n.
- $flags(n)$ is a set of flags associated with node n. There are eight possible flags (H,S,G,U, M,R, C and O), defined below.

The type information $T(n)$ determines the number of fields and outgoing edges in a node. A node can have one outgoing edge for each pointer-compatible field in $T(n)$. An incoming edge can point to an arbitrary field of the node (e.g., the "`&L->Data`" temporary in Figure 3 points to the integer field), but not to any other byte offset. Section 2.1.4 describes how type-unsafe code using pointers to arbitrary byte offests are handled.

The globals $G(n)$ represented by each node can be used to find the targets of function pointers, both by clients of Data Structure Analysis and to incrementally construct the call-graph during the analysis.

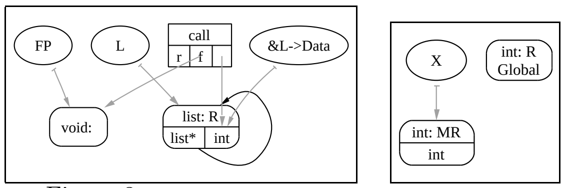


Figure 3: Local DSGraphs for `do_all` and `addG`

2.1.1 Memory Allocation Classes

The **'H'**, **'S'**, **'G'** and **'U'** flags in $flags(n)$ are used to distinguish four classes of memory objects: **H**eap-allocated, **S**tack-allocated, **G**lobals (which includes functions), and **U**nknown objects. Multiple flags may be present in a single DS node, if, for example, analysis finds a pointer which may point to either a heap object or a stack object. Memory objects are marked as Unknown when the instruction creating it is not identifiable, e.g., when a constant value is cast to a pointer value (for example, to access a memory-mapped hardware device), or when unanalyzable address arithmetic is found (these cases occur infrequently in portable programs). Nodes representing objects created in an external, unanalyzed function are *not* marked **'U'**, but are treated as "missing information" as described below.

2.1.2 Mod/Ref information

Our analysis keeps track of whether a particular memory object has been **M**odified or **R**ead within the current scope of analysis, and this is represented via the **'M'** and

'R' flags. For example, in the `do_all` function, the statement "`L = L->Next;`" reads a pointer element from the node pointed to by `L`, which causes the **'R'** flag to be set in $flags(node(E_V(\texttt{L})))$ as shown in Figure 3. Mod/Ref information is useful to a variety of client analyses.

2.1.3 Aggressive analysis with missing information

Practical algorithms must correctly handle incomplete programs: those where code for some functions are unavailable, or where the "program" is actually a library of code without information about the clients. In order to allow an aggressive analysis even under such situations, each DS node tracks whether there may be information missing from it.

For example, in Figure 3, Data Structure Analysis does not yet know anything about the incoming `L` and `FP` arguments because it hasn't performed interprocedural analysis. Inside this function, it can determine that `L` is treated as a `list` object (the construction algorithm looks at how pointers are used, not what their declared types are), that it is read from, and what nodes each variable points to. However, it can not know whether the information it has for this memory object is correct in a larger scope. For example, the `FP` and `L` arguments are speculatively represented as different objects, even though they might actually be aliased to each other when called from a particular call site.

To handle such situations, Data Structure Analysis computes which nodes in the graph are "complete," and marks each one with the **C**omplete flag[2]. If a node is not marked complete, the information calculated for the DS node represents partial information and must be treated conservatively. In particular, the node may later be assigned extra edges, extra flags, a different type, or may even end up merged with another incomplete node in the graph. For example, from the graph in Figure 3 an alias analysis algorithm must assume that `L` and `FP` may alias. Nevertheless, other nodes in such a graph may be complete and such nodes will never be merged with any other node, allowing clients to obtain useful information from graphs with partial information.

This capability is the key to the incremental nature of our algorithm: Because nodes keep track of which information is final, and which is still being created, the graphs constructed by our algorithm are *always* conservatively correct, even during intermediate steps of the analysis.

2.1.4 Field-sensitivity with and without type-safety

A particularly important benefit of the "Complete" flag is that it allows DS Analysis to efficiently provide field-sensitive information *for the type-safe subsets of programs*. This is important because field-sensitivity for type-unsafe structure types can be very expensive [21], but in fact we observe that most portable C code is completely (or mostly) type-safe. The complete flag allows DS analysis to assume speculatively that all access to a node are type-safe, until an access to the node is found which conflicts with the other accesses. Because a node is not marked complete as long as there are potentially unprocessed accesses, this is safe.

DS Analysis provides field-sensitive information by associating a language-specific type, $T(n)$, with each DS node n, and keeping track of a distinct outgoing edge for each pointer element of the type. If all accesses to all objects at the node use a consistent type τ, then $T(n) = \tau$.[3]

[2]This is somewhat similar to the "inside nodes" of [23].

[3]As Section 3 describes, type information is inferred only at actual accesses rather than from the declared types for variables, so that common idioms such as casting a pointer to `void*` and back do not cause a loss of precision.

If operations using incompatible types (as defined in Section 3) are found, a type-safety violation is possible with one of the objects at the node. If this occurs, the type for the node is assumed to be an unsized array of bytes ($T(n) =$ `void*`), and the fields and edges of the node are "c**O**llapsed" into a single field with at most one outgoing edge, using the following algorithm:

```
collapse(dsnode $n$)
  cell $e = \langle$null, 0$\rangle$                        // null target
  $\forall f \in fields(T(n))$
    e = mergeCells($e, E(\langle n, f\rangle)$)// merge old target with $e$
    remove field $f$                   // remove old edge
  $T(n) =$ void*                        // reset type information
  $E(\langle n, 0\rangle) = e$                      // new edge from field 0
  $flags(n) = flags(n) \cup$ 'O'   // mark node Collapsed
```

In the pseudo-code, a "cell" is a $\langle$node,field$\rangle$ pair, used as "sources" of edges in the DS graphs. The function "*mergeCells*(c_1, c_2)" (described in the next section) merges the cells c_1 and c_2 and therefore the nodes pointed to by those cells. This ensures that the targets of the two cells are now exactly equal. Because the above algorithm merges all outgoing edges from the node, the end result is the same as if field-sensitivity were never speculated for node n. If a node has been collapsed (ie, $\mathbf{O} \in flags(n)$), it is always treated in this safe, but field-insensitive, manner.

3. CONSTRUCTION ALGORITHM

DS graphs are created and refined in a three step process. The first phase constructs a DS graph for each function in the program, using only intraprocedural information (a "local" graph). Second, a "Bottom-Up" analysis phase is used to eliminate incomplete information due to callees in a function, by incorporating information from callee graphs into the caller's graph (creating a "BU" graph). The final "Top-Down" phase eliminates incomplete information due to incoming arguments by merging caller graphs into callees (creating a "TD" graph). The BU and TD phases operate on the "known" Strongly Connected Components (SCCs) in the call graph.

