Data Structure Analysis User's Manual

Written by Patrick Simmons

January 1, 2017

1 Introduction

Data Structure Analysis is a form of field-sensitive, context-sensitive, unification-based alias analysis, described fully in Chris Lattner's thesis.[1] DSA is also the LLVM analysis pass implementing this algorithm. The goal of this user's manual is to document the external API of the LLVM DSA pass for the benefit of those who wish to write clients for this pass.

2 Brief Overview of DSA Algorithm

DSA roughly works in three distinct phases: Local, Bottom-Up, and Top-Down. In order to motivate the artifacts created by the DSA algorithm, these phases are briefly discussed below. Many simplifications have been made, and many details have been omitted. Those interested in the workings of DSA in any detail should consult Chris Lattner's thesis.

2.1 Local

The local phase of DSA creates a node ("DSNode") representing each pointer and pointer target within a function. It then creates a graph by creating edges between the nodes representing the "points-to" relationship. However, because DSA is unification-based, there can be only one edge from any single field. Thus, if the local analysis is not able to determine which of multiple objects is the target of a pointer, all possible targets must be merged into a single DSNode. The local analysis is run on each function in the program, creating separate graphs for each.

2.2 Bottom-Up

After the local phase has run, DSA iterates over the callgraph, callees before callers, and clones the callee's graph into the caller by matching the nodes corresponding to callee formal parameters to the actual parameters given in the caller. This cloning is key to the context-sensitivity of DSA. A number of tricks

[1] http://llvm.org/pubs/2005-05-04-LattnerPHDThesis.html

are used which in practice prevent this pass from exhibiting its exponential worst-case behavior.

2.3 Top-Down

The top-down phase of DSA iterates over the callgraph again, this time callers before callees, and merges nodes in callees when necessary. This has the unfortunate effect of destroying some of DSA's context-sensitivity but is necessary for clients that which to use DSA as simply an alias analysis (such DSA-AA).

3 Artifacts of DSA

The artifacts of DSA are a collection of directed graphs, called "DSGraphs", with nodes ("DSNodes") representing sets of objects which may alias and edges representing the points-to relation between these nodes.

3.1 The Function Graphs

A DSGraph is created for every function in the program. This graph contains nodes and edges representing the information from the local function, the callees if bottom-up DSA has been run, and the callees and callers if top-down DSA has been run.

When interpreting the results of a function graph, it is important to keep in mind the "perspective" of the DSA passes that have been run. For instance, under bottom-up DSA, a DSNode representing the target of a pointer argument to a function may not have the `global` flag but still may represent a global variable in some contexts if a particular caller passes the address of a global variable to the function. Roughly, in bottom-up DSA, a function graph contains the information on a function which is true no matter what calling the calling context is. In contrast, under top-down DSA, a function graph summarizes the information about a function taking into account all possible contexts. Under bottom-up DSA, a node which may be either a heap node or a stack node depending on context will have neither the heap nor stack flags set. Under top-down DSA, a node which may be either a heap node or a stack node depending on context will have both flags set. Under bottom-up DSA, two arguments which may or may not alias depending on context will be represented as two separate DSNodes; under top-down DSA, these arguments will be unified into one node.

Thus, bottom-up DSA gives more context-sensitivity to clients, but at the price of giving its results in a different and, for many clients, less useful form. It is up to the client to evaluate the trade-offs between the two passes.

3.2 The Globals Graph

The globals graph represents the interactions between global variables and nodes reachable from global variables. The globals graph contains exactly those

DSNodes which are reachable from some DSNode representing a global variable. The existence and design of this graph are partly motivated by internal DSA optimizations unlikely to be of concern to clients.

It is important not to confuse the globals graph with the `global` flag described later in this manual. While all DSNodes marked with the `global` flag will be represented in the globals graph since nodes with the `global` flag are exactly those which may represent global variables, the globals graph will also contain nodes which do not have the `global` flag, such as the targets of global pointers. The set of nodes in the globals graph is the transitive closure of the set of nodes with the `global` flag under the points-to relation.

4 Interfaces to DSA Results

This section lists selected, generally useful functions from the client API of DSA.

4.1 Special Types

DSA uses a number of special types in its arguments and return values. These are summarized in this section.

4.1.1 DSNodeHandle

DSNodeHandle is a smart pointer designed to contain one thing and one thing only: a pointer to a DSNode. The only interesting function it contains is `getNode()`, which returns the pointer associated with a DSNodeHandle. If a client pass wishes to build a data structure containing DSNode references, this class should be used instead of storing DSNode references directly. The reason for this is that calling getNode() on a DSNodeHandle has the potential to render invalid any raw DSNode pointer held by a client.

