Unified Memory Analysis

Mark Marron Deepak Kapur Darko Stefanovic
Manuel Hermenegildo
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131, USA
{marron, kapur, darko, herme}@cs.unm.edu

April 2006

Abstract

The ability to accurately model the state of program memory and how it evolves during program execution is critical to many optimization and verification techniques. Current memory analysis techniques either provide very accurate information but run prohibitively slowly or run in an acceptable time but produce very conservative results. This paper presents an analysis method that is capable of accurately modeling many important program properties (aliasing, shape, logical data structures, sharing of data elements in collections) while maintaining an acceptable level of performance.

Using several examples we show how our abstract model is able to provide detailed information on the properties of interest in the concrete domain. We demonstrate that the model is a safe approximation of the concrete heap and outline the required data flow operations. Finally, we show that the asymptotic runtime of this method is polynomial in the size of the abstract heap and that in practice it is efficient enough to be used in an optimizing compiler.

1 Introduction

When performing program optimizations the ability to identify relationships between data structures has a significant impact on the effectiveness of the transformation. Research on optimization techniques from improved automatic memory management [HDH03], to optimizing the layout of data structures [LA05], to extracting thread-level parallelism [GH98] and scheduling to increase instruction level parallelism [KE93] has demonstrated the need for accurate information on a range of memory layout properties. These program transformations have used and introduced a number of different abstract properties. The simplest of these is the points-to relation that has been the main subject of alias analysis research [Ste96, CWZ90, LR92]. The work of Ghiya [GH96] used a notion of shape to enable the extraction of thread-level parallelism from common heap-based data structures. Lattner and Adve [LA03] focused on the identification of *logical* data structures on the heap to enable the identification and grouping of temporally related memory regions.

Reps and Sagiv [SRW99] introduced a powerful logical abstraction (3-Valued Logic Analysis), that is able to represent most of the shape and aliasing properties that are explored in the optimization literature. In addition to being expressive enough to model the range of relevant program properties, the TVLA framework is able to model the evolution of these properties through many important program idioms. In [LAS00], TVLA is used to model important operations on lists e.g. sorting, copying, destructive reversal and element insertion/deletion. Similar results can be obtained for other fundamental data structures such as arrays and trees. The ability to model these important transformations is a result of the notion of re-materialization which is used to transform summary representations into potentially more useful representations. This is done by transforming a single summary representation into one (or more) representations that explicitly identify the memory locations that a program is referring to in the summarized representation.

Unfortunately, the methods used in the TVLA approach result in large runtimes, both in theory and in practice. The combination of a logical model and the potential growth due to the number of possible re-materializations for a single summary representation results in unacceptably (in the context of an optimizing compiler) slow runtimes even on small programs. Additionally, logical domains are not amenable to context memoization for improving the performance of the analysis on large codes [WL95]. Recently there has been work on reducing the cost of running TVLA or restricted variations of the method [YR04, HR05] but they they have not been able to eliminate the exponential worst case runtime and have had mixed results in reducing the execution time on various benchmarks.

The goal of our research is to develop a memory analysis technique that will allow effective application of compiler optimizations that depend on accurate memory information. This requires an analysis technique that is efficient, that is capable of representing the properties of interest to various optimization passes, and that can accurately model the most common program idioms. In this paper, we present an abstract heap model that can represent information on aliasing, shape, logical data structures, sharing between variables, and sharing between data elements in collections. We then introduce a restricted version of refinement, based on the ideas presented by Sagiv, Reps and Wilhelm. Using this restricted notion of refinement, we demonstrate how this model can be used to accurately simulate important program events such as list copying, sorting, reversal, and various other destructive operations. Finally, we present a theoretical analysis showing that all of the model operations run in polynomial time, that the restricted version of refinement does not result in multiple new contexts, and that the method is efficient in practice (less than a second for common list operations and a few seconds for each of the Jolden benchmarks).

2 Concrete Domain

Our model is currently centered around the type safe, single inheritance, thread-free, exception-free, object-oriented imperative core of languages like Java or C#. Focusing on this simplified language enables us to focus on the central issues of the analysis and should allow the extension to a large class of source languages.

2.1 Basic Definitions

We will use the term *cell* to indicate an object or array on the concrete heap and *offset* to indicate an individual storage location in a cell. Thus, allocations return new memory cells of the given type and each field identifier or array index is at an offset in a cell. In this work we use pointers as the conceptual representation of the references to cells. We model variables as a mapping from variable names to cells in the heap. A concrete heap is then a tuple $(\{\text{cells}\}, \{\text{pointers}\}, V_m : \text{ident} \rightarrow \text{cell})$, and the concrete domain $H = \mathscr{P}(\{h | h \text{ is a concrete heap}\})$. Using these definitions we can view the concrete heaps as graphs where the cells are vertices and the pointers are the edges.

Given memory cells a, b and an offset o, $(a,o) \rightarrow_p b$ denotes a pointer p that is stored at offset o in a and points to b. We use $a \rightarrow_p b$ to indicate that $\exists$ offset o s.t. $(a,o) \rightarrow_p b$. Two cells can be connected by a *path* ψ. As with pointers we want to represent paths that start at a particular offset as well as paths that may start at any offset in a given node. Thus, we use $(a,o) \rightsquigarrow_\psi b$ to indicate the sequence of pointers $\langle p_1 \ldots p_n \rangle$ s.t. p_1 is stored in offset o in cell a, p_n points to b and $\forall p_i, p_{i+1}$ in the path p_i ends at the same cell, c_i, that p_{i+1} begins at ($\exists o'$ s.t. p_{i+1} is stored at o' in c_i).

We define $a \rightsquigarrow_\psi b$ to denote that $\exists o$ s.t. $(a,o) \rightsquigarrow_\psi b$. We will abuse the notion $\phi \subseteq P$ to denote that all the pointers in the path ϕ are contained in the set of pointers P.

We define a *region* of memory $\Re$ as a subset of the cells in memory, all the pointers that connect these cells and all the cross region pointers that start or end in this region. Given $C \subseteq \{c \mid c \text{ is a cell in memory}\}$, let $P = \{\text{pointer } p \mid \exists a,b \in C, a \rightarrow_p b\}$, i.e., the region defined by C is the subgraph (C,P) of the concrete heap graph. Then let $P_c = \{\text{pointer } p \mid \exists a \in C, x \notin C, a \rightarrow_p x \oplus x \rightarrow_p a\}$ With these definitions a region is then the tuple (C,P,P_c).

2.2 Reachability

Reachability is defined with respect to a single memory cell c and an offset in that cell o as: reachable$(c,o) = \{c' \mid \exists \text{ path } \psi, (c,o) \rightsquigarrow_\psi c'\}$. In some cases, we want to take the set of cells that are reachable from any offset in a cell, and we define reachable$(c) = \bigcup\{o | \text{reachable}(c,o)\}$. A useful corollary of the definitions of reachability and regions is that the set of cells reachable from a given cell defines a region of memory.

2.3 Connectivity

Connectivity within a region describes how cells in the region are connected. Given a region $\Re = (C,P,P_c)$ with cells $a,b \in C$, one of the following will hold:

- a and b are connected: if a, b are in the same weakly-connected component of the graph defined by (C,P).
- a and b are disjoint: if a, b are in different weakly-connected components of the graph defined by (C,P).

2.4 Structure Layout

A property that is important when performing program transformations is the layout of data structures in memory. This notion was explored in Ghiya's work [GH96] and examples of how this

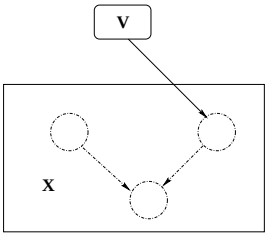


Figure 1: A Topological DAG, but List Shaped Region

information could be used were given in [GH98]. The basic idea is to track the layout of the heap as it appears to a program traversing a data structure. Thus, we introduce the notion of data structure layouts.

The notion of structure layouts is defined on regions of memory to allow for greater precision. Given a region the region $\Re = (C, P, P_c)$, we can define several layout predicates on the graph (C, P) to indicate what types traversal patterns a program can use to navigate through the data structures in the region. We will say a region admits a layout type if there is a subregion that satisfies the corresponding layout predicate. These layouts are not mutually exclusive. In the following definitions, let a, b be cells and ϕ, ψ be paths.

- Cycle Layout iff $\exists$ graph $(C', P'), C' \subseteq C, P' \subseteq P$ s.t. $\exists a \in C', \phi \subseteq P'$ s.t. $a \rightsquigarrow_\phi a$.
- MultiPath Layout iff $\exists$ graph $(C', P'), C' \subseteq C, P' \subseteq P$ s.t. $\exists a, b \in C', \phi, \psi \subseteq P'$ s.t. $(a \neq b) \wedge (\phi \neq \psi) \wedge (a \rightsquigarrow_\phi b) \wedge (a \rightsquigarrow_\psi b) \wedge (C', P')$ does not admit a Cycle Layout.
- Tree Layout iff $\exists$ graph $(C', P'), C' \subseteq C, P' \subseteq P$ s.t. $(\exists a \in C', a$ has 2 or more successors in $C') \wedge (C', P')$ does not admit a Cycle or Multipath Layout.
- List Layout iff $\exists$ graph $(C', P'), C' \subseteq C, P' \subseteq P$ s.t. $(\forall a \in C', a$ has one or zero successors in $C') \wedge (\exists b \in C', b$ has one successor in $C') \wedge (C', P')$ does not admit a Cycle or Multipath Layout.
- Singleton Layout holds for all regions. That is, a program can always safely treat a region as if it were a set of unconnected cells.