Two properties are important for understanding how the analysis works in the presence of incomplete programs, and how it can incrementally construct the call graph even though it operates on the SCCs of the graph. First, the DS graph for a function is correct even if only a subset of its potential callers and potential callees have been incorporated into the graph (i.e., the information in the graph can be used safely so long as the limitations on nodes without **'C'** flags are respected, as described in Section 2.1.3). Intuitively, the key to this property simply is that a node must not be marked complete until it is known that all callers and callees potentially affecting that node have been incorporated into the graph. Second, the result of two graph inlining operations at one or two call sites is independent of the order of those operations. This follows from a more basic property that the order in which a set of nodes is merged does not affect the final result. These properties are formalized in Section 3.3.4.

3.1 Primitive Graph Operations

Data Structure Analysis is a flow-insensitive algorithm which uses a unification-based memory model, similar to

Steensgaard's algorithm [22]. The algorithm uses several primitive operations on DS graphs, shown in Figure 4. These operations are used in the algorithm to merge two cells, merge two nodes while aligning fields in a specified manner, to inline a callee's graph into a caller's graph at a particular call site, and vice versa. The latter two operations are described later in this section.

The fundamental operation in the algorithm is *mergeCells*, which merges the two target nodes specified. This requires merging the type information, flags, globals, outgoing edges of the two nodes, and moving the incoming edges to the resulting node. If the two fields have incompatible types (e.g., $T(n_1) = \texttt{int}$, $f_1 = 0$, $T(n_2) = \{\texttt{int}, \texttt{short}\}$, $f_2 = 1$), or if the two node types are compatible but the fields are misaligned (e.g., $T(n_1) = T(n_2) = \{\texttt{int}, \texttt{short}\}$, $f_1 = 0$, $f_2 = 1$), the resulting node is first collapsed as described in Section 2.1.4, before the rest of the information is merged. Merging the outgoing edges causes the target node of the edges to be merged as well (if the node is collapsed, the resulting node for n_2 will have only one outgoing edge which is merged with all the out-edges of n_1). To perform this recursive merging of nodes efficiently, the merging operations are implemented using Tarjan's Union-Find algorithm.

```
(Merge two cells of same or different nodes; update $n_2$, discard $n_1$)
Cell **mergeCells**(Cell $\langle n_1, f_1\rangle$, Cell $\langle n_2, f_2\rangle$,)
  if (IncompatibleForMerge($T(n_1), T(n_2), f_1, f_2$))
    collapse $n_2$ (i.e., merge fields and out-edges)
  union flags of $n_1$ into flags of $n_2$
  union globals of $n_1$ into globals of $n_2$
  merge target of each out-edge of $\langle n_1, f_j\rangle$ with
        target of corresponding field of $n_2$
  move in-edges of $n_1$ to corresponding fields of $n_2$
  destroy $n_1$
  return $\langle n_2, 0\rangle$ (if collapsed) or $\langle n_2, f_2\rangle$ (otherwise)

(Clone $G_1$ into $G_2$; merge corresponding nodes for each global)
**cloneGraphInto**($G_1, G_2$)
  $G_{1c}$ = make a copy of graph $G_1$
  Add nodes and edges of $G_{1c}$ to $G_2$
  for (each node $N \in G_{1c}$)
    for (each global $g \in G(N)$)
      merge $N$ with the node containing $g$ in $G_2$

(Clone callee graph into caller and merge arguments and return)
**resolveCallee**(Graph $G_{callee}$, Graph $G_{caller}$,
                  Function $F_{callee}$, CallSite $CS$)
  cloneGraphInto($G_{callee}, G_{caller}$)
  clear 'S' flags on cloned nodes
  resolveArguments($G_{caller}$, $F_{callee}$, $CS$)

(Clone caller graph into callee and merge arguments and return)
**resolveCaller**(Graph $G_{caller}$, Graph $G_{callee}$,
                  Function $F_{callee}$, CallSite $CS$)
  cloneGraphInto($G_{caller}, G_{callee}$)
  resolveArguments($G_{callee}$, $F_{callee}$, $CS$)

(Merge arguments and return value for resolving a call site)
**resolveArguments**(Graph $G_{merged}$, Function $F_C$, CallSite $CS$)
  mergeCells(target of $CS[1]$, target of return value of $F_C$)
  for ($1 \le i \le min$(Numformals($F_C$), NumActualArgs($CS$))
    mergeCells(target of arg $i$ at CS, target of formal $i$ of $F_C$)
```

Figure 4: Primitive operations used in the algorithm

3.2 Local Analysis Phase

The goal of the local analysis phase is to compute a *Local DS graph* for each function, without any information about callers and callees. This is the only phase that inspects the actual program representation: the other two phases operate solely on DS graphs.

The local DS graph for a function F is computed as shown in Figure 5. We present this analysis in terms of a minimal language which is still as powerful as C. The assumptions about the type system and memory model in this language were described in Section 2[4].

```
(Compute the local DS Graph for function $F$)
**LocalAnalysis**(function $F$)
  Create an empty graph
  $\forall$ virtual registers $R$, $E_V(R)$ = makeNode($T(R)$)
  $\forall$ globals $X$ (variables and functions) used in $F$
    $N$ = makeNode($T(X)$)); $G(N) \cup= X$; $flags(N) \cup=$ 'G'
  $\forall$ instruction $I \in F$ : case $I$ in:

    X = malloc ...:                    (heap allocation)
      $E_V(X) = makeNode$(void)
      $flags(node(E_V(X))) \cup=$ 'H'

    X = alloca ...:                    (stack allocation)
      $E_V(X) = makeNode$(void)
      $flags(node(E_V(X))) \cup=$ 'S'

    X = *Y:
      mergeCells($E_V(X)$, $E(E_V(Y))$)
      $flags(node(E_V(X)) \cup=$ 'R'

    *Y = X:
      mergeCells($E_V(X)$, $E(E_V(Y))$)
      $flags(node(E_V(X)) \cup=$ 'M'

    X = &Y->Z:                         (address of struct field)
      $\langle n, f\rangle$ = updateType($E_V(Y)$, $typeof(*Y)$)
      $f' = 0$, if $n$ is collapsed; $field(field(n, f), Z)$ otherwise
      mergeCells($E_V(X), \langle n, f'\rangle$)

    X = &Y[idx]:                       (address of array element)
      $\langle n, f\rangle$ = updateType($E_V(Y)$, $typeof(*Y)$)
      mergeCells($E_V(X), \langle n, f\rangle$)

    return X:                          (return pointer-compatible value)
      mergeCells($E_V(\pi), E_V(X)$)

    X = ($\tau$) Y:                    (value-preserving cast)
      mergeCells($E_V(X), E_V(Y)$)

    X = Y($Z_1$, $Z_2$, ... $Z_n$):    (function call)
      callnode $c$ = new callnode
      $C \cup= c$
      mergeCells($E_V(X), c[1]$)
      mergeCells($E_V(Y), c[2]$)
      $\forall i \in \{1...n\}$: mergeCells($E_V(Z_i), c[i+2]$)

    (Otherwise) X = Y op Z:            (all other instructions)

      mergeCells($E_V(X), E_V(Y)$)
      mergeCells($E_V(X), E_V(Z)$)
      $flags(node(E_V(X))) \cup=$ 'U'
      collapse($node(E_V(X))$)

  MarkCompleteNodes()
```