While DSNodeHandle implements the ¡, ¿, and == comparison functions by simply comparing the underlying raw pointers, it is not safe to use DSNodeHandles in STL or other data structures that expect to be used with immutable objects. Any accesses to DSNodes may result in the pointer associated with a DSNodeHandle changing its value and therefore change the result of these comparison operations. In particular, it is not safe to store DSNodeHandles in an STL set or map.

4.1.2 NodeMapTy

This is a typedef for `std::map<const DSNode*, DSNodeHandle>`.

4.1.3 InvNodeMapTy

This is a typedef for `std::multimap<DSNodeHandle, const DSNode*>`.

4.2 DataStructures : public ModulePass

- `DSGraph* getDSGraph(const Function& F) const`: return the DSGraph for the specified function, or null if none
- `DSGraph* getGlobalsGraph() const`: return the globals graph

4.3 DSGraph

- `DSGraph* getGlobalsGraph() const`: return the globals graph
- `node_iterator node_begin()/node_end()`: iterate over all the nodes in the DSGraph. Be extremely careful when using these methods.
- `string getFunctionNames() const`: return a space-separated list of the names of functions in this graph
- `const std::list<DSCallSite>& getFunctionCalls() const`: return a list of call sites in the original local graph
- `fc_iterator fc_begin()/fc_end() const`: iterate over call sites
- `DSNodeHandle& getNodeForValue(const Value* V) const`: return the DSNode in this graph associated with the specified value
- `bool hasNodeForValue(const Value* V) const`: return true if there is a DSNode in this graph associated with the specified value, false otherwise
- `DSNodeHandle& getReturnNodeFor(const Function& F)`: get the return node for the specified function (which must be associated with the current DSGraph)
- `static void computeNodeMapping(const DSNodeHandle& NH1, const DSNodeHandle& NH2, NodeMapTy& NodeMap, bool StrictChecking = true)`: given roots in two different DSGraphs, traverse the nodes reachable from the two graphs and construct the maping of nodes from the first to the second graph. This function is useful in allowing a client to analyze which nodes in two different DSGraphs represent the "same" value in a certain context; however, the most common potential uses of this function are handled by the more specialized functions below.
- `void computeGtoGGMapping(NodeMapTy& NodeMap)`: compute the mapping of nodes in this DSGraph to nodes in the globals graph
- `void computeGGtoGMapping(InvNodeMapTy& InvNodeMap)`: compute the mapping of nodes in the globals graph to nodes in this DSGraph
- `void computeCalleeCallerMapping(DSCallSite CS, const Function& Callee, DSGraph& CalleeGraph, NodeMapTy& NodeMap)`: given a call from a function in the current graph to the "Callee" function, which must be associated with "CalleeGraph", construct a mapping of nodes from the callee to nodes in this graph

4.4 DSNode

The DSNode class represents a node in a DSA function or globals graph and therefore represents a set of objects which may alias. Interesting analysis results about the behavior of this alias set are stored as flags:

```
enum NodeTy {
  ShadowNode      = 0,        // Nothing is known about this node...
  AllocaNode      = 1 << 0,   // This node was allocated with alloca
  HeapNode        = 1 << 1,   // This node was allocated with malloc
  GlobalNode      = 1 << 2,   // This node was allocated by a global var decl
  ExternFuncNode  = 1 << 3,   // This node contains external functions
  ExternGlobalNode = 1 << 4,  // This node contains external globals
  UnknownNode     = 1 << 5,   // This node points to unknown allocated memory
  IncompleteNode  = 1 << 6,   // This node may not be complete

  ModifiedNode    = 1 << 7,   // This node is modified in this context
  ReadNode        = 1 << 8,   // This node is read in this context

  ArrayNode       = 1 << 9,   // This node is treated like an array
  CollapsedNode   = 1 << 10,  // This node is collapsed
  ExternalNode    = 1 << 11,  // This node comes from an external source
  IntToPtrNode    = 1 << 12,   // This node comes from an int cast
  PtrToIntNode    = 1 << 13,  // This node escapes to an int cast
  VAStartNode     = 1 << 14,  // This node is from a vastart call
```