Note that Tree Layout $\Rightarrow$ List Layout $\Rightarrow$ Singleton Layout. Figure 1 shows a region X consisting of three concrete cells. This region has List and Singleton Layouts. It does not have Tree, MultiPath or Cycle Layouts.

A particular layout constrains the access patterns a program may have with respect to the data in a region. The goal of analyzing structure layout will be to exploit such constraints in various program optimizations. For instance, all accesses to the region X in Figure 1 are like list traversals.

3 Abstract Domain

The abstract model is based on the abstract heap graph approach, which has been presented numerous times in the literature [CWZ90, WL95, JM79]. The basic heap graph model uses nodes to

represent sets of concrete cells and edges to represent sets of pointers. We extend the basic heap graph model with a number of additional properties that allow us to track aliasing, reachability, and layout.

The presentation of the abstract model is divided into three subsections. We first present some component abstractions that model various properties that we want to track. Then, we show how to integrate all these component abstractions into the heap graph abstraction.

3.1 Component Abstractions

3.1.1 Numeric Quantities

The only requirement we place on the numeric abstraction is that it differentiates the case of exactly one and the case where there may be some other value. This gives the simple binary domain *one* $<$ # (unknown), where *one* represents the interval [1, 1] and # represents the interval $[-\infty, \infty]$.

3.1.2 Types

The possible types of the cells that a node represents are stored as a set of type names. When used in conjunction with the type hierarchy information, we use these types to model casts, `instanceof` tests and of course determine a restricted set of possible targets of a virtual function call.

3.1.3 Data Layout and Offsets

We want to model each field in a cell independently. Our pre-processing of the class declarations ensures that if two types have fields with the same name then one is a subtype of the other and has inherited that field.

We use a simple summarizing technique to handle arrays. Each array is allowed a single summary storage location called "?". All reads and writes are then assumed to be to this location. This summarization can lead to substantial information loss and some work has gone into solving this problem [GRS05]. In this paper we will not address how to more accurately model arrays.

With these definitions it is then easy to model the contents of a given cell as a set of abstract storage locations keyed on the field names of the types that the node may represent and the "?" symbol if the node may represent an array.

3.1.4 Abstract Layout

To track the concrete Structure Layout property, we introduce a simple domain of layout types $\mathit{Layouts} = \{\mathit{Singleton}, \mathit{List}, \mathit{Tree}, \mathit{MultiPath}, \mathit{Cycle}\}$. The abstract layouts can be given a simple total order: $\mathit{Singleton} < \mathit{List} < \mathit{Tree} < \mathit{MultiPath} < \mathit{Cycle}$. This order can be interpreted as: if a node n has abstract layout ξ then the concrete region, $\Re = \gamma(n)$, where γ is the concretization operator, may have any of the layout properties less than or equal to ξ. E.g. if we have a node with layout type *List* the concrete region may have the *List* or *Singleton* layout properties. If the node has *Cycle* as the layout then the concrete domain may have any of the layout properties. The abstract layout for a node n represents the most general concrete layout that may be encountered by a program traversing the region that is represented by the node n.

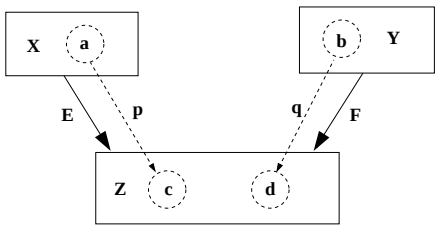


Figure 2: Edges Disjoint

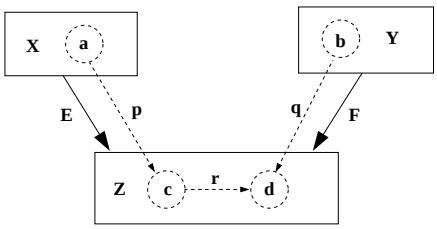


Figure 3: Edges In-Connected

3.1.5 Connectivity

Given the concretization operator γ and two edges e_1, e_2 that start/end at then node n, the predicates that define connectivity in the abstract domain are:

- e_1, e_2 connected if
  $(\exists p_1 \in \gamma(e_1) \wedge p_2 \in \gamma(e_2) \wedge a, b \in \gamma(n))$ s.t. $(p_1$ starts or ends at $a) \wedge (p_2$ starts or ends at $b) \wedge$ (a, b connected).

- e_1, e_2 disjoint if
  $\forall p_1 \in \gamma(e_1) \wedge p_2 \in \gamma(e_2) \wedge a, b \in \gamma(n)$ $((p_1$ starts or ends at $a) \wedge (p_2$ starts or ends at $b) \Rightarrow a, b$ are disjoint)

We then define two predicates that will be useful in later algorithms.
Two edges e_1, e_2 are *outConnected* in a node n if: (e_1, e_2 are both out edges from n) $\wedge$ (e_1, e_2 are connected in n).
Two edges e_1, e_2 are *inConnected* in a node n if: (e_1, e_2 are both in edges to n) $\wedge$ (e_1, e_2 are connected in n).

Figures 2 and 3 show an overlay of the abstract and concrete models. The concrete cells and pointers are shown as dotted circles and lines while the abstract nodes and edges are represented with solid boxes and lines. Edge E is an abstraction of pointer p, edge F is an abstraction of pointer q, while the node Z abstracts cells c, d. Nodes X, Y abstract cells a, b respectively.

In Figure 2 we can see that the targets of p, q (cells c, d) are disjoint. Thus by the definition of the connectivity abstraction, edges E and F are also disjoint.

In Figure 3 there is an additional pointer r, which connects cells c, d. This means that c, d are connected and in the abstraction, E, F are connected and thus E, F are also *inConnected*.

3.1.6 Interference

In the heap graph abstraction, each graph edge represents a set of inter-region pointers. To accurately track the layout when combining regions, it is important to track whenever the pointers that the edge represents all point into disjoint subregions or if there may exist a cell that two or more pointers may be able to reach and thus they *interfere*. An edge e represents interfering pointers if there exist pointers $p, q \in \gamma(e)$ such that the cells that p, q point to are connected. We will use a two-element lattice, $np < ip$, np for edges with all non-interfering pointers and ip for edges with two or more potentially interfering-pointers. This abstraction is a complement to the connectivity

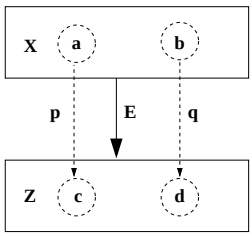


Figure 4: Edge Representing Non-Interfering Pointers

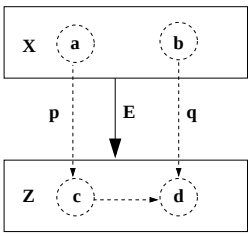


Figure 5: Edge with Interfering Pointers

relation, which tracks the relation between pointers represented by different edges while interference tracks the relation between pointers represented by the same edge.

Figures 4 and 5 show an overlay of the abstract and concrete models. In this example Edge E is an abstraction of pointers p and q, while the node Z abstracts cells c, d. Node X abstracts cells a and b.

In Figure 4 we can see that the targets of p, q (cells c, d) are disjoint. Thus the pointers do not interfere and the edge, E, that abstracts them should be np.

In Figure 5 there is an additional pointer r, which connects cells c, d. This means that c and d are connected and in the abstraction edge E should be ip.

3.2 Heap Graph Abstraction

The heap graph is simply a set of variables, a collection of nodes (which represent regions of the heap) and edges (which represent sets of pointers).

3.2.1 Nodes

The nodes represent regions of the concrete heap and are built from the component abstractions that have been presented above. The types of the concrete cells that a node may represent are stored in a set called *types*. To track the total number of cells that may be in the region represented by this node we use the *size* property. The internal layout of a node is represented by the *layout* component. Finally, we introduce a binary relation *connR* to track the connectivity of the edges that are incident to this node. That is if e_1, e_2 are connected with respect to this node then $(e_1, e_2) \in$ *connR*, if e_1, e_2 are disjoint in this node then $(e_1, e_2) \notin$ *connR*. We will assume that the *connR* relation is symmetric, thus $(e_1, e_2) \in$ *connR* $\Leftrightarrow (e_2, e_1) \in$ *connR*

The abstract domain for nodes is then records of the form `[types, layout, size, connR]`. For clarity we will omit a representation of the *connR* relation for the edges incident to this node. The inclusion of this information complicates the figures substantially and the information is only used a few times. In the cases where the connectivity relation is of interest we will mention it in the description of the figure.

3.2.2 Edges

The edges represent collections of pointers between regions of memory. As in the case of the nodes, we combine several of the component abstractions to create the final edge abstraction. The *offset*

represents where the offsets that the pointers that the edge represents are stored. The maximum number of parallel pointers that this edge represents is tracked with the *maxCut* property. The edge also tracks the possibility that it represents a pair of interfering pointers, *interfere*. Thus, we represent edges as records of the form $\{\texttt{offset, maxCut, interfere}\}$.

3.2.3 Graph

Given these definitions we can define abstract heap graphs as tuples of the form $(N \subseteq \text{ Nodes}, E \subseteq \text{ Edges}, V_n \subseteq \text{ Nodes}, M_e : E \to (N * N))$. For each variable in the program we create a node of type *var* for it and add an edge from it to it's target as given by the V_m map in the concrete domain. Finally, the function M_e defines the structure of the graph by mapping edges e to the pair of nodes (n_s, n_e) such that e begins at n_s and ends at n_e. We use the notation $M_e(e) = (*, n)$ or $M_e(e) = (n, *)$ in the case were we don't care about the identity of the start/end node of the edge. We will assume that the graph is consistent between the *connR* relation of nodes and the mapping induced by M_e. Which means that if $\exists n, e$ s.t. $(\exists e_x \in E, (e, e_x) \in n.connR) \Rightarrow M_e(e) = (n, *) \oplus M_e(e) = (*, n)$.

