Workshop on Lanuguages and Compilers for Parallel Computing of '93.

Analysis of Dynamic Structures for Efficient Parallel Execution*

John Plevyak Vijay Karamcheti Andrew A. Chien

Department of Computer Science†
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

Programs written in high-level programming languages and in particular object-oriented languages make heavy use of references and dynamically allocated structures. As a result, precise analysis of such features is critical for producing efficient implementations. The information produced by this analysis is invaluable for compiling programs for both sequential and parallel machines.

This paper presents a new structure analysis technique handling references and dynamic structures which enables precise analysis of infinite recursive data structures. The precise analysis depends on an enhancement of Chase et al.'s Storage Shape Graph (SSG) called the Abstract Storage Graph (ASG) which extends SSG's with choice nodes, identity paths, and specialized storage nodes and references. These extensions allow ASG's to precisely describe singly- and multiply-linked lists as well as a number of other pointer structures such as octrees, and to analyze programs which manipulate them.

We describe program analysis to produce the ASG, and focus on the key operations: the transfer functions, summarization and deconstruction. Summarization compresses the ASG in such a way as to capture critical interdependencies between references. Deconstruction uses this information, stored by identity paths and refined references and nodes, to retrieve individual nodes for strong updates.

1 Introduction

Compilation, the translation of high-level programming languages to produce efficient program implementation in machine executable code, plays a major role in supporting the widespread use of computer systems. Optimizing compilers that transform programs to increase execution efficiency allow users to focus on high-level concerns rather than low level machine detail while achieving acceptable levels of efficiency. An essential element in such compilers is accurate program analysis which provides the compiler with information as to which program transformations may be safely applied without changing the program's functional behavior. Thus, good program analysis is necessary for effective optimization and efficient implementations.

Recent developments in programming languages and the widespread use of computing in non-numeric applications have produced a broad class of computer applications and algorithms. These applications and algorithms make use of a growing diversity of sophisticated data structures which rely on dynamic storage allocation and the use of references as basic tools for efficiency and expressibility. Though their use is ubiquitous, references and dynamic storage allocation raise difficult problems in program analysis. In this paper we focus on structure analysis, the problem of building a safe approximation of the program's run time data structures. Structure analysis is used to direct program optimization at compile time, consequently its accuracy is critical to improving code efficiency.

†E-mail: {jplevyak,vijayk,achien}@cs.uiuc.edu

*The research described in this paper was supported in part by National Science Foundation grant CCR-9209336, Office of Naval Research grant N00014-92-J-1961, and National Aeronautics and Space Administration grant NAG 1-613. Additional support has been provided by a generous special-purpose grant from the AT&T Foundation.

Structure analysis estimates at compile time the shape of the runtime store. Alias analysis is a related problem, but is subsumed by structure analysis since alias relations are described by the abstract store. However, the shape of unaliased dynamic structures cannot in general be inferred from alias information alone. While the alias information produced by structure analysis suffices for traditional serial optimizations and for dependence analysis based parallelization, the unique structure information is required for other important optimizations. Since determining precise alias information even in a single function is NP-complete [LH88, Mye81] practical structure analysis algorithms approximate the program store.

2 Background

2.1 Related Work

Most of the work on structure and alias analysis has been based on a data flow analysis framework. Such a framework defines a lattice of possible structure or alias situations, a function for each part of a program which models the effect of that part on an element of the lattice, and a meet operator which combines two elements into one.

Three basic approaches to both structure analysis and the alias problem have been explored. These basic approaches are (1) explicit annotation [Lar89] (2) access paths [HN90] and (3) graph based approaches [CWZ90]. Explicit annotations allow the user to supply the compiler with information which either cannot be derived, or is more easily verified than derived. In access path approaches, the aliases at each program point are described by pairs of access paths. In the graph based approaches, an approximation of the entire heap is computed for each program point and the graph nodes and edges express the possible sets of aliases and structure relations at execution time.

Many algorithms also use a combination of these approaches. For instance, the Abstract Dynamic Data Structure description (ADDS) approach [HNH92] combines annotation with access paths and seeks to verify programmer assertions. Larus [LH88] uses graphs to represent the structures reachable from each program variable, but labels the nodes with access paths and uses the labels to identify aliases. In contrast, Chase [CWZ90] uses graphs alone in which each node in the graph corresponds to one or more nodes in the runtime heap and the edges correspond to references. In this paper we do not consider annotations. The addition of program annotations is orthogonal to the problem of analyzing an unannotated program since for any imprecise algorithm, annotations could be added which guide and assist analysis at the cost of programmer effort.