Figure 5: The LocalAnalysis function

```
(Create a new, empty node of type $\tau$)
**makeNode**(type $\tau$)
  n = new Node(type = $\tau$, flags = $\phi$, globals = $\phi$)
  $\forall$ $f$ $\in$ $fields(\tau)$, $E(n, f) =< null, 0 >$
  return $n$

(Merge type of field $\langle n, f\rangle$ with type $\tau$. This may
 collapse fields and update in/out edges via mergeCells())
**updateType**(cell $\langle n, f\rangle$, type $\tau$)
  if ($\tau \neq$ void $\wedge$ $\tau$ $\neq$ $typeof(\langle n, f\rangle)$)
    $m$ = makeNode($\tau$)
    return mergeCells($\langle m, 0\rangle, \langle n, f\rangle$))
  else return $\langle n, f\rangle$
```

Figure 6: makeNode and updateType operations

The "*LocalAnalysis*" first creates an empty node as a target for every pointer-compatible virtual register (entering

[4]We assume that the functions $E(X)$ and $E_V(X)$ return a new, empty node with the type of X (by invoking makeNode($typeof(X)$)) when no previous edge from the cell or variable X existed. For example, in Figure 7(a), the incoming argument L points to such a node. We also abuse the notation by using $E(X) = \ldots$ or $E_V(X) = \ldots$ to change what X points to.

them in the map E_V), and creates a separate node for every global variable. The analysis then does a linear scan over the instructions of the function, creating new nodes at `malloc` and `alloca` operations, merging edges of variables at assignments and the return instruction, and updating type information at selected operations. The type of a cell, $E_V(Y)$, is updated only when Y is actually *used* in a manner that interprets its type, viz., at a dereference operation on Y (for a load or store) and when indexing into a structure or array pointed to by Y. `malloc`, `alloca`, and `cast` operations simply create an node of `void` type. Structure field accesses adjust the incoming edge to point to the addressed field (which is a noop if the node is collapsed). Indexing into array objects is ignored, i.e., arrays are treated as having a single element. `return` instructions are handled by creating a special π virtual register which is used to capture the return value.

Function calls result in a new call node being added to the DS graph, with entries for the value returned, the function pointer (for both direct and indirect calls). For example, the local graph for `addGTList` in Figure 7(a) shows the call node created for the call to function `do_all`. Note that an empty node is created and then merged using *mergeCells* for each entry in order to correctly merge type information, since the argument type may not match the declared type for the formal argument or return value.

Finally, if any other instruction is applied to a pointer-compatible value, (e.g., a cast from a pointer to an integer smaller than the pointer, or integer arithmetic), any nodes pointed to by operands and the result are collapsed and the **U**nknown flag is set on the node[5].

The final step in the Local graph construction is to calculate which DS nodes are **C**omplete. For a Local graph, nodes reachable from a formal argument, a global, passed as an argument to a call site, or returned by a function call may not be marked complete. This reflects the fact that the local analysis phase does not have any interprocedural information. For example, in Figure 7(a), neither of the nodes for for the arguments to `do_all` are marked **'C'**.

3.3 Bottom-Up Analysis Phase

The Bottom-Up (BU) analysis phase refines the local graph for each function by incorporating interprocedural information from the callees of each function. The result of the BU analysis is a graph for each function which summarizes the total effect of calling that function (imposed aliases and mod/ref information) without any calling context information. It computes this graph by cloning the BU graphs of all *known* callees into the caller's Local graph, merging nodes pointed to by corresponding formal and actual arguments. We first describe a single graph inlining operation, and then explain how the call graph is discovered and traversed.

Consider a call to a function F with formal arguments $f_1, \ldots, f_n$, where the actual arguments passed are $a_1, \ldots, a_n$. The function *resolveCallee* in Figure 4 shows how such a call is processed in the BU phase. We first copy the BU graph for F, clearing all **S**tack node markers since stack objects of a callee are not legally accessible in a caller. We then merge the node pointed to by each actual argument a_i of pointer-compatible type with the copy of the node pointed to by f_i. If applicable, we also merge the return value in the call node with the copy of the return value node from the callee. Note that any unresolved call nodes in F's BU graph are copied into the caller's graph, and all the objects representing arguments of the unresolved call in the callee's graph are now represented in the caller as well.

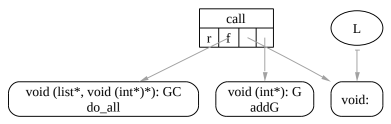
[5]In our compiler [14], type-safe pointer arithmetic is represented with the `&Y->Z` or `&Y[idx]` operations.

3.3.1 Basic Analysis Without Recursion

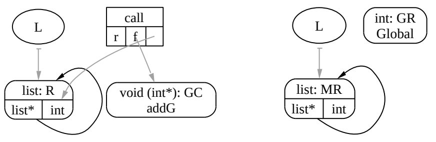
The complete Bottom-Up algorithm for traversing calls is shown in Figure 8. but we explain it for four different cases. In the simplest case of a program with only direct calls to non-external functions, no recursion, and no function pointers, the call nodes in each DS graph implicitly define the entire call graph. The BU phase simply has to traverse this acyclic call graph in post-order (visiting callees before callers), cloning and inlining graphs as described above.

To support programs that have function pointers and external functions (but no recursion), we simply restrict our post-order traversal to only process a call-site if its function pointer targets a **C**omplete node (i.e, its targets are are fully resolved, as explained in §2.1.3), *and* all potential callees are non-external functions (line 1 in the Figure).

Such a call site may become resolved if the function passed to a function pointer argument becomes known. For example, the call to *FP* cannot be resolved within the function `do_all`, but will be resolved in the BU graph for the function `addGToList`, where we conclude that it is a call to *addG*. We clone and merge the indirect callee's BU graph into the graph of the function where the call site became resolved, merging actual and formal arguments as well as return values, using *resolveCallee* just as before (line 2 in the figure). This technique of resolving call nodes as their function pointer targets are completed effectively discovers the call-graph on the fly, and we record the call graph as it is discovered.