We will also restrict the abstract domain by defining a normal form the heap graphs. This normal form is useful in simplifying the structure of the abstract domain and it has several properties that will improve the accuracy of the analysis in practice.

First, we will define what it means for two nodes to to be *recursive*. The purpose of this definition is to make the abstract heap domain finite for a given program. This can be done by noting that the domains for the nodes and the edges are both finite and that if we can limit the maximum size of the graph structure then the number of graphs is finite. This can be done by forcing any recursive structure in the abstract domain to have a constant bounded representation.

Define two nodes $n, n' \in N$ to be *recursive* if:

- $\exists e \in E$ s.t. $M_e(e) = (n, n') \vee M_e(e) = (n', n)$.
- $n.types \cap n'.types \neq \emptyset$.
- $\nexists \mathrm{v}_n \in V_n$ s.t. $\exists e \in E, M_e(e) = (\mathrm{v}_n, n) \vee M_e(e) = (\mathrm{v}_n, n')$.

Another concept that is useful is the notion of ambiguous edges. We would like to be able to assume that given an offset and a node there is a unique edge that is incident to this node with that offset.

Define a node n as having an ambiguous offset if:
$\exists e, e' \in E$ s.t. $M_e(e) = (n, *) \wedge M_e(e') = (n, *) \wedge e.offset = e'.offset$.

A graph $g = (N, E, V_n, M_e)$ is then in normal form if:

- It has no unreachable nodes: $\forall n \in N, \exists \mathrm{v} \in V_n$ s.t. $(\exists$ path ϕ s.t. $\mathrm{v} \rightsquigarrow_\phi n)$.
- It has no recursive nodes: $\nexists n_1, n_2 \in N$ s.t. n_1, n_2 are recursive.
- It has no ambiguous edges: $\nexists n \in N$ s.t. n has an ambiguous offset.

This normal form definition restricts the set of elements in the abstract domain to ensure that there can be no unreachable and thus semantically meaningless components, that each node has uniquely labeled outgoing edges and that for any given program the all of the abstract heap graphs are of a bounded size.

4 Order on the Domain

In this section we define a partial order operator $\leq_h$ for our domain and show that it is monotone with respect to concretization. These restrictions when combined with the upper approximation operator $\tilde{\sqcup}$ which is defined in Algorithm 6 is enough to ensure that our data flow analysis is safe and terminates.

In this section we will first define some operations for nodes, edges and subcomponents of abstract heap graph. Then we will use the operations to define the $\leq_h$ operation and show that it is a valid partial order on the abstract heap graphs.

4.1 Edge Operations

We are using the edge in the abstract domain to represent set of cross region pointers in the concrete domain so we restrict all the operations in the abstract edge domain to pairs e, e' of edges such that $e.\text{offset} = e'.\text{offset}$ and that $M_e(e) = M_e(e') = (n_s, n_e)$.
The edge $\leq_e$ operator is simply a component wise comparison of each field in the edge structure. Thus, we define $e_1 \leq_e e_2$ if: $(e_1.\text{maxCut} \leq e_2.\text{maxCut}) \wedge (e_1.\text{interfere} \leq_{interfere} e_2.\text{interfere})$.

The join operation for edges $\sqcup_e$ is a pairwise join of the fields with a minor modification to model the potential for the two edges to be connected in the node they end at.
$\sqcup_e(e_1, e_2, areConn) = \{e_1.\text{offest}, e_1.\text{maxCut} \leq e_2.\text{maxCut}, e_1.\text{interfere} \sqcup e_2.\text{interfere} \sqcup areConn\}$.

4.2 Node Operations

4.2.1 Node Order

The $\leq_n$ operator is a simple component wise comparison. We define $n_1 \leq_n n_2$ if:
$(n_1.types \subseteq n_2.types) \wedge (n_1.layout \leq_l n_2.layout) \wedge (n_1.size \leq n_2.size) \wedge (n_1.connR \subseteq n_2.connR)$.

This definition depends in the *connR* relation in the two nodes having the same domain. We need to generalize this definition to the case where the domains may not be equal but there is a mapping between the two domains. Thus given a $f_e : Dom(n_1.connR) \mapsto Dom(n_2.connR)$ s.t. f_e is 1-1, define $n_1 \leq_n^{f_e} n_2$ if:
$(n_1.types \subseteq n_2.types) \wedge (n_1.layout \leq_l n_2.layout) \wedge (n_1.size \leq n_2.size) \wedge$
$((e, e') \in n_1.connR \Rightarrow (f_e(e), f_e(e')) \in n_2.connR)$.

4.2.2 Subgraph to Node Transform

Since the nodes are used to abstract regions in the concrete domain and a subgraph in the abstract heap graph can also be though of as abstracting some region in the concrete heap it is natural to

consider how a subgraph $\bar{g_s}$ of an abstract heap graph $\bar{g}$ can be transformed into a single node n.

First we need to define several functions that given a subgraph of the heap return information about paths between two nodes, how nodes can be connected and what traversal types a set of nodes may admit.

Suppose we have a subgraph (N_s, E_s, M_e) and two node $n, n' \in N_s$ then we say there is an edge path ϕ_e from n to n', $n \rightsquigarrow_{\phi_e} n'$ if:
$\exists \langle e_1, \ldots e_k \rangle, e_i \in E_s$ s.t. $M_e(e_1) = (n, *) \wedge M_e(e_k) = (*, n') \wedge$
$(\forall i \in [1, k-1], M_e(e_i) = (*, n_i) \Rightarrow M_e(e_{i+1} = (n_i, *) \wedge ((e_i, e_{i+1}) \in n_i.\text{connR})$.

We then define ϕ_e as an *multi-choice* path if, $\exists e_i \in \phi_e$ s.t. e_i.interfere = ip.

Suppose we have a subgraph (N_s, E_s, M_e) and two nodes $n, n' \in N_s$ then we can determine if n, n' may represent cells that are connected in the region that the subgraph represents. We will define n, n' *connectedR* in (N_s, E_s, M_e) if: $\exists$ an edge path $\phi_e, n \rightsquigarrow_{\phi_e} n'$.

If e, e' s.t. exactly one of the endpoints for e, say node n, and exactly one of the endpoints of e', node n', is a member of N_s and n, n' are *connectedR* in (N_s, E_s, M_e) then we will say e, e' are *connectedR* in (N_s, E_s, M_e)

Given an edge e and a node n we will say e' is a *successor* of e in n if:
$M_e(e) = (*, n) \wedge M_e(e') = (n, *) \wedge (e, e') \in n.\text{connR}$.

A node n is defined to have k successors if:
$\exists e \in E_s, o_e = \langle e_1 \ldots e_k \rangle \subseteq E_s$ s.t. $M_e(e) = (*, n) \wedge (\forall e' \in o_e, M_e(e') = (n, *)) \wedge (\forall e' \in o_e (e, e') \in n.\text{connR})$.

Given the subgraph (N_s, E_s, M_e) we can determine what types of traversals it may admit by looking at the traversals that the nodes in the region admit and the types of traversals that the connectivity information may admit. To determine the types of traversals that the node connectivity in this subgraph admits we proceed by defining layout predicates over the nodes.

- Cycle Traversal iff $\exists$ graph $(N', E', M_e), N' \subseteq N_s, E' \subseteq E_s$ s.t. $\exists n \in N', \phi_e \subseteq E'$ s.t. $n \rightsquigarrow_{\phi_e} n$.
- MultiPath Traversal iff $\exists$ graph $(N', E', M_e), N' \subseteq N_s, E' \subseteq E_s$ s.t. $(\exists n_a, n_b \in N', \phi_e, \psi_e \subseteq E'$ s.t. $(n_a \neq n_b) \wedge (\phi_e \neq \psi_e) \wedge (n_a \rightsquigarrow_{\phi_e} n_b) \wedge (n_a \rightsquigarrow_{\psi_e} n_b) \wedge (N', E', M_e)$ does not admit a Cycle Traversal) $\vee (\exists n_a, n_b \in N', \phi_e \subseteq E'$ s.t. $(n_a \rightsquigarrow_{\phi_e} n_b) \wedge e$ is a *multi-choice* path $\wedge (N', E', M_e)$ does not admit a Cycle Trave
- Tree Traversal iff $\exists$ graph $(N', E', M_e), N' \subseteq N_s, E' \subseteq E_s$ s.t. $(\exists n_a \in N', n_a$ has 2 or more successors in $N') \wedge (N', E')$ does not admit a Cycle or Multipath Traversal.
- List Traversal iff $\exists$ graph $(N', E', M_e), N' \subseteq N_s, E' \subseteq E$ s.t. $(\forall n_a \in N', n_a$ has one or zero successors in $N') \wedge (\exists n_b \in N', n_b$ has one successor in $N') \wedge (N', E')$ does not admit a Cycle or Multipath Traversal.
- Singleton Traversal holds for all abstract graphs.

With these definitions in place the transformation can be performed in a straight forward manner. We will denote this transformation as the function $summarizeSet(N_s, E_s, M_e)$. Given an abstract heap graph g take a subgraph (N_s, E_s, M_e) of (N, E, V_n, M_e) we can then determine the relevant node properties as follows:

- $types \leftarrow \bigcup\{n.\text{types} \mid n \in N_s\}$.
- $layout \leftarrow \bigsqcup\{n.\text{layout} \mid n \in N_s\} \sqcup \bigsqcup\{t \mid t, \text{ an admissible traversal in } (N_s, E_s, M_e)\}$.
- $size \leftarrow \sum\{n.\text{size} \mid n \in N_s\}$.
- $connR \leftarrow \{(e, e') \mid e, e' \text{ are } connectedR \text{ in } (N_s, E_s, M_e)\}$.