In recent work, access path and graph based approaches have been drawn together [CBC93]. With access paths, the aliases at a program point are described by alias relations (pairs of access paths). An access path is a tuple consisting of a cell and a sequence of fields. The cell is either a program variable or a storage location name and the set of alias relations determines the edges of a directed graph. If the alias relations are stored with only one level of dereferencing, the aliases are precisely the edges of a graph whose vertices are the starting cells of the access paths [CBC93]. As a result, access paths are used to implement an essentially graph based algorithm.

Current alias and structure analysis algorithms are unable to accurately analyze many common data structures. Structure analysis approaches based on k-limited graphs [JM81, LH88, HPR89] or on k-limited naming schemes [CBC93] are unable to adequately describe recursive structures such as lists; in general, structures extend beyond the k-limit and the summary process destroys all structure beyond k nodes. If the list is traversed beyond k elements, the analysis cannot preserve the list structure, losing precision in analysis which leads to missed opportunities for optimization.

Chase et al.'s SSG algorithm [CWZ90] circumvents the k-limitation by augmenting the basic reference graph with heap reference counts. Thus, for singly-linked structures, SSGs can sometimes obtain a precise analysis of the program. However, the SSG algorithm suffers from two major limitations: it cannot analyze singly-linked lists in the face of mutation nor can it precisely analyze multiply linked structures.[1]. To remedy these limitations, the ASG incorporates extensions to the SSG (not based on heap reference counts). These extensions are described in detail in Section 3.

2.2 Project Context

This work has been done as part of the *Concert* Project [CKP93]. The objective of the *Concert* system is to achieve efficient, portable implementations of fine-grained concurrent object-oriented languages on parallel machines. Current commercial multicomputers are the primary target and present a variety of difficult problems since the cost of fine grained synchronization, consistency, communication and context switching in these machines can easily overwhelm that of computation. Controlling these costs requires determining the runtime shape of program structures and transforming both them and the program to increase the computation grain size.

These transformations fall in the general category of optimizations addressed via *grain size tuning* [CFKP92] which seeks to match the amount of serial processing between communication or synchronization points in the program to that efficiently supported by hardware. In parallel programs, merging computational grains requires enhancing data locality. For programs with complex data structures, accurate structure information at compile time allows identification of collections of data which are transformable for enhanced locality. Two examples of these transformations and the structure relations which facilitate them are given below:

object fusion When an object (dynamic structure) has an invariant reference to an unaliased object, the second object may be fused with (allocated as part of) the first object. A special case of object fusion is *tiling* in which recursive data structures consisting of unaliased objects can be blocked along dimensions and treated as a unit. Operations on the resultant object are scheduled as a single computation grain.

object clustering When a group of objects is identified which exhibit strong internal cohesion, they can be designated as a *cluster* for scheduling, placement and migration purposes.

These data transformations not only increase the execution grain size, they also increase the effectiveness of traditional optimizations such as instruction scheduling and register allocation which are more effective on the resulting larger schedulable units of instructions.

2.3 Overview of the Paper

The general mechanism of the reference graph based approach to structure analysis follows a standard dataflow analysis framework consisting of an abstract store as the lattice, abstract interpretation as transfer functions, and a safe merge as the meet operator. The approach combines an approximation[2] of the runtime store with an approximation of the effect of the program.

In this paper, we describe an extension of the generic reference graph (called the Abstract Storage Graph) which can model multiply-linked infinite structures precisely in a finite representation. Our structure analysis algorithm interprets the effects of the program on the abstract store while preserving the information which it contains. The algorithm deconstructs the finite representation to reveal portions of the infinite structure enabling precise updates.

In Section 3, we discuss the Abstract Storage Graph (ASG). Section 4 describes the construction of the ASG through abstract interpretation of program statements. Section 5 presents the finite summarization and iteration mechanisms. The safety and complexity of the intraprocedural algorithm are discussed in Sections 6 and 7. Finally, we conclude with current status and future work in Sections 8 and 9.

[1] An extension to heap reference counts is mentioned to handle the construction of multiply-linked structures, but is not developed.

[2] In this paper *approximation* should be taken to mean *safe approximation* in the sense of describing or creating at least as many aliases as that which it approximates.

3 Abstract Storage Graphs

A store representation finitely approximates the reference pattern of a program at an execution point. It must safely approximate the pattern so that transformations which use the reference information will preserve program semantics. In this section, we describe our store representation, the Abstract Storage Graph (ASG), and relate it to Chase et al.'s Storage Shape Graph (SSG) [CWZ90].

A generic reference graph is a store representation containing nodes which model dynamic program structures, and edges which model references. In order to produce a precise approximation, the reference graph must preserve as much deterministic reference information as possible. Determinism in the graph refers to the knowledge that a particular reference *does* exist as opposed to *may* exist.

The ASG is a variant of the generic reference graph with the following extensions which enable it to preserve deterministic reference information:

- **single nodes** and **summary nodes** refine generic storage nodes.
- **deterministic references** and **nondeterministic references** refine generic references.
- **choice nodes** encode unique alternatives between references.
- **identity paths** are used to annotate summary nodes preserving their internal structure.

