Data Structure Analysis: An Efficient Context-Sensitive Heap Analysis

Chris Lattner, Vikram Adve

University of Illinois at Urbana-Champaign

Abstract. This paper presents an efficient context-sensitive heap analysis algorithm called Data Structure Analysis designed to enable analyses and transformations on entire disjoint recursive data structures. The analysis has several challenging properties needed to enable such transformations: *context-sensitivity with cloning* (essential for proving disjointness), *field-sensitivity*, and the use of an *explicit heap model* rather than just alias information. It is also applicable to arbitrary C programs. To our knowledge no prior work provides all these properties *and* is efficient and scalable enough for large programs. Measurements for 29 programs show that the algorithm is extremely fast, space-efficient, and scales almost linearly across 3 orders-of-magnitude of code size.

1 Introduction

There has been extensive research on alias analysis for programs containing complex pointer-based data structures. This work has been successful guiding traditional scalar and memory hierarchy optimizations, which operate at the level of individual memory references or data objects. Such 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 logical data structures* such as an entire list, heap, or graph. Many reasons exist for this disparity, including the possibility of non-type-safe memory accesses in common programming languages (e.g., C and C++), the inability of traditional analyses to distinguish between different instances of a data structure, and high compilation time required to perform the necessary analyses and transformations. In the long-term, we believe that such techniques, which we term *macroscopic data-structure analyses and transformations*, could provide new opportunities for program optimization, safety checking, and debugging tools. An example of such a technique is *automatic pool allocation*, which transforms an ordinary C program with dynamic allocation so as to *segregate* disjoint instances of logical data structures into separate memory pools within the heap [15]. This transformation has a number of interesting applications, including static checking of heap safety for a large class of type-safe C programs [6].

Such transformations require a powerful memory analysis capable of identifying disjoint logical data structures, building a static representation of the run-time heap, and detecting type-unsafe data structures. Traditional alias and pointer analysis algorithms do not attempt to provide such information because

of the potential cost, as explained below. In contrast, "shape analysis" algorithms are powerful enough to provide the information we require and more, e.g., enough to identify a particular structure as a "linked-list" or "binary tree" [10, 19]. Shape analysis, however, is too expensive to be practical for use in commercial compilers.

Our goal in this work is to develop an analysis algorithm that is somewhat more powerful than traditional pointer analysis but not as powerful as shape analysis, and which has the properties required to provide the information above. In particular, the analysis we require must include the following key properties, each of which is challenging to achieve efficiently, especially for languages like C:

- **Context-sensitivity with cloning**: Identifying disjoint data structures requires distinguishing (or "cloning") heap objects created via different call paths in a program, even if the objects are allocated at a common allocation site. Naïve cloning can lead to an explosion in the size of the heap representation (because there may be an exponential number of call paths), and can make recursion tricky to handle. In practice, therefore, most pointer or alias-analysis algorithms use more restricted naming schemes for heap objects such as distinguishing objects based on static allocation site alone. Even many *context-sensitive* algorithms do not attempt to distinguish heap objects by call paths [7, 24, 8, 23]. Although the naming scheme in many algorithms can be replaced to use cloning, making this efficient and scalable is a fundamental challenge.
- **Field-sensitivity**: Identifying the internal connectivity pattern of a logical data structure requires distinguishing the points-to properties of different structure fields. Such "field-sensitivity" is also difficult to support efficiently in languages lacking strict type enforcement (like C) (e.g., see [21]).
- **An explicit heap model:** An explicit heap model includes information about all the relevant memory objects visible in each procedure, and is required to extract the internal connectivity of relevant data structure instances. In contrast, many algorithms only record alias pairs that determine the aliasing behavior of pointers within each procedure [7, 3, 12] because building an explicit heap model can be more expensive.

In this paper, we develop a practical, scalable algorithm called "Data Structure Analysis" which provides the properties above and applies to arbitrary C programs. The key technical contributions of the Data Structure Analysis algorithm are as follows:

(i) Data Structure Analysis provides a scalable, context-sensitive, field-sensitive heap analysis with full cloning. The algorithm handles the full generality of C programs, including type-unsafe code, incomplete programs, function pointers and recursion.

(ii) The algorithm includes several novel features. First, it explicitly and efficiently tracks "incomplete" memory nodes, which ensures that it is sound even at intermediate stages, and that it can analyze incomplete programs safely. Second, the algorithm does not require the call graph to be provided

as input. It discovers the call graph during the analysis (similar to [8]), and uses a novel technique to handle strongly connected components (SCCs) of the call graph explicitly (in order to avoid iteration), even when recursion happens via function pointers. Finally, the algorithm is not iterative either within or across procedures in the sense that it visits every instruction only once in the local phase and incorporates the effect of a particular callee procedure at a particular call site only once during the interprocedural phases.

(iii) We show that the worst case complexity (with a graph-size-limiting heuristic that has never been invoked in practice) is $\Theta(n\alpha(n) + ks^2)$ where n, k, and s denote the number of instructions, the maximum size of a data structure graph for a single procedure, and the maximum number of functions in an SCC of the call graph, respectively.