4.3 Partial Order

Given a heap graph we can partition it into a set of subgraphs Π_n. This partition induces a partition on the edges that start and end in different subgraphs defined by the partition Π_n. For each pair of partitions π_1, π_2 and each *offset* o let $\varepsilon_{(\pi_1,\pi_2,o)} = \{e \mid e.\text{offset} = o \wedge \exists n \in \pi_1.N, n' \in \pi_2.N\}$ s.t. $M_{e1}(e) = (n, n')\}$. Call this partition Π_e.

Given two abstract heap graphs $\bar{g_1} = (N_1, E_1, V_{n1}, M_{e1}), \bar{g_2} = (N_2, E_2, V_{n2}, M_{e2})$ we will say that $\bar{g_1} \leq_h \bar{g_2}$ if:

- $\exists$ a abstract heap subgraph $\bar{g_2'} = (N_2', E_2', V_{n2}', M_{e2}')$ s.t. $N_2' \subseteq N_2 \wedge E_2' \subseteq E_2 \wedge V_{n2}' = V_{n2} \wedge M_{e2}' = M_{e2}$, and
- $\exists$ a partition Π_n of $\bar{g_1}$ into subgraphs an isomorphism $\Phi_n : \Pi_n \mapsto N_2'$ and an isomorphism $\Phi_e : \Pi_e \mapsto E_2'$, s.t.
  - $\forall \pi_n \in \Pi_n, summarizeSet(\pi_n) \leq_n^{\Phi_e} \Phi_n(\pi_n)$.
  - $\forall \pi_{n1}, \pi_{n2} \in \Pi_n$, *offset* o, $\exists e' \in E_2$ s.t. $M_{e2}(e') = (\Phi_n(\pi_1), \Phi_n(\pi_2)) \wedge e'.\text{offset} = o \wedge \bigsqcup_e(\varepsilon_{(o,\pi_1,\pi_2)}, summarizeSet(\pi_2)) \leq_e e'$.
  - $\forall \pi_e \in \Pi_e, M_{e1}(\pi_e) = (\pi_{n1}, \pi_{n2}) \Rightarrow M_{e2}(\Phi_e(\pi_e)) = (\Phi_n(\pi_{n1}), \Phi_n(\pi_{n2}))$.
  - $\forall \text{v} \in V_{n1}, \Phi(\text{v}) =^{\Phi_e} n \Rightarrow n \in V_{n2}$.

4.4 Equality

We will assume a version of the node equality operator, $=_n^{\Phi_e}$, under an edge mapping exists with a similar definition to the $\leq_n^{f_e}$ operator. Two graphs $\bar{g_1} = (N_1, E_1, V_{n1}, M_{e1}), \bar{g_2} = (N_2, E_2, V_{n2}, M_{e2})$ are equal if $\exists$ isomorphisms $\Phi_n : N_1 \mapsto N_2$ and $\Phi_e : E_1 \mapsto E_2$ s.t.

- $\forall n \in N_1, n =^{\Phi_e} \Phi_n(n)$.
- $\forall e \in E_1, e = \Phi_e(e)$.

- $\forall e \in E_1, M_{e1}(e) = (n, n') \Rightarrow M_{e2}(\Phi_e(e)) = (\Phi_n(n), \Phi_n(n'))$.
- $\forall \ \mathtt{v} \ \in V_{n1}, \Phi(\mathtt{v}) =^{\Phi_e} n \Rightarrow n \in V_{n2}$.

4.5 Valid Partial Order

Given this definition it is then easy to show that the $\leq_h$ relation is a valid partial order.

- $\bar{g_1} \leq_h \bar{g_1}$: is trivial given $\Pi_n = ident$ and $\Pi_e = ident, \Phi_n = ident$ and $\Phi_e = ident$.
- $\bar{g_1} \leq_h \bar{g_2} \wedge \bar{g_2} \leq_h \bar{g_1} \Rightarrow \bar{g_1} = \bar{g_2}$: is slightly more complicated, if there is partition and isomorphism from the nodes/edges of $\bar{g_1}$ to a subgraph of $\bar{g_2}$ that respects the order of the nodes and a partition and isomorphism from the nodes/edges of $\bar{g_2}$ to a subgraph of $\bar{g_1}$ that respects the order of the nodes then that there is a isomorphism between the nodes of the graphs and similarly for the edges. Thus, $\bar{g_1} = \bar{g_2}$. This is not true in general but since we know that the nodes in the variable sets must match, that there can be no unreachable components and that each offset has a unique edge associated with it in a given node we know the mapping is forced to be isomorphic.
- $\bar{g_1} \leq_h \bar{g_2} \wedge \bar{g_2} \leq_h \bar{g_3} \Rightarrow \bar{g_1} \leq_h \bar{g_3}$: can easily be shown by composing the partitions and maps that are implied by the $\bar{g_1} \leq_h \bar{g_2}$ and $\bar{g_2} \leq_h \bar{g_3}$ relationships.

4.6 Concretization and Monotonicity

Given the definitions for the abstract and concrete domains we want to look at how they are related. We define a concretization function to take an element in the abstract domain $\bar{g}$ to an element in the concrete domain G. We will also show that this definition satisfies the monotonicity properties required to ensure that the abstract domain is a safe simulation of the concrete domain.

In Section 2 we defined the properties that we are interested in simulating in this data flow analysis. Then in Section 3 we defined several abstract properties and related them to the properties of interest in the concrete domain. The concretization operator γ enumerates the possible ways each node could be concretized into a region based on the semantics given to the node properties in Section 3, then enumerates all the possible ways the edges could be concretized into cross region pointers, again using the semantics given to the edge properties.

An important corollary of this definition of γ is that, if we consider the subgraph defined by (N_r, E_r, M_e) then $\gamma(N_r, E_r, M_e) \subseteq \gamma(\mathit{summarizeSet}(N_r, E_r, M_e))$. This lemma is simply derived by noting that the *summarizeSet* operation is an increasing function with respect to the properties in the nodes and is by definition a safe approximation of the properties of interest in the subgraph.

With this definition it is clear that, if $\bar{g_1} \leq_h \bar{g_2}$ then $\gamma(\bar{g_1}) \subseteq \gamma(\bar{g_2})$. Which follows from the corollary about the monotonicity and safety of the *summarizeSet* function.

5 Example: Build A List

In order to clarify some of the concepts presented in this section we will present a simple example to give some intuition into how the analysis and the model work. We will look at a simple loop

```
ListNode p, q;
p = null;
for(int i = 0; i < M; ++i)
{
    q = new ListNode();
    q.data = new DataNode;
    q.next = p;
    p = q;
}
```

Figure 6: Build a linked list

that constructs a linked list. The code to do this is shown in Figure 6.

Figure 7(a) shows the state of the abstract heap after allocating the `ListNode` (LN). We see that the variable `q` points to a node of type `ListNode` and since we just allocated the object that this node represents we know that it represents at most one cell and must have a *Singleton* layout. Figure 7(b) shows the state of the heap after allocating and assigning the data object, a cell of type `DataNode` (DN). Again it is a node of size one with a *Singleton* layout. The edge connecting them is stored at the `data` offset and again since it was just created it must represent a single pointer and be *np*. Finally, Figure 7(c) shows the heap at the end of the first loop iteration. In this figure we have assigned `p` to point to the newly created list entry and nullified the variable `q` since it is now dead.

Figure 7(d) shows the abstract heap at the end of the second loop iteration. We have created new nodes to represent the `ListNode` and `DataNode` cells allocated in this iteration of the loop. The newly allocated list entry has been put at the head of the list and the old list is linked in with an edge stored at the `next` offset. If we were to continue it is clear the heap abstraction would continue to grow in an unbounded manner. To prevent this, we will normalize the abstract heap. This is described in detail in Section 7 but all we need to know here is that we end up wanting to merge the two nodes with type `ListNode`. To do this we will replace the two nodes with a single summary node that represents both of them. Since both nodes are of type `ListNode` the resulting summary node should also be of type `ListNode`. By looking at the edge connecting the two nodes and the internal layouts we can determine that the internal layout of the new summary node should be *List* since we have two *Singleton* regions connected by an edge of size one. Since each region is of size one we know the summary region must be of size larger than one, represented by, *#*, in our abstract domain. Finally, we need to update the internal connectivity information for the new summary node. In particular we need to note that the two outgoing edges are *connected*. The state of the heap after this merge is shown in Figure 7(e).

After combining the list nodes we have ambiguous targets for the `data` abstract store location. We want to remove this ambiguity by merging the two potential targets into a single summary node and by combining the edges that refer to these targets into a single summary edge. The combination of these two nodes is similar to the combination of the list nodes except that we want to note that the two incoming edges are *disjoint*. After combining the nodes we need to combine the two edges. Since the new summary edge now represents two pointers its *maxCut* is now *#*. To determine the new value of the *interfere* property we need to see if either edge is *ip* or if the targets

of the edges are *inConnected*. Since we know that the edges pointed to disjoint nodes we know that they are not *inConnected* and therefore cannot *interfere*. Thus, we set the new summary edge as *np*. The result is shown in Figure 7(f) which is also the fixed point for the loop.

6 Refinement

During the data flow analysis we want to be able to summarize portions of an abstract heap graph into a single node in order to improve efficiency and to eliminate unbounded recursive data structures. This summarization could cause substantial losses in accuracy if it is overly aggressive. Instead of attempting to infer what summarizations are good and which would cause excessive losses in precision we instead define a method, that for most of the common cases encountered in a program, will allow us to undo the summarization of a subgraph.