The nodes of the ASG are *variables*, *single nodes*, *summary nodes* and *choice nodes*. The edges represent different types of references between nodes and include *deterministic* (d-references) and *non-deterministic* (n-references) references[3]. The different components of the ASG are described below (Figure 1 shows the symbols used in the rest of the paper).

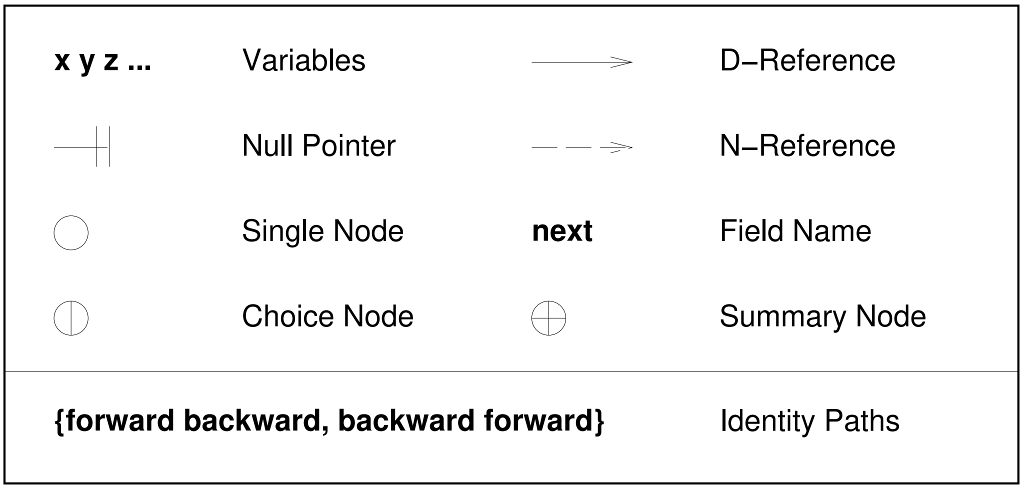

Figure 1: Abstract Storage Graph Elements

References can have two orthogonal attributes. All references incident on storage nodes have an attribute indicating whether or not they are deterministic, and all references leaving storage nodes have a field name attribute. Deterministic references indicate that a reference emanating from exactly one node enters exactly one instance of the target node[4]. Nondeterministic references indicate that a node may be referenced zero or more times. N-references arise from control flow confluence, summarization

[3] One restriction is that variables may not be the target of references.
[4] Contingent on the existence of both the source and destination nodes

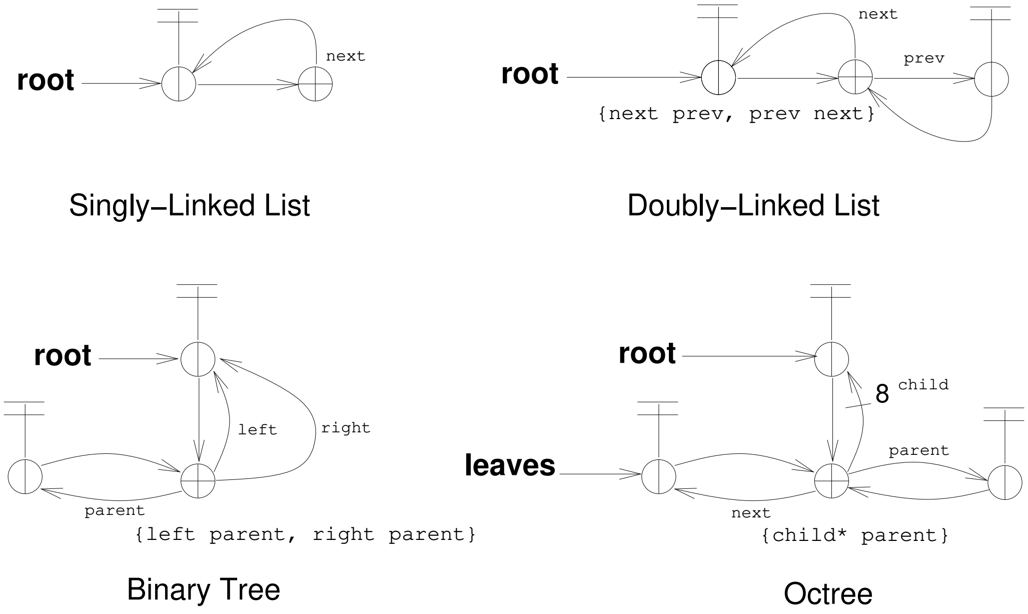

Figure 2: ASG Examples

and other sources of imprecision. They correspond to the traditional definition of a reference in a generic reference graph where nodes represent multiple pieces of storage.