(iv) We experimentally evaluate the algorithm on **29** C programs, showing that the algorithm is extremely efficient in practice (in both performance and memory consumption), including programs that contain complex heap structures, recursion, and function pointers. For example, it requires less than **2.5** seconds of analysis time and less than **9MB** to analyze `povray31`, a program consisting of over 130,000 lines of code. We evaluate the impact of some of the key design choices on the analysis precision by comparing several configurations of the algorithm as well as Steensgaard's [22] and Andersen's [1] algorithms.

The two closest previous algorithms to ours are those by Fähndrich et al. [8] and by Liang and Harrold [17]. Both algorithms are context-sensitive, flow-insensitive, and appear comparable to ours in terms of analysis time. The former, however, is implemented by naming heap objects based only on allocation site, i.e., would not identify disjoint data structure instances in many common programs, as discussed above. It also uses a more limited form of context-sensitivity. The algorithm by Liang and Harrold [17] is similar to ours in several key ways, particularly the use of local, bottom-up, and top-down passes, and in the use of a separate globals graph to avoid duplicating globals in procedures that do not use them. Their algorithm, however, has several practical limitations compared with ours: it requires a complete C program, it does not correctly handle non-type-safe programs (except by turning off field-sensitivity entirely), and it requires the call graph to be computed previously for programs with function pointers[1]. This algorithm is contrasted with ours in more detail in Section 5.

The remainder of the paper is organized as follows: Section 2 describes the semantics of the analysis graph representation. Section 3 then describes the algorithm used to construct the graph. Section 4 evaluates the analysis time, memory usage, and precision of our alias analysis. Section 5 compares our work with prior work in the field in more detail. Finally, Section 6 summarizes the results of the paper and outlines some directions for future work.

[1] An outline for a fairly complex iterative scheme is presented to compute the call graph on the fly, but it is not described in detail or evaluated.

```
typedef struct list { struct list *Next;
                      int Data; } list;
int G = 10;
void do_all(list *L1, void (*FP)(int*)) {
  do { L2 = phi(L1, L3);  /* SSA phi node */
       FP(&L2->Data);
       L3 = L2->Next;
  } while(L3);
}
void addG(int *X) { (*X) += G; }
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);
  addGToList(Y);
}
```

Fig. 1. C code, in SSA form, for running example

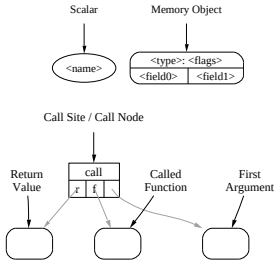


Fig. 2. Graph Notation

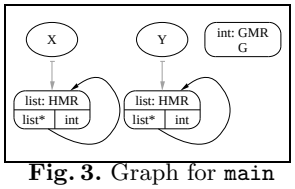


Fig. 3. Graph for `main`

2 The Data Structure Graph

Data Structure Analysis summarizes a program's memory composition and connectivity patterns by building a Data Structure Graph (DS graph) for each available function in the program. Many design features of DS graphs have been carefully chosen to make the analysis very efficient in practice. The major components are described in turn below.

The code in Figure 1 will be used as a running example throughout the paper. We use the notation illustrated in Figure 2 to represent all data structure graphs. Despite the complexity of the example, Data Structure Analysis is able to prove that the two lists `X` and `Y` are disjoint, as shown by the final DS graph for function `main` in Figure 3. Figure 4 shows the initial DS graph computed for the `do_all` function without interprocedural information. We refer to this graph in the description below.

2.1 Data Structure Nodes

A DS graph contains memory nodes and call nodes. Each memory node represents a set of memory objects, and edges represent *may-point-to* relationships. We restrict each pointer to point to a single target node (i.e., we use a "unification-based" approach similar to Steensgaard's algorithm [22]). If the analysis discovers two different nodes that may be pointed to by the same pointer, it merges the two nodes together. Unification allows an efficient non-iterative analysis of pointer assignments [22]. More importantly for us, unification is crucial for preventing exponential growth of the graph representation when doing a context-sensitive analysis with full cloning.

Each memory node in a DS graph includes 3 pieces of information: (a) an array of *fields*; (b) a program *type* for the objects represented by the node; and

(c) a set of *flags*. The field array and flags make node merging very efficient, while the type allows merging to detect incompatible merges, causing nodes to be collapsed as neccesary.

Formally, "fields" denote potential *outgoing edges*. They are stored as an array of (possibly null) edges, with one entry for each `sizeof(pointer)` bytes in the node [2]. A node may be "Collapsed", in which case the entries in the field array are folded into a single field, thus forcing the destination nodes for all outgoing edges to be merged. This loses field sensitivity for the node but retains correctness (this can happen under 3 situations, described below). In Figure 4, the node `list` has two fields, with an outgoing edge from the first field, and incoming edges both to the node (byte 0) and to the second field (byte 8).

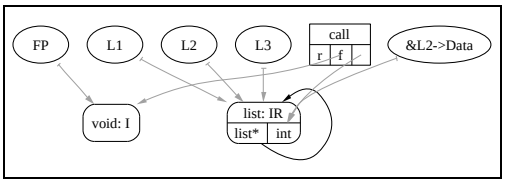


Fig. 4. Local DSGraph for `do_all`

If a potential pointer value (or bytes thereof) is stored to a node at any other offset (i.e., is misaligned), the node is collapsed. In practice we find that misaligned pointers are rare because they cause run-time faults on some processors. Tracking only aligned pointer fields greatly improves analysis efficiency and space requirements for most C programs.