The purpose of refinement is to transform summary representations into forms that make certain relationships explicit, so that the information in these relationships can be utilized more easily. In particular we want to turn summary nodes into a number of nodes of size one so that strong updates can be performed and exact relations between variables can be maintained. This section introduces a restricted notion of refinement that will allow for the accurate modeling of the common cases encountered during program analysis while remaining efficient.

In order to eliminate the state explosion that is possible with refinement, we adopt the approach of only doing refinement in those cases in which we can be sure that there is a unique way in which new nodes can be materialized. This limits the level of detail that can be achieved, as it is easy to demonstrate scenarios where refinement into multiple possibilities is needed to get results with the desired accuracy. Fortunately, it appears that these simple cases (where there is only a single way that refinement can be done) handle most of the situations that are encountered in practice. Thus, the results produced by the conservative method are in general near optimal.

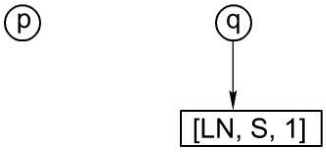
There are three kinds of layout types that we are interested in refining. The first is a summary node that represents several disjoint regions of the concrete heap. In this case we want to make each of these sub-regions into separate nodes in the abstract graph. The next layout is a list with a single incoming edge. In this case we want to make explicit the unique memory location that the variable must refer to in the list structure. Finally, we want to look at trees with a single incoming edge. This case is almost identical to the situation with the list and again we want to make explicit the unique target of the variable edge.

We will define the function, *refineNodeToRegion*(n,g), which takes a node and a graph and if possible replaces the node with it's refined representation in the graph, using the rules below.

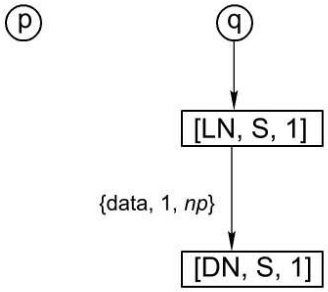
6.1 Disjoint Region Separation

It is possible for a single summary node to represent several entirely disjoint regions. If there is a partition of incoming edges (from variables or pointers) such that the edges can be partitioned based on the *inConnected* relationship, then we can transform this node into a number of nodes each one representing a single element of this partition.

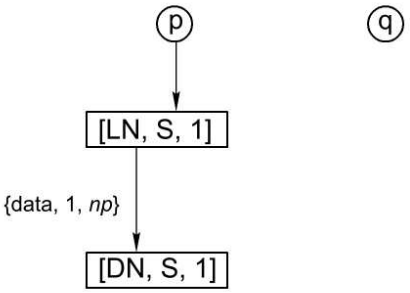
Formally, given a node *n* with the set of incoming edges *ies*, find the partition IEP s.t. e_1, e_2 are in the same equivalence class iff e_1, e_2 are in the transitive closure of the *inConnected* relation. Given this partition, we want to create a new node for each equivalence class. If the partition of in



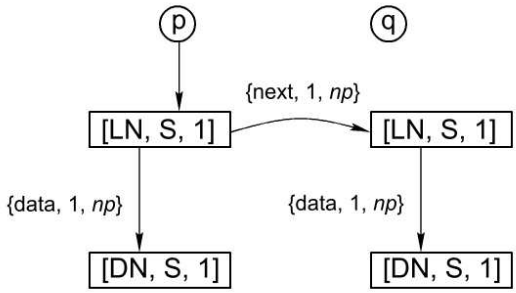
(a) After allocating the list node



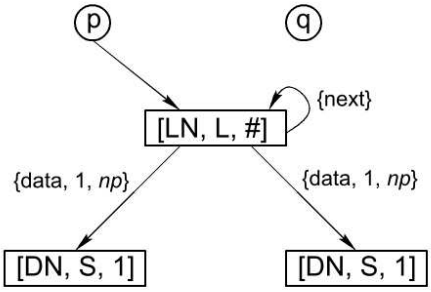
(b) After allocating the data object



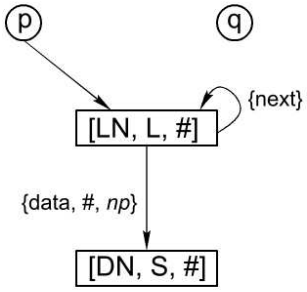
(c) At the end of the first iteration



(d) At the end of the second iteration

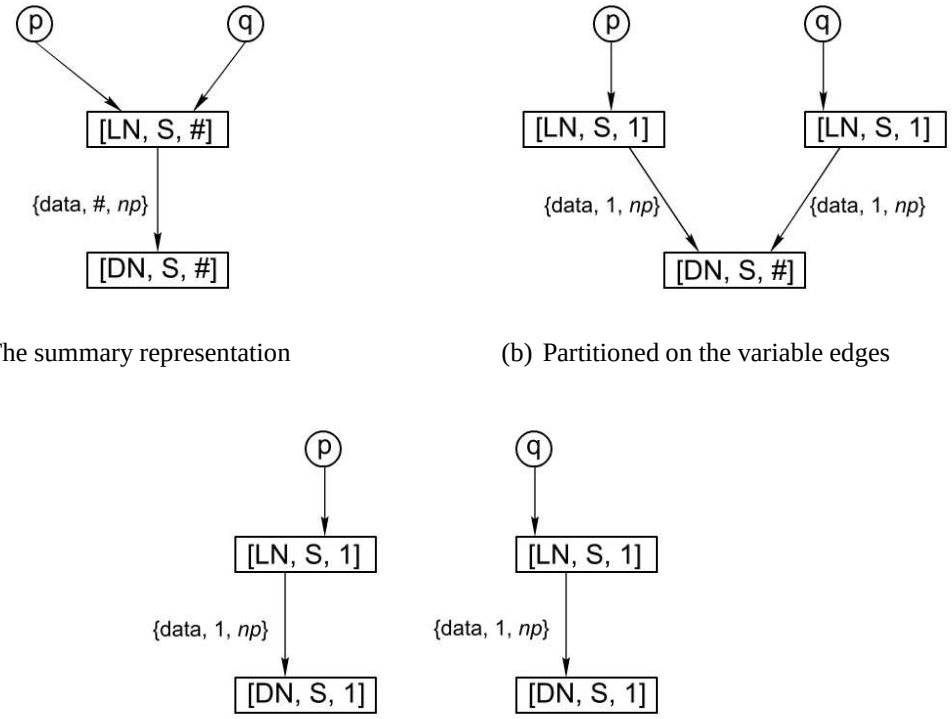


(e) After the first normalization step



(f) Finished

Figure 7: Building a linked list



(a) The summary representation

(b) Partitioned on the variable edges

(c) Partitioned on the pointer edges and done

Figure 8: Refinement of a region with disjoint sub-regions

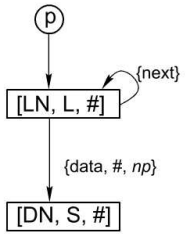
edges represents at most a single pointer, then we can safely assume that the node representing this partition represents at most one cell. This special case check is important to enable strong updates in later analysis steps.

For example consider the case in Figure 8(a) where the variables p and q point in to the same node, further assume that the edges from p and q are not *inConnected*. After partitioning we have the result in Figure 8(b) where we have partitioned the summary node based on the *inConnected* relation. Since the edge that was split contained all non-interfering pointers we know that the two edges incident to the node representing the *DataNode* cells cannot be *inConnected*. This now allows us to apply refinement again, the results are shown in Figure 8(c).

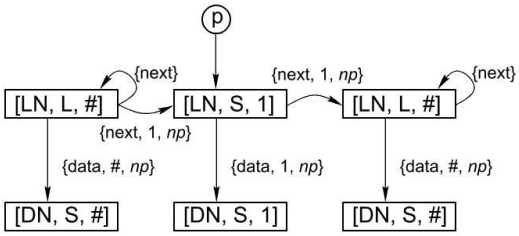
6.2 Refinement on Lists

Refinement on lists is slightly more complex than the refinement of singleton regions. Since we apply the singleton refinement prior to the list or tree refinement we know that the incoming edges to the given list node may potentially be connected. If there are multiple incoming edges we cannot determine an ordering for them in the list, or even if they must be connected in the list, so we only consider lists with a single incoming edge.

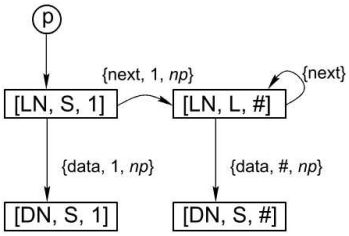
Figure 9(a) shows a list with one incoming variable. Figure 9(b) shows the most general way in which a list can be referred to by a single program variable; there is a single cell that the variable points to, then a section of the list after this cell and a section of the list that is before this cell.



(a) The summary representation



(b) Most general refined representation



(c) Removed unreachable components

Figure 9: Refinement of a list

We can safely ignore the section of the list before the cell that the variable refers to since it is unreachable and therefore cannot affect the program in any way. Thus we are left with the abstract graph in Figure 9(c).

6.3 Refinement on Trees

The case of refining trees is very similar to the refinement of lists. The implementation of these tree refinement is almost identical to the implementation of the refinement of a list so we omit the detailed description.

7 Dataflow Operators

This section defines the required data flow operators and presents selected proofs that they satisfy the required simulation properties, and thus are upper bound operators on the abstract domain lattice. We will first look at how to merge nodes and edges. Then we will define the normalization routines for nodes and graphs. Finally, we will use these to build the heap graph upper bound and comparison operations.