These flags may be accessed with the following accessors:

```
unsigned getNodeFlags() const;
bool isAllocaNode()     const { return NodeType & AllocaNode;    }
bool isHeapNode()       const { return NodeType & HeapNode;      }
bool isGlobalNode()     const { return NodeType & GlobalNode;    }
bool isExternFuncNode() const { return NodeType & ExternFuncNode; }
bool isUnknownNode()    const { return NodeType & UnknownNode;   }
bool isModifiedNode()   const { return NodeType & ModifiedNode;  }
bool isReadNode()       const { return NodeType & ReadNode;      }
bool isArrayNode()      const { return NodeType & ArrayNode;     }
bool isCollapsedNode()  const { return NodeType & CollapsedNode; }
bool isIncompleteNode() const { return NodeType & IncompleteNode;}
bool isCompleteNode()   const { return !isIncompleteNode();      }
bool isExternalNode()   const { return NodeType & ExternalNode;  }
bool isIntToPtrNode()   const { return NodeType & IntToPtrNode;  }
bool isPtrToIntNode()   const { return NodeType & PtrToIntNode;  }
bool isVAStartNode()    const { return NodeType & VAStartNode;   }
```

Other useful functions of this class are summarized below:

- `DSGraph* getParentGraph() const`: get the DSGraph of which this node is a part

- `unsigned getNumReferrers() const`: return the number of edges pointing to this node
- `unsigned getSize() const`: return the size of the largest value represented by this node
- `bool hasLink(unsigned offset) const`: return true if any value represented by this node may have a pointer to another node at the specified offset from the start of the value.
- `DSNodeHandle& getLilnk(unsigned offset)`: returns the DSNodeHandle associated with the node pointed to by this node at the specified offset
- `void markReachableNodes(llvm::DenseSet<const DSNode*>& ReachableNodes) const`: populates ReachableNodes with pointers to all DSNodes reachable from this node's edges.

4.5 DSCallSite

This class provides an interface to the call graph of the program as seen by DSA. The interesting functions are printed below.

```
/// isDirectCall - Return true if this call site is a direct call of the
/// function specified by getCalleeFunc.  If not, it is an indirect call to
/// the node specified by getCalleeNode.
///
bool isDirectCall() const { return CalleeF != 0; }
bool isIndirectCall() const { return !isDirectCall(); }


// Accessor functions...
const Function     &getCaller()     const;
CallSite            getCallSite()   const { return Site; }
      DSNodeHandle &getRetVal()           { return RetVal; }
const DSNodeHandle &getRetVal()     const { return RetVal; }
      DSNodeHandle &getVAVal()            { return VarArgVal; }
const DSNodeHandle &getVAVal()      const { return VarArgVal; }

DSNode *getCalleeNode() const {
  assert(!CalleeF && CalleeN.getNode()); return CalleeN.getNode();
}
const Function *getCalleeFunc() const {
  assert(!CalleeN.getNode() && CalleeF); return CalleeF;
}

unsigned getNumPtrArgs() const { return CallArgs.size(); }
```

4.6 Potential Pitfalls

The most important pitfall to using DSA successfully is to misunderstand the information returned by it. This is more of a problem when using a bottom-up-only analysis than when using a top-down analysis. When designing a client pass, one must take care not to infer things from DSA that are not implied by the information given. It may be advisable for a pass author to reread §**??** of this manual, especially if bottom-up DSA is to be used.

5 DSA Passes

Depending on their needs, clients may wish to use the results of any one of a number of DSA passes. It is important to understand the differences between them. The two most common passes are listed below. A client uses a pass by indicating that it requires its results in the normal LLVM manner:

```
EXAMPLE
```

5.1 EQTDDataStructures

This is the "final" form of DSA and is probably the appropriate pass for most external clients to use. By including this pass, a client will get the results of the DSA analysis after all phases of DSA have been run. Merging has been performed for aliasing in both callers and callees, so fewer nodes will be incomplete than in bottom-up. Aliasing information will therefore be most precise.

The primary disadvantage of using EQTDDataStructures is the loss of precision caused by the additional merging of nodes relative to bottom-up. Depending on the needs of a client, therefore, EquivBuDataStructures may be more appropriate in specific cases.

5.2 EquivBUDataStructures

This pass runs all forms of DSA except top-down propagation. The DSGraphs returned by this pass take into account aliasing caused by calleees of a function, but not by callers. Thus, this pass is not appropriate for use as a traditional not-shared/may-shared/must-shared alias analysis. However, as this pass is more context-sensitive than EQTDDataStructures, it may be a better choice for certain clients.