Each node in the graph tracks a type for the objects it represents, e.g., `void` and `list` at the top of the memory nodes in the figure. When two nodes are merged, their type information is merged if the node types are compatible element-wise; otherwise the node is collapsed.

Memory Allocation Classes Data Structure Analysis distinguishes between four different classes of objects: **H**eap-allocated, **S**tack-allocated, **G**lobals, and **U**nknown objects, denoted **H**, **S**, **G** and **U** in our examples. Merging may cause a single node may represent memory objects of different allocation classes, i.e., to have multiple flags set. Functions are explicitly represented as Global memory objects. Memory objects are marked as Unknown 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.

We use a low-level code representation that distinguishes memory locations and virtual registers, where automatic (i.e., local) scalar variables that do not have their address taken can be promoted to virtual registers. Virtual registers are not represented as nodes in the graph since they cannot have incoming edges (although they are shown as ellipses in our figures, e.g., `FP, L1, L2, L3, &L2->Data` in Figure 4). Instead, a `ScalarMap` maps each register with a pointer-compatible type to the node the register points to. Conceptually, a map entry is just a very simple node that can have a single outgoing edge but no incoming edges.

[2] Incoming edges, however, may point to any byte offset within a node as described in Section 2.2.

Representing Incomplete Information DS graphs correctly represent incomplete programs where some functions are unavailable for analysis. To do this efficiently, each node in the data structure graph contains a bit to indicate if it is "**I**ncomplete". If this bit is set for a node, there may be some missing information, specifically, missing outgoing edges, type information, or flag information. If the bit is clear, the node is fully represented. Two different memory nodes in a DS graph may represent a common runtime memory location if they *both* have the "**I**ncomplete" flag set. In all other cases, two memory nodes represent disjoint memory locations.

In Figure 4, both memory nodes (labelled `void` and `list`) have the **I** flag set because the pointers from formal arguments `L1` and `FP` imply that those nodes may be modified outside the context of the current function, and even may be aliased. These **I** flags will be eliminated later using interprocedural information.

Because we track which nodes in the graph may contain incomplete information, the DS graph is sound regardless of how much information has been incorporated into it. This also dramatically simplifies the construction algorithms presented in Section 3. Note that client analyses must be aware of the potential for incomplete nodes: for example, an alias analysis can only conclude that two pointers are distinct if the pointers point to different nodes *and* at least one of the nodes is complete.

Mod/Ref Information The last two bits tracked by a DS graph node, **M**od and **R**ef, indicate whether or not any of the objects represented by node have been modified or read. The partitioning of memory objects in the DS graph provides a natural granularity to represent this information. In Figure 4, the "ref" bit is set on the list node because the `Next` field is read.

2.2 Data Structure Edges

In a DS graph, an edge goes from a field or a scalar to a <*node*,*offset*> pair. If the *node* entry is not NULL, the *offset* identifies a byte offset within the node to which the edge points, permiting field-sensitivity for C programs. In Figure 4, the edges from `FP`, `L1`, `L2`, and `L3` all have an offset of zero, but the edge from `&L2->Data` has a node offset of 8 bytes (which is the size of a pointer in our target system).

2.3 Call Site Information and Return Values

The DS graph for a function may contain "call nodes" in addition to traditional memory nodes. The presence of a call node indicates an unresolved function call, which may occur either due to an incomplete program or unfinished analysis. In the local DS graph in Figure 4, a call node exists representing the unresolved call to the function pointed to by `FP`.

Call nodes contain fields for the return value of the call (marked '`r`'), the called function (`'f'`), and each pointer compatible argument. Note that call

nodes model all calls as general indirect calls for uniformity. Finally, if the function returns a pointer type, the function's graph represents the returned object as a special scalar labeled "returning" with an edge to the object being returned.

3 Construction Algorithm

Data structure graphs are created in a three step process. First, an intraprocedural phase processes each function in the program, abstracting the behavior of each into a "Local" data structure graph, ignoring callers and turning call sites into call nodes. Next, a "Bottom-Up" analysis clones and merges callee graphs into their callers. The final "Top-Down" phase clones and merges caller graphs into their callees. The local analysis phase is the only phase that inspects the actual program representation. The other two phases operate solely on DS graphs because our analysis is flow-insensitive and because call sites are recorded in the graphs.

All three phases of the algorithm use a common set of routines to update and merge nodes, which are defined in Appendix A. Below, we continue to use the example program of Figure 1 as a motivating example. It illustrates some of the high-level challenges that our algorithm can handle.

3.1 Local Analysis Phase

The local analysis phase captures the memory usage behaviors of individual functions without including calling or caller context. Due to the potential for type-unsafe pointer arithmetic and casts, we consider any value capable of holding a pointer to be a pointer type (our implementation assumes C programs and therefore considers actual pointer types and any integers of pointer size or larger to be potentially a pointer type). We process all operations on any values of these types.