(a) Local `addGToList` graph



(b) After inlining `do_all` (c) Final BU graph

Figure 7: Construction of the BU DS graph for `addGToList`

Note that the BU graph of the function containing the original call still has the unresolved call node. We do not re-visit previously visited functions in each phase, but that call node will eventually be resolved in the top-down phase. The BU graph for the function where the call was resolved now fully incorporates the effect of the call. This property of the algorithm is crucial, and is discussed further in Section 3.3.4 below. For example, inlining the BU graph of `addG` into that of `addGToList` yields the finished graph shown in Figure 7(c). The **M**odified flag in the node pointed to by L is obtained from the node $E_V(X)$ from `addG` (Figure 3), which is merged with the second argument node inlined from `do_all`. This graph for `addGToList` is identical to that which would have been obtained if `addG` was first inlined into `do_all` (eliminating the call node) and the resulting graph was then inlined into `addGToList`.

After the cloning and merging is complete for a function in the SCC, we identify new complete nodes (Section 3.2) (line 5) and remove unreachable nodes from the graph (line 6). The latter are created because copying and inlining callee graphs can bring in excess nodes not accessible within the current function (and therefore not accessible in any of its callers as well). This includes non-global nodes not reachable from any virtual register, global node, or call node.

3.3.2 Recursion without Function Pointers

Our strategy for handling recursion is essentially to apply the bottom-up process described above but on Strongly Connected Components (SCCs) of the call graph, handling each multi-node SCC separately. The key difficulty is that call edges are not known beforehand, and instead are discovered incrementally by the algorithm. The overall Bottom-Up analysis algorithm is shown in Figure 8. It uses an adaptation of Tarjan's linear-time algorithm to find and visit Strongly Connected Components (SCCs) in the call graph in postorder [20].

Assume first that there are only direct calls, i.e., the call graph is known. For each SCC, all calls to functions outside the SCC are first cloned and resolved as before (these functions will already have been visited because of the postorder traversal over SCCs). Once this step is complete, all of the functions in the SCC have empty lists of call sites, except for intra-SCC calls and calls to external functions (the latter are simply ignored throughout). In an SCC, each function will eventually need to inline the graphs of all other functions in the SCC at least once (either directly or through the graph of a callee). A naive algorithm can produce an exponential number of inlining operations, and even a careful enumeration can require $O(n^2)$ inlining operations in complex SCCs (which we have encountered in some benchmarks). Instead, because there are an infinite number of call paths through the SCC, we choose to completely ignore intra-SCC context-sensitivity. We simply merge the partial BU graphs of all functions in the SCC, resolving all intra-SCC calls in the context of this single merged graph.

3.3.3 Recursion with Function Pointers

The final case to consider is a recursive program with indirect calls. The difficulty is that some indirect calls may induce cycles in the SCC, but these call edges will not be discovered until the indirect call is resolved. We make a key observation, based on the properties described earlier, that yields a simple strategy to handle such situations: some call edges of an SCC can be resolved *before discovering that they form part of an SCC*. When the call site "closing the cycle" is discovered (say in the context of a function F_0), the effect of the complete SCC will be incorporated into the BU graph for F_0 though not the graphs for functions handled earlier.

Based on this observation, we have slightly adapted Tarjan's algorithm to revisit partial SCCs as they are discovered (but visiting only unresolved call sites). After the current SCC is fully processed (i.e., after step (5) in Figure 8), we check whether the SCC graph contains any newly inlined call nodes that are now resolvable. If so, we reset the *Val* entries for all functions in the SCC, which are used in *TarjanVisitNode* to check if a node has been visited. This causes

```
    BottomUpAnalysis(Program $P$)
      $\forall$ Function $F \in P$
.       $BUGraph\{F\} = LocalGraph\{F\}$
.       Val[$F$] = 0; NextID = 0
      while ($\exists$ unvisited functions $F \in P$)    (visit main first if available)
        TarjanVisitNode($F$, new Stack)

    TarjanVisitNode(Function $F$, Stack Stk)
      NextID++; Val[F] = NextID; MinVisit = NextID; Stk.push(F)
      $\forall$ call sites $C \in BUGraph\{F\}$
        $\forall$ known non-external callees $F_C$ at $C$
          if (Val[$F_C$] == 0)        ($F_C$ unvisited)
            TarjanVisitNode($F_C$, S)
          else MinVisit = min(MinVisit, Val[$F_C$])
        if (MinVisit == Val[$F$])      (new SCC at top of Stack)
          SCC S = { $N$: $N = F \lor N$ appears above $F$ on stack }
          $\forall F \in S$:  Val[$F$] = MAXINT; Stk.pop($F$)
          ProcessSCC(S, Stk)

    ProcessSCC(SCC $S$, Stack Stk)
      $\forall$ Function $F \in S$
(1)     $\forall$ resolvable call sites $C \in BUGraph\{F\}$       (see text)
          $\forall$ known callees $F_C$ at $C$
            if ($F_C \notin S$)        (Process funcs not in SCC)
(2)           ResolveCallee(BUGraph{$F_C$}, BUGraph{$F$}, $F_C$, $CS$)
(3)   SCCGraph = BUGraph{$F_0$}, for some $F_0 \in S$
      $\forall$ Function $F \in S, F \neq F_0$    (Merge all BUGraphs of SCC)
        cloneGraphInto(BUGraph{F}, SCCGraph)
        BUGraph{F} = SCCGraph
(4)   $\forall$ resolvable call sites $C \in SCCGraph$        (see text)
        $\forall$ known callees $F_C$ at $C$    (Note: $F_C \in S$)
          ResolveArguments(SCCGraph, $F_C$, $CS$)
(5)   MarkCompleteNodes() - Section 3.2
(6)   remove unreachable nodes
(7)   if (SCCGraph contains new resolvable call sites)
        $\forall F \in S$:  $Val[F] = 0$      (mark unvisited)
        TarjanVisitNode($F_0$, Stk), for some $F_0 \in S$   (Re-visit SCC)
```

Figure 8: Bottom-Up Closure Algorithm

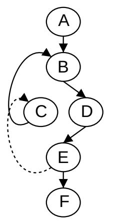
all the nodes in the *current* SCC to be revisited, but only the new call sites are processed (since other resolvable call sites have already been resolved, and will not be included in steps (1) and (4)).

For example, consider the recursive call graph shown in Figure 9(a), where the call from E to C is an indirect call. Assume this call is resolved in function D, e.g., because D passes C explicitly to E as a function pointer argument. Since the edge $E \rightarrow C$ is unknown when visiting E, Tarjan's algorithm will first discover the SCCs { F }, { E }, and then { D } (Figure 9(c)). Now, it will find a new call node in the graph for D, find it is resolvable as a call to C, and mark D as unvisited (Figure 9(b)). This causes Tarjan's algorithm to visit the "phantom" edge $D \rightarrow C$, and therefore to discover the partial SCC { B, D, C }. After processing this SCC, no new call nodes are discovered. At this point, the BU graphs for B, D and C will all correctly reflect the effect of the call from E to C, but the graph for E will not[6]. The top-down pass will resolve the call from E to C (within E) by inlining the graph for D into E.

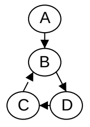
Note that even in this case, the algorithm only resolves each callee at each call site once: no iteration is required, even for SCCs induced by indirect calls.