7.1 Edge Join

The edge join method is used when two edges start at the same offset in the same node and both end at the same node. The edge join method checks the end connectivity information to determine

how the component abstraction should be combined. Intuitively, if the edges are *inConnected* then the pointers that these edges represent may interfere and we should set the summary edge as *ip*, otherwise we only need to take the join of the *interfere* types of the edges. For the rest of the components that are used to represent an edge, we can simply combine them component-wise with respect to the possibility that these edges originated in separate graphs. That is, when we join two heap graphs that are from separate flow paths in the program we know that there can be no interaction between edges from different control contexts and we can utilize this fact.

In the first example, Figure 7(e) and 7(f) show how two edges with *disjoint* targets are joined. If the edges were *connected* we would have needed to set the *interfere* property to *ip* instead of *np* but otherwise the result would have been identical. Algorithm 1 outlines the code to handle this operation.

Algorithm 1: Join Edges ($\sqcup_e$)

input : g the heap graph, e_a, e_b edges
output: None
if (e_a, e_b *from the same context*) **then**
    e_a.maxCut $\leftarrow e_a$.maxCut $\widetilde{+}$ e_b.maxCut;
else
    e_a.maxCut $\leftarrow e_a$.maxCut $\sqcup$ e_b.maxCut;

$n_s \leftarrow$ the node the edges start at;
$n_e \leftarrow$ the node the edges end at;
if e_a, e_b *are inConnected in* n_e **then**
    e_a.interfere $\leftarrow ip$;
else
    e_a.interfere $\leftarrow e_a$.interfere $\sqcup_{\text{interfere}}$ e_b.interfere;

updateInternalConnInfoEdgeJoin(n_s, n_e, e_a, e_b);
deleteEdge(g, e_b);

The edge join algorithm uses the helper method *updateInternalConnInfoEdgeJoin*, which updates the internal connectivity info in n_s and n_e to represent the fact that e_a is now representing pointers from e_a and e_b.
$n_s.\text{connR} \leftarrow (n_s.\text{connR} \setminus \{(e,e') | e = e_b \vee e' = e_b\}) \cup \{(e_a,e') | (e_b,e') \in n_s.\text{connR} \vee (e_b,e') \in n_s.\text{connR}\}$.
And similarly for n_e.

The *deleteEdge* method simply removes the edge from the heap graph.

7.1.1 Proof of Edge Join Simulation Properties

The edge join operation may potentially affect all of the edge and graph connectivity properties that we are concerned with.

- Graph Structure: even though an edge is removed from the graph the reachability wrt. to the graph structure remains unchanged since we know that there is another edge that was

parallel to the one that was removed. This implies that the reachability in the graph structure is safely simulated.

- Internal connectivity: the internal connectivity of the start and end nodes is updated to ensure that e_a now safely simulates the connectivity relations for the edges that were joined. That is if a relation held for either edge prior to the join, it now holds for e_a. Thus, connectivity is safely simulated.

- Interference: if the endpoints of the edges are *inConnected* that implies that the endpoints may not be disjoint in the concrete domain. Thus the edges may interfere. Otherwise the edges may interfere if either edge contains interfering pointers. In both cases the algorithm handles the situation correctly.

7.2 Node Join and Combine

When we want to summarize two nodes, n_a and n_b, either because they are recursive or in the process of resolving ambiguous abstract storage locations, there are three distinct possibilities. The first is that neither node can reach the other. In this case we want to *join* them. If there are only edges in one direction between nodes, from n_a to n_b or n_b to n_a, then we want to *combine* them. If there are edges from n_a to n_b and from n_b to n_a, then we replace n_a, n_b with a single node n_c that is a conservative approximation of n_a, n_b.

Figure 7(e) and 7(f) show that the node join is a component-wise operation. The combine operation on a pair of nodes that have a connecting edge is more complicated as can be seen in Figures 7(d) and 7(e), where the two nodes with type `ListNode` are being combined into a single summary node. In particular we need to account for the fact that the edge(s) connecting nodes n_a and n_b will affect the layout of the new node and the internal connectivity in the new summary node. This means we need to compute the join of the contents, the new internal connectivity and the new internal layout for the node.

Algorithm 2: Combine Nodes ($\widetilde{+}_{\text{node}}$)

input : graph g, n_a, n_b nodes, *ebt* set of edges from n_a to n_b
output: None
n_a.type $\leftarrow$ n_a.type $\cup$ n_b.type;
n_a.size $\leftarrow$ n_a.size $\widetilde{+}$ n_b.size;
n_a.layout $\leftarrow$ computeCombineLayout(n_a.layout, n_b.layout, *ebt*);
n_a.connR $\leftarrow$ combineConnR(n_a.connR, n_b.connR, *ebt*);
remap all edges incident to n_b to be incident to n_a;
deleteNode(g, n_b);

The algorithm *combineLayout* is based on a casewise analysis of the internal layout for the node that results from the possible combinations of layouts for n_a, n_b and the relations between the edges in *ebt*.

The *combineConnR* function updates the internal connectivity information in n_a to represent the fact that it now represents the combined regions for n_a and n_b. This involves computing the binary connectivity relation for all the edges that are incident to the new summary node based on

Algorithm 3: computeCombineLayout

input : l_a, l_b layout types, *ebt* set of edges from n_a to n_b
output: the layout of the combined node
$mayInterfere \leftarrow \bigvee\{e \in ebt | e.\text{interfere} = ip\}$;
$totalCut \leftarrow \sum\{e \in \text{ebt} \,|e.\text{maxCut}\}$;
$isDAGgraph \leftarrow totalCut > 1 \wedge s_a \neq \text{Singleton} \wedge s_b \neq \text{Singleton}$;
$l_r \leftarrow l_a \sqcup_{\text{layout}} l_b$;
case *mayInterfere* $\vee$ *isDAGgraph*
    return $l_r \sqcup_{layout}$ *MultiPath;*
case $l_a = List$
    return $l_r \sqcup_{layout} Tree;$
case $l_a = l_b = Singleton$
    return $l_r \sqcup_{layout} List;$
otherwise
    return $l_r;$

the connectivity information in the argument nodes n_a, n_b, and the edges that connect the argument nodes, *ebt*.
$combineConnR(n_a, n_b, ebt) = \{(e,e') | (e,e') \in (n_a.\text{connR} \cup n_b.\text{connR}) \wedge e \notin ebt \wedge e' \notin ebt\}$
$\cup\{(e,e') | \exists e_b \in ebt \text{ s.t. } (e,e_b) \in n_a.\text{connR} \wedge (e_b,e') \in n_b.\text{connR}\}$.

7.2.1 Proof of Node Combine Simulation Properties

The node combine operation may potentially affect all of the node and graph connectivity properties that we are concerned with.

- Graph Structure: since all of the edges that were incident to n_a and n_b before the call are re-mapped to be incident to n_a, it is clear that the graph posses the same connectivity after the merging as it did before.
- Internal connectivity: assuming that *combineConnR* correctly updates the internal connectivity information wrt. the *ebt* edges from n_a to n_b, the internal connectivity simulation property is maintained.
- Layout: is computed via the *computeCombineLayout*. This method performs a simple case-wise analysis, the safety proof follows simply from the algorithm. So, assuming the correct behavior of this method layout property is safely simulated.

7.3 Normalization Operators

In order to simplify the abstract operators and the dataflow operations we define normal forms for the heap graphs and nodes. This section presents the method definitions that transform an arbitrary graph or node into its normal form.

Algorithm 4: Normalize Node

input : node n, graph g, $n \in g$
output: None
while $\exists$ *offset o with more than 1 edge* **do**
    $e_1, e_2 \leftarrow$ two edges with offset o;
    $n_1 \leftarrow$ endpoint of e_1;
    $n_2 \leftarrow$ endpoint of e_2;
    if $n_1 \neq n_2$ **then**
        if $\exists$ *edges from* n_1 *to* n_2 **then**
            combine(g, n_1, n_2);
        else
            join(g, n_1, n_2);

    $\sqcup_e(e_1, e_2, g)$;

Algorithm 5: Normalize Graph

input : graph g
output: None
Remove all unreachable nodes from g;
while g *is changing* **do**
    while $\exists$ *node n s.t. n can be normalized* **do**
        normalize(n, g);

    while $\exists$ *node n s.t. n can be refined* **do**
        refineNodeToRegion(n, g);

    while $\exists$ *nodes* n, n' *that are recursive* **do**
        combineNodes(g, n, n');

7.4 Heap Graph Upper Bound Operator

In order to join two heaps, we first normalize them and then mark which graph each edge and node belonged to originally. Then we take variables with the same name and union their targets. Once this is done the resulting graph is normalized.

Algorithm 6: Heap Graph Upper Bound, $\tilde{\sqcup}$

input : graph g_a, g_b
output: None
$g_{an} \leftarrow$ normalize(g_a);
set all nodes/edges in g_{an} as context a;
$g_{bn} \leftarrow$ normalize(g_b);
set all nodes/edges in g_{bn} as context b;
$g_{res} \leftarrow g_{an} \cup g_{bn}$;
normalizeGraph(g_{res});

7.4.1 $\tilde{\sqcup}$ is an Upper Bound Operator

Since the graph normalization preserves all of the simulation properties this upper bound must safely approximate all the simulation properties. Since the upper bound operator safely approximates all the properties from g_a and g_b then the result, $g_a \tilde{\sqcup} g_b$ must be a safe approximation of g_a and g_b, $g_a \leq_h g_a \tilde{\sqcup} g_b$ and $g_b \leq_h g_a \tilde{\sqcup} g_b$; where $\leq_h$ is the order operator on the abstract domain, which is defined in Appendix 4.3.