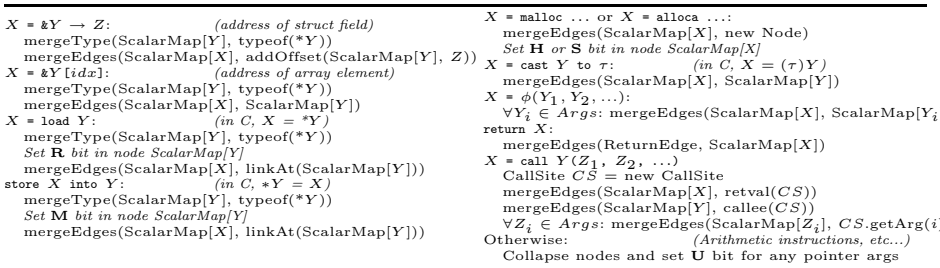


Fig. 5. ProcessInstruction actions for the LLVM Intermediate Representation

Since local analysis must examine the code, we describe this phase in terms of the LLVM instruction set [14], for which we implemented the analysis. This is a simple 3-address language with virtual registers in Static Single Assignment (SSA) form [4]. Memory locations are not in SSA form, and can only be accessed via load or store operations on typed pointers. All heap and stack memory

(including variables retained on the stack because their address is taken) are allocated using the primitive operations `malloc` and `alloca`, respectively.

The analysis starts by creating entries in the "ScalarMap" (§2.1) for any memory value directly referenced by the function, such as globals and constants. For example, in Figure 6(a), the two **G**lobal nodes are created. The next phase of the analysis is a flow-insensitive linear pass over the program representation (which is similar to Steensgaard's algorithm, but calculated one function at a time). In LLVM, each instruction is processed as shown in Figure 5. We describe a few cases in detail here.

`malloc` and `alloca` operations create a new memory node with the appropriate memory class bit set. `load` instructions update type information for the value loaded, updates mod/ref information, and then merge the source and destination pointers. The operations used to perform these tasks are defined in Appendix A. `cast` instructions (a copy is just a degenerate cast) simply merge the source and destination pointers, and do not update type information. This allows our algorithm to handle casts to and from `void*` pointers (i.e., generic C data structures) without collapsing nodes in most cases, simply by deferring the type determination to instructions (like load) where the type is actually used.

`return` instructions are handled by updating the return value for the current DS graph (§2.3). Each `call` instruction is represented as a new call site object in the graph. Note that the `mergeType` call is responsible for collapsing a node into a single field if the memory object is used in a non-type-safe manner. For all other instructions involving a pointer-compatible operand or result (these can only be arithmetic operations like `add` or `shift`), we set the "**U**nknown" bit and collapse the node to indicate that something untrackable occurred.

The final step in the Local graph construction is to calculate which data structure nodes are complete and which are incomplete. For a Local graph, any node reachable from a formal argument, global, passed as an argument to a call site, or returned by a call site is marked as incomplete.

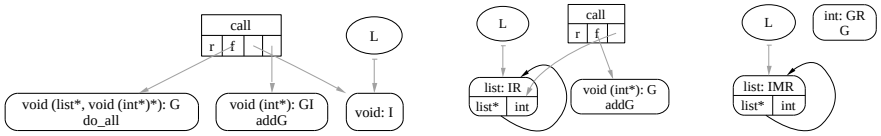
3.2 Bottom-Up Analysis Phase

The Bottom-Up (BU) analysis phase creates a graph for each function in the program, summarizing 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 $a_1,\ldots,a_n$. We first copy the BU graph for F, clearing all **S**tack node markers. Any unresolved call nodes in F's BU graph are retained. For each actual argument a_i of pointer type, we merge the node pointed to by a_i with the copy of the node pointed to by f_i. If applicable, we also merge the return value with the return marker from the call node.

The Bottom-Up algorithm for traversing calls is shown in Figure 7, 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 call-sites that are fully resolved, i.e., direct calls or indirect calls where the function pointer targets a "complete" nodes (§2.1). An unresolved call copied from a callee may become resolved if the function passed to a function pointer argument becomes known. This allows the indirect call to be resolved by copying and inlining the indirect callee's BU graph into the graph of the function where the call site became resolved. This technique of resolving call nodes as their function pointer targets are completed effectively discovers the call-graph on the fly.



(a) Local `addGToList` graph (b) After inlining `do_all` (c) Finished

Fig. 6. BU DSGraphs for `addGToList`

For example, Figure 6(a) shows the the local graph of `addGToList`, and Figure 6(b) shows the graph obtained by inlining the graph of `do_all`. This brings in a new call node that was not resolvable in `do_all` (see Figure 4), but is now known to call `addG`. This call node is now complete and we inline the graph for the `addG` function. This yields the finished graph for `addGToList` shown in Figure 6(c). Note that the BU graph for `do_all` still contains the unresolved call node, and that will not be resolved until the top-down phase.

Relaxing the final constraint, recursion, is trickier. We use an adaptation of Tarjan's linear-time algorithm for finding Strongly Connected Components (SCCs) to visit the SCCs of the call-graph in postorder. Given a simple Tarjan SCC iterator, the final Bottom-Up analysis algorithm is shown in Figure 7.

Assume first that there are only direct calls, i.e., the call graph is known. At the highest level, the BU phase iterates over all of the SCCs as they are identified by the Tarjan SCC iterator. For each SCC, all call sites to functions outside the SCC are cloned and resolved as before. 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.