The graph of Figure 10 shows the BU graph calculated for the `main` function of our example. This graph has disjoint subgraphs for the lists pointed to by X and Y. These were proved disjoint because we cloned and then inlined the BU

[6]Nor should it. A different caller of E may cause the edge to be resolved to a different function, thus the BU graph for E does not include information about a call edge which is not necessarily present in all calling contexts.



(a) Recursive Call Graph (indirect call is dotted)



(b) Call Node Edges, After inlining F & E

1. { F }
2. { E }
3. { D }: mark unvisited
4. { B, D, C }
5. { A }

(c) SCC visitation order

Figure 9: Handling recursion due to an indirect call in the Bottom-Up phase

graph for each call to `addGToList()`. This shows how the combination of context sensitivity with cloning can identify disjoint data structures, even when complex pointer manipulation is involved.

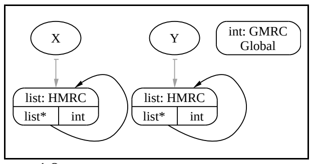


Figure 10: Finished BU graph for `main`

3.3.4 Correctness of the Bottom-Up Algorithm

A detailed proof of the correctness of the Bottom-Up algorithm is lengthy and is far beyond the scope of this paper. The proof is under development and will be made available online as an updated version of a technical report in the near future [15]. We limit ourselves here to noting some important basic properties of the algorithm.

Lemma 1: Given three cells, c_1, c_2, and c_3, the result of $\mathit{mergeCells}(c_1, \mathit{mergeCells}(c_2, c_3))$ is identical to the result of $\mathit{mergeCells}(\mathit{mergeCells}(c_1, c_2), c_3)$.

This property follows directly from the fact that we use a unification-based algorithm [22, 21], and the *mergeCells* operations are essentially a union operation on attributes of a node followed by merging the corresponding outgoing edges. Furthermore, if either merge causes the node to be collapsed, the final node will be collapsed in both sequences.

Lemma 2: Suppose function F_0 calls F_1 at call site C_1, and F_1 calls F_2 at call site C_2. Let G_0, G_1, G_2 be BU graphs for F_0, F_1, F_2 before these call sites are resolved (so $C_1 \in$ call nodes of G_0; $C_2 \in$ call nodes of G_1). Suppose G' is the graph computed by the sequence $\mathit{ResolveCallee}(G_1, G_2, F_2, C_2)$; $\mathit{ResolveCallee}(G_0, G_1, F_1, C_1)$, and G'' is computed by $\mathit{ResolveCallee}(G_0, G_1, F_1, C_1)$; $\mathit{ResolveCallee}(G_0, G_2, F_2, C_2)$. Then the graphs G' and G'' are isomorphic.

This lemma shows that a call site (here, C_2) can be deferred and resolved in a later (ancestor) function. Note, that it is irrelevant whether the nodes being considered are part of the same or different graphs: informally, the three graphs can be merged (e.g., via $G = \mathit{cloneGraphInto}(G_2, \mathit{cloneGraphInto}(G_1, G_0)))$ *before* any merging operations are performed. Second, it is not difficult to see for any node N in G_0 that is reachable from at most one of the two call sites (say, from C_1): such a node is unaffected when C_2 is resolved. A node N reachable from both call sites is more difficult, but two key insights are important here: (a) For any path from actual argument a_1, there is a unique corresponding path from the corresponding formal argument f_1 (so that the nodes on the common prefix of those paths can be paired $1-1$). (b) For every acyclic path from $C_1 \longrightarrow N$, there is a well-defined set of nodes that will be merged with node N in $\mathit{ResolveCallee}(G_0, G_1, F_1, C_1)$. The final set of nodes merged with N is exactly the union of all such sets of nodes, due to all acyclic paths. The same now holds for every path from $C_2 \longrightarrow N$. Furthermore, the final set of nodes merged with N is the union of these two sets, regardless of the order in which the two *ResolveCallee* operations are performed. The result then follows from Lemma 1.

An important corollary of the above properties is that, if a indirect call edge $F_X \rightarrow F_Y$ that is part of an SCC S is resolved in the context of a function F_0, "higher up" in the call graph, the graph computed for F_0 is conservative (i.e., "safe") approximation of the the graph that would have been computed if $F_X \rightarrow F_Y$ were resolved directly in F_X (e.g., if the edge were known *a priori*). Essentially, this follows because we can show that every graph inlining operation performed before resolving F_0 in the latter would also be performed before resoving F_0 in the former[7].

Another simple corollary of the above properties is that the BU and TD graphs for a function that is potentially part of an SCC will be correct even if it is not known that the function is part of an SCC, i.e., even if some of its callers and some of its callees are not reflected in the graph.

3.4 Top-Down Analysis Phase

The Top-Down analysis pass is used to propagate information from callers to callees. The goal of this phase is to mark nodes pointed to by incoming argument **C**omplete, which we can do if all possible contexts where the function is invoked are known. The graphs constructed by the Top-Down pass are particularly useful for applications like alias-analysis.

The Top-Down construction phase is very similar to the Bottom-Up construction phase. It is actually somewhat simpler because the BU phase has already identified the call graph, so that the TD phase can traverse the SCCs of the call graph directly using Tarjan's algorithm; it does not need to "re-visit" SCCs as the BU phase does. Note that some SCCs may have been visited only partially in the BU phase, so the TD phase is responsible for merging their graphs.

Overall, the TD phase differs from the BU phase in only 4 ways: First, the TD phase never marks an SCC as unvisited as explained above: it uses the call edges discovered and recorded by the BU phase. Second, the TD phase visits SCCs of the call graph computed by the Bottom-Up traversal in reverse postorder instead of postorder. Third, the Top-Down pass inlines each function's graph into each of its callees (rather than the reverse), and it inlines a caller's graph into all it's potential callees directly (it never needs

[7]We believe but have yet to prove that the graph for F_0 is actually equal in both cases.

to "defer" this inlining operation since the potential callees at each call site are known). The final difference is that formal argument nodes are marked complete if all callers of a function have been identified by the analysis, i.e., the function is not accessible to any external functions. Similarly, global variables may be marked complete, unless they are accessible to external functions.

An important aspect of Data Structure Analysis is that different graphs may be useful for different purposes. The TD graphs are useful for applications like alias analysis, which want the most "complete" information possible. The BU graphs are useful for accurately determining the effect of a particular call site — they provide parameterized pointer information [17] for each function. For example, to perform Interprocedural Mod/Ref analysis, we compute the Mod-/Ref side-effects of a function call by clearing the Mod/Ref bits in a copy of the caller's TD graph, then inlining the BU graph for the callee (or all potential callees) into this copy. This provides context-specific information for the side-effects of the callee(s) at a particular call site, distinct from their behavior at other call sites. Even the local graphs are useful: we have built a field-sensitive implementation of the Steensgaard's algorithm starting with the local graphs.