7.4.2 $\tilde{\sqcup}$ Satisfies Finite Chain Condition

We nee to show that the upper bound operation always has a finite ascending chain. This could be a problem either because the individual components fail to have a chain or the graph structure continues to change. Clearly, the abstractions of the nodes, edges have finite chains since all of the non-structural component domains are finite.

For the case of the graph structure we will assume that all recursive structures are singly recursive. This is a trivial limitation to remove, see [Deu94] for a solution. Since there are only a finite number of graphs with k or fewer nodes if we have a bound on the number of nodes any graph may have we have an upper bound on the height of a chain. Using non-recursive types it is clearly impossible to build an unbounded structure from a finite set of types, thus the only issue is in detecting recursive types but these are joined in the normalization routine and thus cannot grow in an unbounded fashion. Thus, the graph is bounded in size and the fixpoint computation halts.

7.5 Heap Graph Equivalence

Defining equivalence on the heap graphs is simple if we require that they are in normal form. This implies that each abstract storage location has a unique edge and we can then simply compare the graphs for structural equality and equality of the data in the nodes and edges.

7.6 Abstract Program Operators

The abstract simulation operators for the various concrete operators are for the most part simple translations of the effects of the concrete version. So, we will give high level descriptions of them and simply point out any subtleties but will omit any detailed formal description. In these operations we are assuming that the nodes that the variables `x` and `y` refer to are in normal form.

- `x = y;` Create an edge from `x` to the same node that the variable `y` refers to.
- `x = alloc(ty);` Allocate a new node with empty contents and the given type; `x` is assigned to refer to this node.
- `x = NULL;` Clear all the targets of `x`.
- `x = y.f;` Assuming `y` has a structure type this operation sets `x` to refer to the same node as the edge stored at field `f` in the node referred to by `y`. In the case that the edge stored at field `f` in the node pointed to by `y` refers to a node with a list or a tree layout we want to refine this node on the incoming edge before we do the assignment.
- `x = y[k];` This operator behaves similarly to the field load operator, the only difference is the use of range offsets and array typed nodes.
- `x.f = y;`
  If `y` is null then if possible we clear all the edges in the abstract store location or if this is not possible then nothing is done.

  If the target of `x` is not the same node as the target of `y`, this operation creates and stores an edge that refers to the same thing as `y` and then stores this edge in the abstract storage location in the node referred to by `x` with the offset `f`. In the case that the size of the node referred to by `x` one we can strongly update the storage location.

  If the targets of the variables are the same node then we need to update the internal connectivity information of that node as needed and model the effects on the interfere properties of all the other edges incident to the node.
- `x[k] = y;` This operator behaves similarly to the field store operator, the only difference is the use of range offsets and array typed nodes.

8 Example: Copy a List

In Section 3, we looked at how our method handles the analysis of a linked list construction routine. In this section we look at the more interesting example of copying a linked list (for simplicity we will create a reversed copy of the list). During the copy operation there are several attributes that we want to preserve: the original list should be unaffected, the resulting copy should be a list and if the input list contained all independent data elements then the copied list should contain all independent data elements as well.

Figure 10 is the code that we are going to analyze. For simplicity assume that we know that the source list is already pointed to by `p`.

```
ListNode q, x, t;
x = p;
q = null;
while(x != null)
{
    t = q;
    q = new ListNode();
    q.next = t;
    q.data = x.data;
    x = x.next;
}
```

Figure 10: Copy a linked list (in reverse)

Figure 11(a) shows the refined list that exists at the start of this code block. Figure 11(b) shows the results at the end of the first loop iteration. The head element of the list has been copied, since `t` is dead it is nullified and `x` has been indexed down the list. Note that in indexing down the list we have refined the list on the `next` edge so that the node that `x` refers to is made explicit (the node is a singleton of size 1).

Figure 11(c) shows the heap during the second iteration of the loop after creating new list node and assigning it to point to the next data node in the list we are copying. At the end of the loop 11(d) we have indexed the variable `x` again and nullified `t` since we know it is dead. Similarly to the case of the list copy we now have recursive nodes (for simplicity assume that we know that keeping `p`, `q` refined does not matter, if we keep them refined the result is the same, it just takes an extra loop iteration and results in a larger graph). Thus, we need to compress them during normalization. This results in the graph shown in Figure 11(e). This is clearly the fixed point of the loop and if we interpret the exit condition of the loop we can see that the result of the copy loop is the heap graph show in Figure 11(f).

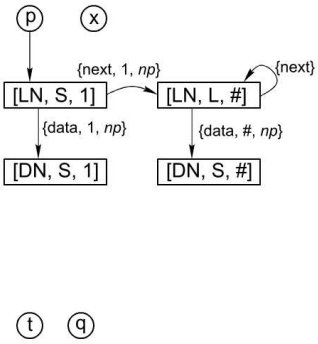
9 Performance

In order for this analysis to be of use in the context of optimizing compilers, it needs to be reasonably efficient and to have a well behaved worst case runtime. In this section, we will look at the asymptotic runtime of the method as well as its performance on some smaller benchmarks.

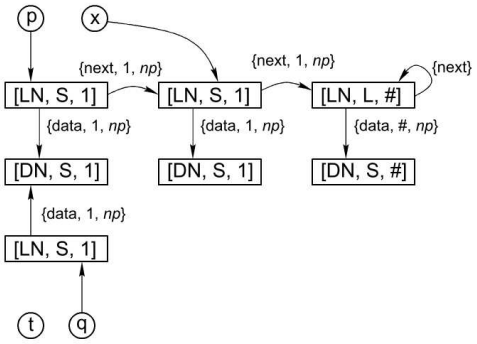
9.1 Theoretical Performance

In this section we will assume that the number of nodes in the abstract heap graph is n, that each node has at most max_k incident edges and that there are at most max_t types used in the program.

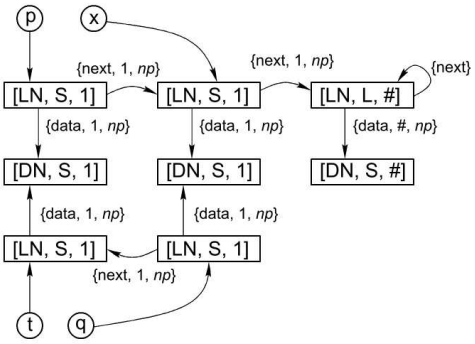
First, we look at the time it takes to join two edges. The data in the edges is very simple and can be joined in constant time. However, we need to update the connectivity information in the nodes that the joined edges are adjacent to. This update can take $O(max_k)$ time since the information between the joined edges and every other edge incident to the start and end nodes of the joined



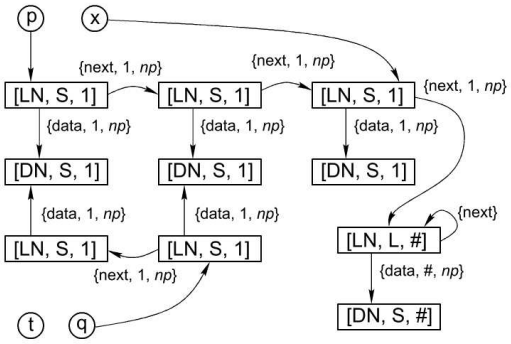
(a) At the start of the method



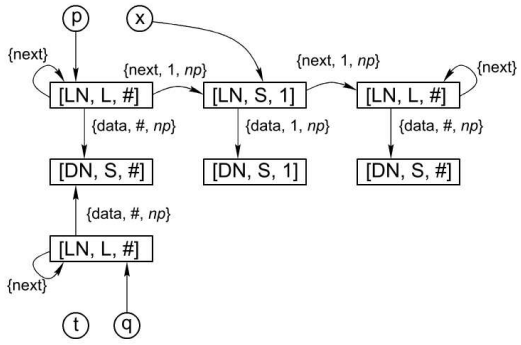
(b) After the first loop iteration



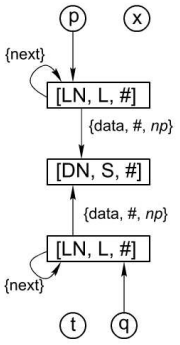
(c) After creating the copy node in the second iteration



(d) At the end of second the loop iteration



(e) After the normalization step



(f) Finished

Figure 11: Copying a linked list (in reverse order)

edges needs to be updated.

When combining two nodes, as described in Algorithm 4, we need to look at combining the type sets which is $O(max_t * log(max_t))$, at remapping the incident edges which is $O(max_k)$ and finally we need to handle the combination of the internal connectivity information which may take $O(max_k^3)$ operations. The total time is $O(max_t * log(max_t) + max_k + max_k^3)$. If we assume that max_t is a small constant we get the time to normalize a node is proportional to $O(max_k^3)$.

Since the abstract versions of the program operators have only local effects the most expensive part of executing any of them is removing any ambiguous edges in the abstract nodes that are affected. This simply requires joining at most max_k edges and nodes.

The dominant cost of the algorithm is the graph normalization, Algorithm 5, which requires removing all the unreachable sections of the heap graph $O(n * max_k)$, normalizing each node which is $O(n * max_k * max_k^3)$, refining all possible nodes $O(n * max_k)$ and finally removing all recursive nodes, which looks at each edge and if the types of the start and end node are the same combines them, so it takes time proportional to $O(n * max_k * max_k^3)$. These operations need to be done until none of them can be applied any more, which implies a total time of $O(n^2 * max_k^4)$ for the normalization routine.

The cost to join or compare two heaps is the cost to normalize the heaps plus the cost to union and normalize them, or the cost to do a traversal and pairwise comparison of each node and edge. In either case the cost of the operation is dominated by the cost to do the normalization, which is proportional to $O(n^2 * max_k^4)$.