At this point, simply traversing the SCC in some order (e.g., postorder) and inlining intra-SCC calls is not guaranteed to terminate, so we use a simple heuristic to make our algorithm halt. 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). Our heuristic simply achieves this directly by cloning the graph of each SCC function exactly once into each others

```
    BottomUpAnalysis(Program P)
      ∀ Function F ∈ P
        BUGraph{F} = LocalGraph{F}
      ∀ SCC S ∈ CallGraph(P)   - using Tarjan's algorithm
        ProcessSCC(S)

    ProcessSCC(SCC S)
      ∀ Function F ∈ S
(1)     CallSiteList CSL = getUnresolvedCallSites(BUGraph{F})
(2)     ∀ <Function,CallSite> <Callee,CS>∈ CSL
          if (Callee ∉ S)      - Process funcs not in SCC
            CloneAndResolve(BUGraph{F}, BUGraph{Callee}, CS)
(3)   ∀ Function F ∈ S
        CallSiteList CSL = getUnresolvedCallSites(BUGraph{F})
        CloneGraphs(S − {F}) - Copy SCC graphs except self
        ∀ <Function,CallSite> <Callee,CS>∈ CSL
          if (Callee ∈ S)      - Process funcs in SCC
            ResolveArguments(Callee, CS, BUGraph{F})
(4)   mark nodes incomplete - Section 3.1
(5)   remove unreachable nodes
(6)   ∀ Function F ∈ S
        if (ContainsResolvableCallSites(BUGraph{F}))
          MarkSCCUnvisited(S)  - Reevaluate SCC if funcpointer resolved
          TarjanVisitNode(SomeEntry(S))
```

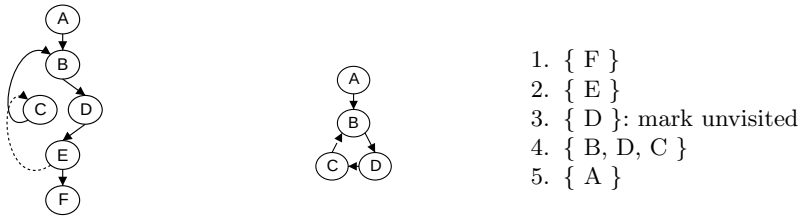
Fig. 7. Bottom-Up Closure Algorithm

graph, and then resolving arguments without any further inlining. In particular, inlining graphs into F_1 may result in one or more unresolved call nodes for some other SCC function F_2. Rather than inline the graph of F_2 again, we simply merge the corresponding nodes for each actual argument in the different calls, and then mark all call nodes to F_2 resolved. This may sacrifice some context-sensitivity for multiple intra-SCC calls to a single function (compared to a more complex SCC-traversal scheme) but the difference seems unlikely to be significant in practice, and this scheme has the advantage of being simple and efficient.

After the cloning and merging is complete for a function in the SCC, we identify incomplete nodes (§3.1). Finally, we remove unreachable nodes from the graph since copying and inlining callee graphs can bring in excess nodes not accessible within the current function (e.g., unused globals or heap nodes).

The final case to consider is a recursive program with indirect calls. Indirect calls that never induce any cycle in the call graph get resolved just as before, i.e., in each context where the function pointer gets fully resolved. Some indirect calls, however, may induce cycles in the SCC and these cycles will not be discovered until the indirect call is resolved. We make a key observation, however, that yields a simple strategy to handle such a situation: If A calls B and B calls C, the BU graph for A is the same regardless of which order the graphs are inlined: (i) C into B then B into A, or (ii) B into A then C into A. In both cases, the graph for A will reflect the side-effects of both calls. The difference is that in case (i), the graph for B will reflect the call to C but not in case (ii).

Based on this observation, we have slightly adapted Tarjan's algorithm to revisit partial SCCs as they are discovered. After the current SCC is fully processed (i.e., after step (5) in Figure 7), we check each SCC function to see if any newly inlined call nodes are now resolvable. If so, we reset the "Visit" flags



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

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

(c) SCCs visited

Fig. 8. Handling recursion due to an indirect call in the Bottom-Up phase

used in Tarjan's algorithm to mark if a node has been completely visited, for all functions in the SCC[20]. This causes the nodes in the 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 the list in step (1)).

For example, consider the recursive call graph shown in Figure 8(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 8(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 8(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 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 (exactly as illustrated in the observation above). 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 3 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 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.

3.3 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 construct a graph for each function which describes the memory access behavior of the function within all of the possible contexts in which the function is invoked. This allows us to reduce the number of nodes that are marked incomplete.

Because the Top-Down construction phase is very similar to the Bottom-Up construction phase, we omit the pseudo-code here. There are 3 differences between the BU and TD phases: First, the Top-Down pass visits SCCs of the call graph computed by the Bottom-Up traversal in reverse postorder instead of

postorder. Second, in the **ProcessSCC** function, the Top-Down pass inlines the caller graph into each of its callees (rather than its callers). The third difference is that argument nodes are not marked incomplete if all callers of a function have been identified by the analysis. Similarly, global variables need only be marked incomplete if they may be visible to external functions.

An important aspect of Data Structure Analysis is that different graphs may be useful for different purposes. For example, the BU graphs provide parameterized pointer information [17], useful for accurately determining the effect of a particular call site. The TD graphs are useful for applications like alias analysis, which want the most "complete" information possible. Even the local graphs are useful: we have built an field-sensitive implementation of the standard Steensgaard's algorithm using the local graphs as input.

3.4 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. If there were a caller of `main` in the example, that graph would not contain any of the nodes defined in `main`: unreachable node elimination would remove them. However, Figures 6(c) and 3 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.