3.5 The Globals Graph

One reason the DS graph representation is so compact is that each function graph need only contain the data structures reachable from that function. However, Figures 7(c) and 10 illustrate a fundamental violation of this strength. In both of these graphs, the global variable `G` makes an appearance even though it is not directly referenced and no edges target it. Such nodes cannot simply be deleted because they may have to be merged with other nodes in callers or callees of each function.

If left untreated, all global variables defined in the program would propagate bottom-up to `main`, then top-down to all functions in the program. This would balloon the size of each graph to include every global variable in the program, allowing a potential $O(N^2)$ size explosion.

In order to prevent this unacceptable behavior, our implementation uses a separate "Globals Graph" to hold information about global nodes and all nodes reachable from global nodes. This allows us to remove global variables from a function's graph if they are not used in the current function (even though they may be used in callers or callees of that function). For example, this eliminates the two `G` nodes in the example graphs.

For the steps below, all nodes reachable from virtual registers (which includes formal parameters and return values of the current function, and call node arguments within the current function, but not globals) are considered to be *locally used*. Call nodes are also considered to be locally used.

More specifically, we make the following changes to the algorithm:

- In the BU phase (respectively, TD phase), after all known callees (respectively, callers) have been incorporated in step 4, we copy and merge in the nodes from the globals graph for every global G that has a node in the current graph, plus any nodes reachable from such nodes. This ensures that the current graph reflects all known information about such globals from other functions.
- After step 5 in the BU phase, we copy all global nodes and nodes reachable from such nodes into the globals graph, merging the global nodes with the corresponding nodes already in the Globals Graph, if any (which will cause other "corresponding" nodes to be merged as well). We clear the **S**tack markers on nodes being copied into the Globals Graph, for the same reason as in *ResolveCallee*. We also clear the **C**omplete markers since those markers will be re-computed correctly within the context of each function.

  By the end of the BU phase, all the known behavior about globals will be reflected in the Globals Graph. Therefore, globals do not need to be copied from the TD graph to the Globals graph in the TD phase.

- In step 6 of the BU phase, we identify global nodes that are not reachable from any locally used nodes *and do not reach any such nodes*. The latter requirement is necessary because we may revisit the current function later, resolving previously unresolved call sites, which can bring in additional globals. Merging such globals will not correctly merge other reachable nodes in the graph if a global that can reach a locally reachable node is removed from the graph. The latter requirement is not needed for the TD phase since no further inlining needs to happen after eaching step 6. We simply drop all these identified nodes from the BU or TD graph for the function.

In practice, we have found the Globals graph to make a remarkable difference in running time for global-intensive programs, speeding up the top-down phase by an order of magnitude or more.

3.6 Bounding Graph Size

In the common case, the merging behavior of the unification algorithm we use keeps individual data structure graphs very compact, which occurs whenever a data structure is processed by a loop or recursion. Nevertheless, the combination of field sensitivity and cloning makes it theoretically possible for a program to build data structure graphs that are exponential in the size of the input program. Such cases can only occur if the program builds and processes a large complex data structure using only non-loop, non-recursive code, and are thus *extremely* unlikely to occur in practice.

Using a technique like k-limiting [10] to guard against such unlikely cases is unattractive because it could reduce precision for reasonable data structures with paths more than k nodes long. Instead, we propose that implementations simply impose a hard limit on graph size (10,000 nodes, for example, which is much larger than any real program is likely to need). If this limit is exceeded, node merging can be used to reduce the size of the graph. Our results in Section 4 show that the maximum function graph size we have observed in practice across a wide range of programs is only 167 nodes, which is quite small (we exclude virtual registers since those are not propagated between functions).

3.7 Complexity Analysis

The local phase adds at most one new node, ScalarMap entry, and/or edge for each instruction in a procedure (before node merging). Furthermore, node merging or collapsing only reduces the number of nodes and edges in the graphs. We have implemented node merging using a Union-Find data structure, which ensures that this phase requires $O(n\alpha(n))$ time and $O(n)$ space for a program containing n instructions in all [22].

The BU and TD phases operate on the DS graphs directly, so their performance depends on the size of the graphs being cloned and the time to clone and merge each graph. We denote these by K and l respectively, where l is $O(K\alpha(K))$ in the worst case. These phases also depend on the average number of callee functions per caller (not call site), which we denote c.

For the BU phase, each function must inline the graphs for c callee functions, on average. Because each inlining operation requires l time, this requires fcl time if there are f functions in the program. The call sites within an SCC do not introduce additional complexity, since every potential callee is again inlined only once into its caller within or outside the SCC (in fact, these are slightly faster because only a single graph is built, causing common nodes to be merged). Thus, the time to compute the BU graph is $\Theta(fcl)$. The space required to represent the Bottom-Up graphs is $\Theta(fK)$. The TD phase is identical in complexity to the BU phase.

In practice, we have found that efficient handling of globals using the Globals Graph to prevent replicating globals across many functions is by far the most important performance issue in practice. When those are handled efficiently, the next most important issue appears to be the number of inlining operations at call sites, particularly in programs with large tables of function pointers (e.g., in 254.gap in the next section). This, however, only contributes a small constant factor and not any significant asymptotic growth in time, and this was borne out in our experiments.

4. EXPERIMENTAL RESULTS

We have implemented the complete Data Structure Analysis algorithm in the LLVM Compiler Infrastructure, using a C front-end based on GCC [12]. The analysis is performed entirely at link-time, using stubs for standard C library functions to reflect their behavior (as in other work [2]).

We evaluated Data Structure Analysis on four sets of benchmark programs: the Olden benchmark suite, the "ptrdist" 1.1 benchmark suite, the SPEC 2000 integer and floating point (C) benchmarks[8] and a set of other, unbundled, programs. The Olden benchmarks are widely-used pointer and recursion-intensive codes, while the "ptrdist" and SPEC codes, and some of the other codes have been frequently used to evaluate pointer analysis algorithms. Note that the `povray31` test includes the sources for the `zlib` and `libpng` libraries.

Table 1 describes relevant properties of the benchmarks. "LOC" is the raw number of lines of C code for each benchmark, "MInsts" is the number of memory instructions[9] for each program in the LLVM representation, and "SCC" is the size of the largest SCC in the call-graph for the program.

We evaluated the time and space usage of our analysis on a Linux workstation with a 3.06GHz Xeon processor. For these experiments, we compiled the LLVM infrastructure with GCC 3.2 at the `-O3` level of optimization. Table 1 shows information about the running times and memory usage of DS Analysis. The columns labelled "Local", "BU", and "TD" show the breakdown of analysis time for the three phases of the analysis.