9.2 Benchmarks

In this section we look at the actual performance of the algorithm on a number of benchmarks. All of our benchmarks were run on a Pentium M 1.5 GHz laptop with 1 GB of RAM. We have two general classes of benchmarks that we are looking at. The first set is a number of simple list manipulation methods that are useful for ensuring that the information computed by this analysis is accurate. Since some of them have been used in some of the TVLA work, we can get a rough comparison in performance. These list benchmarks include insertion, deletion, and find operations as well as copying and sorting. The goal is to ensure that the listness and data independence properties are preserved through all of these operations.

The next set of benchmarks is a selected set of problems from the jolden suite [CM01]. We selected some of the larger benchmarks from this suite to demonstrate that even though the algorithm is currently completely context sensitive, it is able to scale to non-trivial code sizes.

Table 12 shows the results for running each of the benchmarks. The first column shows the benchmark name, the second column is the runtime for the analysis method presented in this paper and the third column is the runtime given in [RSY04]. These benchmarks were also run on a Pentium M 1.5 GHz laptop with 1 GB of RAM (the TVLA benchmarks also include a check for null pointer dereferencing in addition to checking for the listness properties). In all of the simple list tests, our analysis is able to maintain the desired listness properties and in the olden benchmarks it is able to identify many useful properties of the program including the bipartite graph in the em3d benchmark.

Benchmark	UMA	TVLA
Find	0.10	22.3
Insert	0.28	41.2
Delete	0.20	42.0
Reverse	0.31	23.7
Append	0.20	NA
Copy	0.39	NA
Quick Sort	1.50	NA
treeadd	1.80	NA
health	4.60	NA
em3d	3.40	NA
power	4.03	NA

Figure 12: Assorted List and jolden analysis times

10 Conclusion

This paper presented a graph based model of the concrete heap that is capable of representing the heap properties (aliasing, shape, sharing, data dependencies, logical data structure identification, etc.) needed to perform most optimizations. We then introduced a more restricted version of refinement than has been used in previous work. This version of refinement is able to prevent the explosion in the number of contexts that can occur during analysis while still enabling the accurate modeling of important program operations (copying, sorting, destructive updates, etc.). In particular we presented in detail an example of copying a list where the method is able to determine that given a list of distinct data elements the result is a list of distinct data elements. Similar levels of accuracy are obtained for the quick-sort, destructive list reversal and list append benchmarks.

We then turned to the issue of performance. Theoretical analysis of the method shows that all the program operations on the model are $O(max_k^2)$ and the join/equality operations are $O(n^2 * max_k^4)$ where n is the number of nodes in the heap graph and max_k is the maximum number of edges incident to any node. Further, by construction, the refinement operation does not increase the number of contexts that need to be analyzed. Although our interprodcedural analysis is fully context sensitive, past work [WL95] has shown that heap graph based approaches are amenable to context memoization and we have made sure that the construction of our model does not interfere with the effectiveness of these approaches.

Finally, we looked at several benchmarks. The first set was a number of micro-benchmarks designed to test the ability of the analysis method to accurately model important program operations. The method was able to analyze this set quickly while maintaining all the relevant properties. Next, we ran the several codes from the jolden benchmark suite. These range in size from a few hundred lines to a around a thousand lines of code. Our runtimes on these benchmarks never exceed several seconds even using a fully context sensitive interprocedural analysis method. In addition, the analysis we presented is capable of discovering sophisticated properties of many of the benchmarks. In the list benchmarks, the algorithm is able to successfully maintain the list property of the regions, is able to distinguish between different lists and is able to determine the independence of the data elements in the lists. In the em3d benchmark, the method correctly determines that the

graph structure in method *compute* is a bipartite graph between two distinct linked lists and that the *coeffs* and *fromNodes* arrays are unique to each node in these lists.

Acknowledgements

The first author would like to thank Jack, John, Mario and Stefan for their input and comments on the work presented in this paper.

References

[CM01] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in java. In *IEEE PACT*, pages 280–291. IEEE Computer Society, 2001.

[CWZ90] David Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures (with retrospective). In McKinley [McK04], pages 343–359.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond -limiting. In *PLDI*, pages 230–241, 1994.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL*, pages 1–15, 1996.

[GH98] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *POPL*, pages 121–133, 1998.

[GRS05] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In Palsberg and Abadi [PA05], pages 338–350.

[HDH03] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In Ron Crocker and Guy L. Steele Jr., editors, *OOPSLA*, pages 359–373. ACM, 2003.

[HR05] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In Palsberg and Abadi [PA05], pages 310–323.

[JM79] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In *POPL*, pages 244–256, 1979.

[KE93] Daniel R. Kerns and Susan J. Eggers. Balanced scheduling: instruction scheduling when memory latency is uncertain (with retrospective). In McKinley [McK04], pages 515–527.

[LA03] Chris Lattner and Vikram 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, Apr 2003.

[LA05] Chris Lattner and Vikram S. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 129–142. ACM, 2005.

[LAS00] Tal Lev-Ami and Shmuel Sagiv. Tvla: A system for implementing static analyses. In Jens Palsberg, editor, *SAS*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer, 2000.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing (with retrospective). In McKinley [McK04], pages 473–489.

[McK04] Kathryn S. McKinley, editor. *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*. ACM, 2004.

[PA05] Jens Palsberg and Martín Abadi, editors. *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. ACM, 2005.

[RSY04] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural functional shape analysis using local heaps. Tech. Rep. 26, Tel Aviv Uni., November 2004. Available at "*http://www.math.tau.ac.il/∼maon*".

[SRW99] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118, 1999.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *PLDI*, pages 1–, 1995.

[YR04] Eran Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In William Pugh and Craig Chambers, editors, *PLDI*, pages 25–34. ACM, 2004.

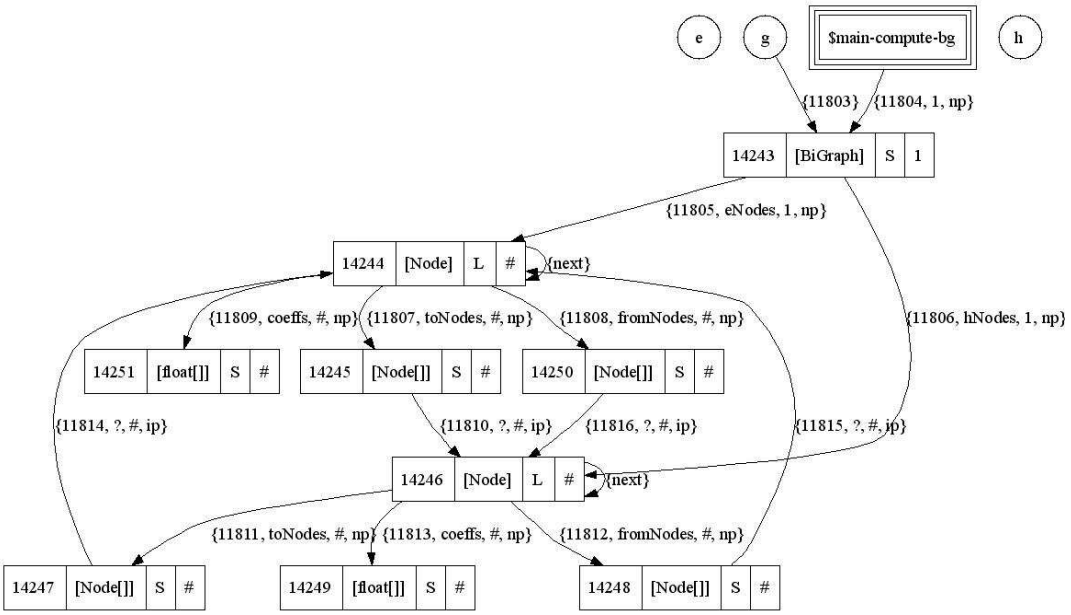


Figure 13: Heap at entry to compute method

A Application to em3d

This section presents an example of applying our analysis method to the *em3d* benchmark from the *Jolden* benchmark suite. For this benchmark we took the results generated by our analysis and, by hand but in a purely mechanical fashion, applied some of the simple transforms suggested by Ghyia et. al. for doing thread level parallelization of the list traversal during the *compute* method. For em3d, we also used the fact (which is obtainable by a simple analysis) that only reads are done from the *coeffs* and *fromNodes* arrays to unroll the key loop in the node *computeNewValue* method.

Our analysis code is able to automatically generate the abstract heap graphs which we use for debugging and evaluation purposes. The structure of these graphs is almost identical to the format described in Section 3. Figure 13 shows the abstract heap graph generated from our analysis at entry to the *compute* method. In this figure you can see the variable g points to the *BiGraph* object with the *eNodes* field pointing to one list of nodes and the *hNodes* field pointing to another disjoint list of nodes. Each of the nodes in these list regions have 3 sub-component arrays and the analysis determines that none of these arrays are shared between nodes in the list.

In testing the effectiveness of the parallelization we ran the code on at dual processor 1.8GHz G5 machine.

These results in Table 14 show very effective identification and exploitation of the parallelism in em3d. Although the speed-up is not difficult to exploit once the lists have been identified, the ability to identify them is a substantial step as the structure of the heap is non-trivial and the authors are not aware of any benchmarks for automatic parallelization of the serial em3d benchmark.

-n	-d	-i	Orig.	Parallel 2x
500	100	20	0.032	0.014
500	200	20	0.060	0.025
1000	200	20	0.124	0.053
1000	500	20	0.289	0.110
2000	200	20	0.270	0.129
2000	500	20	0.664	0.283
4000	200	20	0.663	0.262
4000	500	20	1.552	0.654

Figure 14: Serial and 2x parallel em3d runtimes