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 moves all global variables and call sites to a "Globals Graph" if they are not used in the current function. All nodes reachable by scalars are considered to be locally used, and all call sites which have a locally used node as an argument are considered to be locally used. All other nodes and data structures may safely be moved and merged into the Globals graph, which eliminates the two `G` nodes the the example graphs. This happens for each function at the end of the Bottom-Up phase. Finally, when the Bottom-Up phase is complete, all global nodes in `main` (if the function `main` is available) are also copied to the globals graph.

In practice, we have found the Globals graph to make a remarkable difference in running time for global-intensive programs, and for programs with call sites which invoke external functions.

3.5 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 using only non-loop code, and are thus *extremely* unlikely to occur in practice.

Using a technique like k-limiting [12] 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 graph size we have observed in practice is only 144 nodes, which is quite small.

3.6 Complexity Analysis

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

For the BU and TD phases, the size of the program and number of memory instructions are inconsequential, because thee phases operate on the DS graphs directly. Instead, their performance depends on the size of the graphs being cloned and the time to clone and merge each graph (which we will denote 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, which we denote c.

For the BU phase, each function must inline the graphs for c callee functions. Because each inlining operation requires l time, this requires fcl time if there are f functions in the program. Handling SCCs requires no additional time for a single node SCC (the most common), whereas building the final graph for each function in an s node SCC requires $\Theta(ls^2)$ time (because each function needs to be inlined into each other function's graph). Thus, the time to compute the BU graph is $\Theta(fcl + ls^2)$ where s is the size of the largest SCC. The space required to represent the Bottom-Up graphs is $\Theta(fK)$. The TD phase is identical in complexity to the BU phase, with the role of callers and callees exchanged.

Asymptotically, the worst-case running time is dominated by the time to process SCCs in the call graph, which occurs when the entire program is one SCC. In this case, our worst-case analysis time is $\Theta(ls^2)$. Note that s is *much smaller* than the number of memory instructions in the program. In the worst case, it is equal to the number of functions in the program. Even in the worst case however, we find that processing functions in SCCs in postorder allows our algorithm to scale nearly linearly with the size of the program, and in practice, our algorithm is able to handle large SCCs without a problem.

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 [14]. The analysis is performed entirely at link-time, using stubs for standard C library functions to reflect their aliasing behavior as in other work [3, 11].

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 benchmarks[3] and a set of other, unbundled, programs. The Olden benchmarks are widely-used pointer and recursion-intensive codes [18, 2], 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.

	Code Size			Analysis Time (sec)				Mem (KB)		# of Nodes		
Benchmark	LOC	MInsts	SCC	Local	BU	TD	Total	BU	TD	Total	Max	Collapsed
Olden-treeadd	245	41	1	0.0004	0.0004	0.0004	0.0012	15	11	18	7	0
Olden-bisort	348	103	1	0.0006	0.0006	0.0009	0.0021	18	17	24	8	0
Olden-mst	432	144	1	0.0008	0.0025	0.0013	0.0046	32	23	87	14	4
Olden-perimeter	484	106	1	0.0006	0.0007	0.0006	0.0019	17	16	20	6	0
Olden-health	508	213	1	0.0010	0.0012	0.0015	0.0037	34	25	62	15	6
Olden-tsp	579	181	1	0.0008	0.0008	0.0009	0.0025	28	20	27	9	0
Olden-power	615	286	1	0.0008	0.0008	0.0011	0.0027	32	24	52	12	0
Olden-em3d	682	239	1	0.0013	0.0016	0.0019	0.0048	46	36	158	23	1
Olden-voronoi	1106	740	1	0.0026	0.0032	0.0047	0.0105	88	53	136	13	30
Olden-bh	2085	646	1	0.0024	0.0024	0.0033	0.0081	78	56	119	12	47
ptrdist-anagram	647	198	1	0.0014	0.0015	0.0023	0.0052	50	39	127	16	7
ptrdist-ks	782	338	1	0.0018	0.0021	0.0031	0.0070	64	35	190	29	0
ptrdist-ft	2157	330	1	0.0018	0.0024	0.0033	0.0075	71	44	173	17	0
ptrdist-yacr2	3979	1384	1	0.0064	0.0091	0.0116	0.0271	194	118	859	58	18
ptrdist-bc	7295	2599	1	0.0102	0.0164	0.0274	0.0540	267	171	632	36	80
181.mcf	2405	662	1	0.0034	0.0037	0.0054	0.0125	98	49	161	26	45
256.bzip2	4649	1303	1	0.0053	0.0050	0.0077	0.0180	138	89	311	28	10
164.gzip	8616	2028	1	0.0083	0.0079	0.0103	0.0265	179	109	413	23	45
197.parser	11391	5273	3	0.0251	0.0417	0.0918	0.1586	462	509	1404	71	318
300.twolf	20468	16319	1	0.0439	0.0448	0.0831	0.1718	459	449	2116	84	351
255.vortex	67221	28447	38	0.0907	0.2642	0.8326	1.1875	1756	2479	8215	95	774
sgefa	1218	311	1	0.0017	0.0033	0.0057	0.0107	94	79	88	26	0
sim	1569	1090	1	0.0029	0.0026	0.0040	0.0095	74	45	226	54	0
burg	6392	3756	2	0.0156	0.0272	0.0451	0.0879	512	319	1578	58	223
gnuchess	10595	6194	1	0.0277	0.0449	0.0774	0.1500	380	406	1946	144	213
larn	15179	2398	1	0.0111	0.0227	0.0291	0.0629	305	219	992	63	100
flex	20534	5608	3	0.0190	0.0258	0.0658	0.1106	323	319	1391	70	112
moria	36010	13372	30	0.0585	0.1174	0.4998	0.6757	1029	1672	3156	63	891
povray31	136951	42178	103	0.1454	0.7030	1.6435	2.4919	4139	4408	8648	96	2738

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

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 memory instructions[4] for each program in the LLVM representation, and "SCC" is the size of the largest SCC in the call-graph for the program.