The three largest programs in our study, `255.vortex`, `254.gap` and `povray31` are both fairly large and contain non-trivial SCCs in the call graph. Nevertheless, it takes only between 2.3 and 7.9 seconds to perform all three steps of the algorithm on these programs. To put these numbers in perspective, we compared them to the total time to compile and link the benchmarks with GCC 3.2 at the `-O3` level of optimization, on the same system. Data Structure Analysis required **10%**, **23%**, and **31%** of the GCC compile times for `255.vortex`, `gap` and `povray31` respectively. Note that GCC 3.2 includes *no aggressive interprocedural optimization*, indicating that this is a very reasonable cost for an aggressive interprocedural analysis which has many potential applications.

The table shows that memory consumption of DS Analysis is also quite small. The "Mem" column shows the amount of memory used by results of the BU and TD the analysis algorithm. The total memory consumed for the largest code (for both BU and TD) is less than **24MB**, which seem very reasonable for a modern optimizing compiler. These numbers are noteworthy considering that the algorithm is performing a context-sensitive whole-program analysis *with cloning*, and memory consumption (not running time) can often be the bottleneck in scaling such analyses to large programs[10].

The "# of Nodes" columns show statistics collected during the construction process. The "Total" column shows the *aggregate* number of nodes contained in the TD graphs for all functions in the program, "Max" is the maximum size of any particular function's graph, and "Collapsed" indicates the number of nodes in the TD graphs which had to be collapsed due to apparent type violations. In large programs, we have found that consistently about 10% of the nodes are collapsed in large programs. The most common reason for collapsing of nodes appears to be due to merging of Global nodes (e.g., all the format strings passed to `printf` in the program are usually merged!), and in some cases this in turn causes other nodes to be merged. Perhaps most importantly, the table shows that the aggregate number of nodes in the graphs as well as the maximum per graph both grow quite slowly with total program size.

We have also examined the scaling behavior of the analysis (omitted here due to lack of space). Our experience shows that running time appears to scale roughly as $\Theta(nlogn)$ with number of memory instructions in the program, across a range of over three orders-of-magnitude of code size.

5. RELATED WORK

There is a vast literature on pointer analyses (e.g., see the survey by Hind [10]), but the majority of that work focuses on context-insensitive alias information and does not attempt to extract properties that are fundamental to our goals (e.g., identifying disjoint data structure instances). Due to limited space, we focus on techniques whose goals are similar to ours.

[8]We omit gcc and perlbmk, which do not currently work because of incidental bugs in our C front-end.

[9]Memory instructions are `load`, `store`, `malloc`, `alloca`, `call`, and addressing instructions.

[10]Even in the closest comparable analysis [17], for example, field-sensitivity had to be disabled for the `povray3` program for the analysis to fit into 640M of physical memory. Judging by LOC, it also appears that the `zlib` and `libpng` libraries were not linked into the program for analysis.

	Code Size			Analysis Time (sec)				Mem (KB)		# of Nodes		
Benchmark	**LOC**	**MInsts**	**max |SCC|**	**Local**	**BU**	**TD**	**Total**	**BU**	**TD**	**Total**	**Max**	**Coll-apsed**
Olden-treeadd	220	38	1	0.0003	0.0004	0.0003	0.001	29	25	22	8	0
Olden-bisort	316	126	1	0.0005	0.0007	0.0006	0.0018	48	42	34	9	0
Olden-mst	389	182	1	0.0008	0.0014	0.0011	0.0033	62	54	108	14	0
Olden-perimeter	430	151	1	0.0006	0.0008	0.0006	0.002	51	44	24	7	0
Olden-health	408	293	1	0.0008	0.0012	0.0016	0.0036	71	56	57	14	2
Olden-tsp	520	262	1	0.0007	0.0009	0.0008	0.0024	57	50	37	10	0
Olden-power	524	354	1	0.0008	0.0010	0.0008	0.0026	65	55	57	13	0
Olden-em3d	587	324	1	0.0012	0.0022	0.0024	0.0058	93	100	273	29	1
Olden-voronoi	982	730	1	0.0020	0.0037	0.0031	0.0088	166	150	249	44	0
Olden-bh	1391	824	1	0.0019	0.0026	0.0022	0.0067	141	119	113	11	25
ptrdist-anagram	647	271	1	0.0012	0.0028	0.0019	0.0059	111	89	163	18	11
ptrdist-ks	684	546	1	0.0017	0.0025	0.0022	0.0064	97	68	207	24	0
ptrdist-ft	1301	433	1	0.0013	0.0024	0.0021	0.0058	112	86	150	14	0
ptrdist-yacr2	3212	1621	1	0.0051	0.0078	0.0063	0.0192	222	212	369	17	0
ptrdist-bc	6627	3729	1	0.0099	0.0241	0.0174	0.0514	591	408	738	29	16
179.art	1283	919	1	0.0025	0.0032	0.0029	0.0086	117	94	235	39	0
183.equake	1513	1582	1	0.0031	0.0041	0.0043	0.0115	167	108	314	60	7
181.mcf	2412	1104	1	0.0022	0.0033	0.0034	0.0089	157	115	216	18	3
256.bzip2	4647	1723	1	0.0048	0.0077	0.0066	0.0191	269	212	425	29	16
164.gzip	8616	2634	1	0.0065	0.0101	0.0081	0.0247	333	266	610	29	9
197.parser	11391	7799	3	0.0254	0.0668	0.0683	0.1605	1039	1063	1506	60	275
188.ammp	13483	9645	2	0.0232	0.0369	0.0289	0.089	653	563	1054	64	140
175.vpr	17729	8586	1	0.0258	0.0483	0.0487	0.1228	951	894	2461	56	238
300.twolf	20459	23340	1	0.0388	0.0483	0.0417	0.1288	750	617	1262	45	166
186.crafty	20650	15866	2	0.0390	0.0981	0.1137	0.2508	916	961	1996	107	7
255.vortex	67223	39326	38	0.0950	0.9569	1.2428	2.2947	4661	6143	8597	85	606
254.gap	71363	40808	20	0.1252	2.4300	1.3767	3.9319	12.5MB	12.2MB	7038	59	803
sgefa	1176	429	1	0.0016	0.0030	0.0027	0.0073	126	109	116	23	11
sim	1569	1247	1	0.0028	0.0035	0.0028	0.0091	118	80	215	28	0
burg	6438	5224	2	0.0165	0.0405	0.0406	0.0976	886	717	1614	29	162
gnuchess	10595	8058	1	0.0276	0.0689	0.0436	0.1401	889	814	2132	56	79
larn	15179	13134	1	0.0336	0.0939	0.0857	0.2132	1138	1053	2740	49	293
flex	20534	7113	3	0.0204	0.0409	0.0412	0.1025	638	608	1597	57	85
moria	36010	38299	4	0.0726	0.2928	0.5492	0.9146	2604	4713	2433	76	386
povray31	136951	57562	102	0.1471	1.2116	6.5859	7.9446	7901	16.0MB	26687	167	2573

Table 1: Program information, analysis time, memory consumption, and graph statistics

The most powerful class of related algorithms are those referred to as "shape analysis" [11, 9, 19]. These algorithms are strictly more powerful than ours, allowing additional queries such as "is a given data structure instance a singly-linked list?" However, this extra power comes at very significant cost in speed and scalability, particularly due to the need for flow-sensitivity and iteration [19]. Significant research is necessary before such algorithms are scalable enough to be used for moderate or large programs.