4.1 Analysis Time & Memory Consumption

We evaluated the time and space usage of our analysis on a Linux workstation running a 1.7GHz AMD Athlon 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.

[3] Those which the LLVM C front-end is currently capable of compiling.

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

The columns labelled "Local", "BU", and "TD" show the breakdown of analysis time for the three phases of the analysis.

The two largest programs in out test suite, `255.vortex` and `povray31` are both respectably large and contain non-trivial SCCs in the call graph. It takes 1.19 and 2.5 seconds, respectively, to calculate Data Structure Analysis for these programs. To put these numbers in perspective, we compared them to the total time to compile the benchmarks with GCC 3.2 at the `-O3` level of optimization. Data Structure Analysis required **4.3%** and **6.7%** of time to compile `255.vortex` and `povray31` with GCC, 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 9MB, 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[5].

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 type-safety violations. Unfortunately, a substantial fraction of nodes are collapsed for some programs because the LLVM C front-end fails to extract type information for large functions (and *not* due to any weakness of our analysis). As we see in Section 4.2, this can substantially impact the precision of client analyses. We expect that improvements to its type inference algorithms (unrelated to this analysis) will help reduce collapsing significantly[6].

4.2 Analysis Accuracy Comparison

Data Structure Analysis is designed to support aggressive new *macroscopic data structure transformations*. For these applications, context-sensitivity with cloning is crucial, field-sensitivity makes the analysis much more precise (especially with unification), and unification is necessary to making cloning scalable. In order to evaluate the benefit/loss due to these specific choices, we compared

[5] 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 appears that the `zlib` and `libpng` libraries were not linked into the program for analysis.

[6] We do not expect fewer collapsed nodes to lead to longer analysis times because collapsing nodes causes many spurious globals to be pulled into the graphs, slowing down the current analysis significantly.

the precision of four different configurations of our analysis: context-sensitive (DS) and context-insensitive (St), with field-sensitivity (fs) and without (fi). Note that "St-fi" is effectively the classical Steensgaard's algorithm [22], and DS-fs is the algorithm evaluated in Table 1. We also include data for Andersen's [1] algorithm, for comparison with a non-unification based approach.

As a metric to compare the four analyses, we measure how well the analyses support alias analysis queries. Note, however, that we are explicitly not advocating alias analysis as the primary application of Data Structure Analysis: it simply provides a well-understood client that is useful to evaluate precision. To gather alias analysis precision numbers, we perform pairwise queries of all pointers visible in each function, leading to $n^2/2$ queries for each function, where n is the number of pointers in the function. In Table 2, we show the percentage of queries which return "May-Alias" for each analysis, i.e., which were not disambiguated (smaller is better).

Benchmark	Basic	St-fi	St-fs	DS-fi	DS-fs	And
Olden-treeadd	82%	68%	68%	68%	61%	68%
Olden-bisort	87%	82%	82%	82%	55%	82%
Olden-mst	88%	48%	28%	48%	34%	75%
Olden-perimeter	59%	57%	57%	57%	32%	57%
Olden-health	83%	61%	61%	61%	22%	59%
Olden-tsp	88%	87%	87%	87%	34%	86%
Olden-power	74%	26%	22%	26%	8%	23%
Olden-em3d	89%	54%	25%	54%	17%	17%
Olden-voronoi	56%	28%	28%	24%	23%	22%
Olden-bh	73%	20%	20%	20%	16%	19%
ptrdist-anagram	86%	26%	26%	26%	25%	18%
ptrdist-ks	92%	48%	39%	48%	32%	35%
ptrdist-ft	95%	77%	55%	77%	35%	64%
ptrdist-yacr2	98%	27%	27%	25%	19%	24%
ptrdist-bc	75%	57%	56%	43%	28%	47%
181.mcf	64%	57%	57%	44%	43%	47%
256.bzip2	76%	27%	27%	27%	26%	27%
164.gzip	77%	34%	34%	29%	29%	27%
197.parser	91%	78%	77%	71%	69%	73%
300.twolf	97%	94%	94%	34%	19%	93%
255.vortex	83%	72%	75%	63%	64%	77%
sgefa	94%	28%	28%	24%	19%	24%
sim	98%	12%	12%	12%	9%	12%
burg	88%	67%	51%	36%	30%	31%
gnuchess	68%	35%	35%	31%	30%	17%
larn	83%	27%	27%	22%	23%	27%
flex	98%	92%	92%	46%	46%	48%
moria	72%	64%	64%	54%	54%	50%
povray31	78%	66%	67%	50%	50%	67%

Table 2. Percent of "may alias" responses (smaller is better)

All of the alias analyses in LLVM use the "basic" analysis algorithm if they cannot answer a query, thus the "basic" column of Table 2 is the baseline for all analyses. The "basic" algorithm uses purely local techniques to answer queries, and it thus limited to handling simple cases (e.g., A[1] does not alias A[2]). We chose this strategy because it most closely models usage in a real compiler [11].

The results have several interesting aspects. First, we find that field sensitivity does not help the context-insensitive "St" analysis significantly (even in cases where excessive node collapsing does not happen): it only improves 7 of the 29 programs by 10% or more. Context-sensitivity helps about as much: 7 programs are improved by 10% or more between the "St-fi" and "DS-fi" columns, although the amount of improvement due to context-sensitivity is often much greater than field sensitivity (e.g., 94% to 34% for `300.twolf`). Perhaps the most interesting result is that *the combination of field-sensitivity and context-sensitivity seems to make a much greater difference than either of the two individually.* Comparing St-fi to DS-fs shows that 10 programs improve by more than 20%, some much more than that (e.g., `health`, `tsp`, `bc`, and `twolf`). Of the remaining programs, 12 have substantial node collapsing, implying that they may show larger improvements if node collapsing were reduced by a better C front-end.

A key choice in our algorithm is to accept node merging (i.e., unification) in order to enable context-sensitivity with cloning and field-sensitivity. Com-

parison's between the "St-fi" and "And" columns shows us the (well known and substantial) loss of precision due to using a unification based algorithm rather than an iterative one. More interestingly, comparing the "DS-fs" and "And" columns shows that combining field-sensitivity and context-sensitivity provides an analysis which is substantially more precise than Andersen's algorithm in some cases (e.g., `bisort, mst, perimeter, health, tsp, ft, bc, and 300.twolf`), and roughly comparable in the others. Furthermore, Andersen's analysis is incapable of supporting *macroscopic* data structure transformations because it is context insensitive, and we believe the real long-term value of Data Structure Analysis lies in such applications.

5 Related Work

There is a vast literature on pointer analyses (e.g., see the survey by Hind [12]), 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.

The most powerful class of related algorithms are those referred to as "shape analysis" [13, 10, 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, field-sensitivity, and detailed tracking of aliases [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. MoPPA requires a precomputed call-graph in order to analyze indirect calls through function pointers. MOPPA 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.

Both the FICS algorithm of Liang and Harrold [16] and the Connection Analysis of Ghiya and Hendren [9] 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 [3] 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 ($O(n^3)$ and $O(n^2)$ respectively).

Deutsch [5] presents a powerful heap analysis algorithm that is both flow- and context-sensitive and uses access paths represented by regular expressions, instead of k-limiting, to represent recursive structures efficiently. His algorithm appears to have higher complexity and seems much more expensive in practice.

In our earlier work on the automatic pool allocation [15] 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[7]) [7, 24, 8, 23]. 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 29 benchmarks spanning 3 orders-of-magnitude of code size. We believe this algorithm could enable novel approaches to the analysis and transformation of pointer-intensive codes, by operating on entire recursive data structures (in a sense, achieving some of the goals of shape analysis, via a weaker but more efficient approach). We are exploring several such applications in our research, including automatic pool allocation [15], static analysis of heap safety [6], transparent pointer compression, pointer prefetching, and automatic parallelization.

[7] In principle, such algorithms can be implemented to use cloning, but the cost would become exponential [24, 8]. Making cloning efficient is the key challenge.

References

1. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
2. B. Cahoon and K. McKinley. Data flow analysis for software prefetching linked data structures in java. In *International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, Sept. 2001.
3. 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.
4. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, pages 13(4):451–490, October 1991.
5. 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.
6. D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. Submitted for publication, Feb 2003.
7. 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.
8. 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.
9. 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.
10. 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.
11. R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 47–58. ACM Press, 2001.
12. 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.
13. 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.
14. 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`.
15. 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.

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*, 2001.
18. A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2), Mar. 1995.
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 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.

7 Appendix: Common Graph Manipulation Routines

All three phases of data structure analysis use a shared set of routines that centralize common operations (such as merging two nodes) along with code for creation, manipulation, and destruction of nodes and graphs. Data Structure Analysis uses four main data types: graphs (§2), nodes (§2.1), edges (§2.2), and call sites (§2.3). Operations on these data types include:

collapseNode(Node A) - This routine collapses the specified node down to a single byte of memory and a single field. This is used to maintain conservative correctness with non-type-safe programs. Note that *collapsing* and *merging* are two very different operations.

mergeType(Edge E, Type τ) - This is used to check to see if the node and offset specified by E can be treated as the specified type. If it would not be type safe to merge the two types together[8], the node is collapsed.

mergeNodes(Node A, Node B, uint Offset) - This routine implements unification-style merging of two nodes. First, all edges targeting B are changed to point to A. If not already collapsed, *mergeType* is used to collapse the nodes if not type-compatible. Finally, all outgoing edges of A and B are recursively merged together, the flags are bitwise or'd together, and B is destroyed.

mergeEdges(Edge A, Edge B) - This routine uses *mergeNodes* to ensure that two edges to point to the same node. If both edges target NULL nodes (i.e., they haven't been set yet), a dummy node is created for both of them to target (for example, the `void` node in Figure 4). Otherwise the two nodes targeted by the edges are merged, at an offset specified by the edge offsets in A and B.

[8] Type-safety details depend on the type-system used.