The prior work most closely related to our goals is the recent algorithm by Liang and Harrold [17], named MoPPA. The structure of MoPPA is similar to our algorithm, including Local, Bottom-Up, and Top-Down phases, and using a separate Globals Graph. The analysis power and precision of MoPPA both seem very similar to Data Structure Analysis. Nevertheless, their algorithm has several limitations for practical programs. MoPPA can only retain field-sensitivity for completely type-safe programs, and otherwise must turn it off entirely. It requires a precomputed call-graph in order to analyze indirect calls through function pointers. It also requires a complete program, which can be a significant limitation in practice. Finally, MoPPA's handling of global variables is much more complex than Data Structure Analysis, which handles them as just another memory class. Both algorithms have similar compilation times, but MoPPA seems to require much higher memory than our algorithm for larger programs: MoPPA runs out of memory analyzing `povray3` with field-sensitivity on a machine with 640M of memory.

Ruf's synchronization removal algorithm for Java [18] also shares several important properties with ours and with MoPPA, including combining context-sensitivity with unification, a non-iterative analysis with local, bottom-up and top-down phases, and node flags to mark global nodes. Unlike our algorithm, his work requires a call graph to be specified, it is limited to type-safe programs, and does not appear to handle incomplete programs. Nevertheless, we believe that the application in their work (synchronization removal via escape analysis) could be a powerful application for our algorithm as well.

Fähndrich et al. [7] describe an algorithm that is context-sensitive in a fairly limited form, flow-insensitive, and discovers the call graph incrementally during the analysis. It appears comparable to ours in terms of analysis time. However, it is implemented by naming heap objects based only on allocation site, i.e., would not identify disjoint data structure instances in many common programs.

Both the FICS algorithm of Liang and Harrold [16] and the Connection Analysis of Ghiya and Hendren [8] attempt to disambiguate pointers referring to disjoint data structures. But both ignore heap locations not relevant for alias analysis, and both algorithms have higher complexity.

Cheng and Hwu [2] describe a flow-insensitive, context-sensitive algorithm for alias analysis, which has two limitations relative to our goals: (a) they represent only relevant alias pairs, not an explicit heap model; and (b) they use a k-limiting technique that would lose connectivity information for nodes beyond k links. They allow a pointer to have multiple targets (as in Andersen's algorithm), which is more precise but introduces several iterative phases and incurs significantly higher time complexity than our algorithm.

Deutsch [4] presents a powerful heap analysis algorithm that is both flow- and context-sensitive and uses access paths represented by regular expressionsto represent recursive structures efficiently. Although based on access paths, it appears possible to reconstruct heap information from the regular expressions created. In practice however, his algorithm appears to have much a higher complexity than ours.

In our earlier work on the automatic pool allocation [13] we presented a preliminary algorithm similar to, but much weaker than, Data Structure Analysis. That algorithm only used a bottom-up traversal, was exponential in the worst case, and much more expensive in common cases. The lack of a top-down pass made it produce many more incomplete results. We also did not evaluate the algorithm since it was not the primary goal of that paper.

As discussed in the Introduction, even many context-sensitive algorithms do not clone heap objects in different calling contexts. Instead, it is common to use a more limited naming schemes for heap objects (often based on static allocation site[11]) [6, 24, 7, 23, 3]. This precludes obtaining information about disjoint data structure instances, which is fundamental to all applications of *macroscopic* data structure transformations. In the case of Figure 1, for example, all nodes of both lists are created at the same `malloc` site, which would force these algorithms to merge the memory nodes for the `X` and `Y` lists, preventing them from proving that the lists are disjoint.

6. CONCLUSION

This paper presented a heap analysis algorithm that is designed to enable analyses and transformations on disjoint instances of recursive data structures. The algorithm uses a combination of techniques that balance heap analysis precision (context sensitivity, cloning, field sensitivity, and an explicit heap model) with efficiency (flow-insensitivity, unification, and a completely non-iterative analysis). We showed that the algorithm is extremely fast in practice, uses very little memory, and scales almost linearly in analysis time for 35 benchmarks spanning 3 orders-of-magnitude of code size. We believe this algorithm enables novel approaches for the analysis and transformation of pointer-intensive codes, by operating on entire recursive data structures (achieving some of the goals of shape analysis, with a weaker but more efficient approach). We are exploring several such applications in our research, including further applications of automatic pool allocation [13], transparent pointer compression, pointer prefetching, and automatic parallelization.

7. REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] B.-C. Cheng and W. mei Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evalutation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, Vancouver, British Columbia, Canada, June 2000.

[3] M. Das, B. Liblit, M. Fahndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS '01: The 8th International Static Analysis Symposium,*, July 2001.

[4] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

[5] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, San Diego, CA, Jun 2003.

[6] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, FL, June 1994.

[7] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proc. 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI00)*, Vancouver, Canada, June 2000.

[8] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, 1996.

[9] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Symposium on Principles of Programming Languages*, pages 1–15, 1996.

[10] M. Hind. Pointer analysis: haven't we solved this problem yet? In *ACM SIGPLAN — SIGSOFT workshop on on Program analysis for software tools and engineering*, pages 54–61. ACM Press, 2001.

[11] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 21–34, July 1988.

[12] C. Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. See `http://llvm.cs.uiuc.edu`.

[13] C. Lattner and V. Adve. Automatic Pool Allocation for Disjoint Data Structures. In *Proc. ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, Jun 2002.

[14] C. Lattner and V. Adve. The LLVM Instruction Set and Compilation Strategy. Tech. Report UIUCDCS-R-2002-2292, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Aug 2002.

[15] C. Lattner and V. Adve. Data structure analysis: An efficient context-sensitive heap analysis. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Jul 2003. (available via `llvm.cs.uiuc.edu`).

[16] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *ESEC / SIGSOFT FSE*, pages 199–215, 1999.

[17] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analysis. In *Static Analysis Symposium*, July 2001.

[18] E. Ruf. Effective synchronization removal for java. In *Proc. PLDI'00 Conf. Prog. Lang. Design and Implementation*, June 2000.

[19] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1), Jan. 1998.

[20] R. Sedgewick. *Algorithms*. Addison-Wesley, Inc., Reading, MA, 1988.

[21] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Computational*

[11] In principle, such algorithms can be implemented to use cloning, but the cost would become exponential [24, 7]. Making cloning efficient is the key challenge.

Complexity, pages 136–150, 1996.

[22] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, Jan 1996.

[23] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 35–46. ACM Press, 2001.

[24] R. P. Wilson and M. S. Lam. Effective context sensitive pointer analysis for C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.