The nodes of a generic reference graph (here called **storage nodes**) are refined into summary nodes and single nodes which correspond to single pieces of storage. A single node of a particular type can have only one field reference for each field in its type, and this reference refers to only one node (choice or storage). Summary nodes correspond to an indeterminate number of pieces of storage of the same type. Summary nodes arise when storage nodes are combined, and may be deconstructed (uncombined) while preserving deterministic reference information. A d-reference to a summary node indicates that each summarized node is referenced exactly once along that arc.

Choice nodes describe a set of disjoint possibilities and along with identity paths are used to avoid unnecessary introduction of non-determinism into the ASG. Choice nodes are introduced during the combining phase of the algorithm, preserving the two sets of disjoint alternatives represented by the nodes being combined. For instance, if two nodes each of which have a single deterministic reference are combined, a choice node is inserted between the two incoming references and the combined node. The choice node preserves the information that the combined node can have only a single incoming deterministic reference; thus, the two references which come into the choice node are unaliased. Note that converting the two references into a non-deterministic reference would have resulted in a loss of information. Since choice nodes do not model real program structures, references which are incident on choice nodes define connectivity. A storage node is reachable from an particular reference if a path can be found through a set of choice nodes to the storage nodes.

Identity paths are pairs of labels which describe cycles in the reference graph. Identity paths preserve the internal structure of summary nodes enabling storage nodes to be summarized and deconstructed while preserving d-references between them. This allows, a precise, finite representation of some infinite data structures. For instance, in a doubly-linked list, the identity path **(next prev)** describes the relationship that the previous node of any next node is the node itself. Reversing labels in an identity path does not always produce a valid identity path. For example, in a binary tree, **(left parent, right parent)** is a valid identity path, but **(parent left)** is not (see Figure 2. Section 5.2 shows how identity paths can be used to extract a list of storage nodes from a single summary node without introducing non-determinism.

3.1 Examples

ASGs can model many common recursive data structures precisely in a small amount of space. Some examples of different structures and how they are modeled using an ASG are shown in in Figure 2. A singly-linked list is modeled with a choice node and a summary node. The key piece of information is that each node has only one incoming reference (a reference count of one); this describes the structure as acyclic. For a doubly-linked list the ASG consists of two choice nodes and a summary node annotated with the identity paths **(next prev, prev next)**. The key information is that (*i*) the reference from *root* shares the d-reference incident on the summary node with the summary node's own **next** reference. This means that the one of the summarized nodes is pointed to directly by *root*, and the list which follows is unaliased along **next**. And (*ii*) the identity paths preserve the information that the references in the **next** and **prev** fields are inverses of each other. As a result, we can conclude that the graph accessible from *root* is indeed that of a doubly-linked list.

3.2 Comparison with SSG

ASGs and SSGs model similar information, but our analysis produces ASGs which contain more information than SSGs. An ASG can be transformed into a comparable SSG by removing information. For instance, the ASG which describes a singly-linked list can be transformed to the SSG for the same structure (see Figure 3).

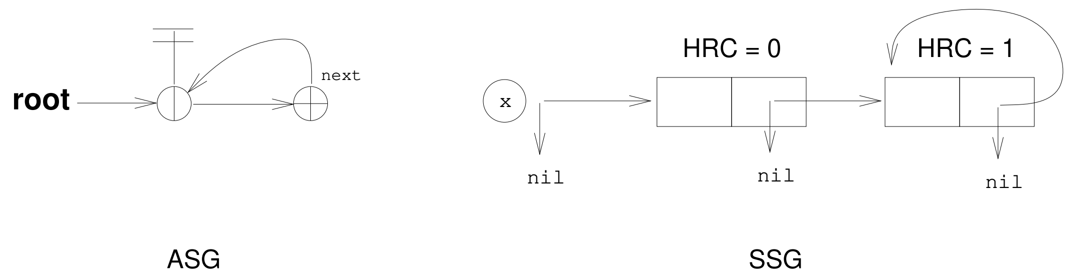

Figure 3: ASG & SSG Singly-Linked List

The general transformation is complicated by the succinctness of the ASG which allows us to represent a singly-linked list with a single summary node. The transformation requires an informal definition of *deconstruction* (formalized later). Deconstruction is a technique which splits a storage node out of a summary node allowing that single node to be dealt with separately.

To transform an arbitrary ASG to an SSG:

1. Deconstruct any storage nodes referenced directly from variables (this mimics the SSG's *deterministic variables*).

2. Set the heap reference count (HRC) by counting the number of incoming D-references of each node which do not originate at a variable and mapping all counts greater than 1 to ∞. Any node with an incoming N-reference has an HRC of ∞.

3. Remove choice nodes, adding labeled references to all the objects reachable through choice nodes, and either D- or N-references to the storage nodes themselves.

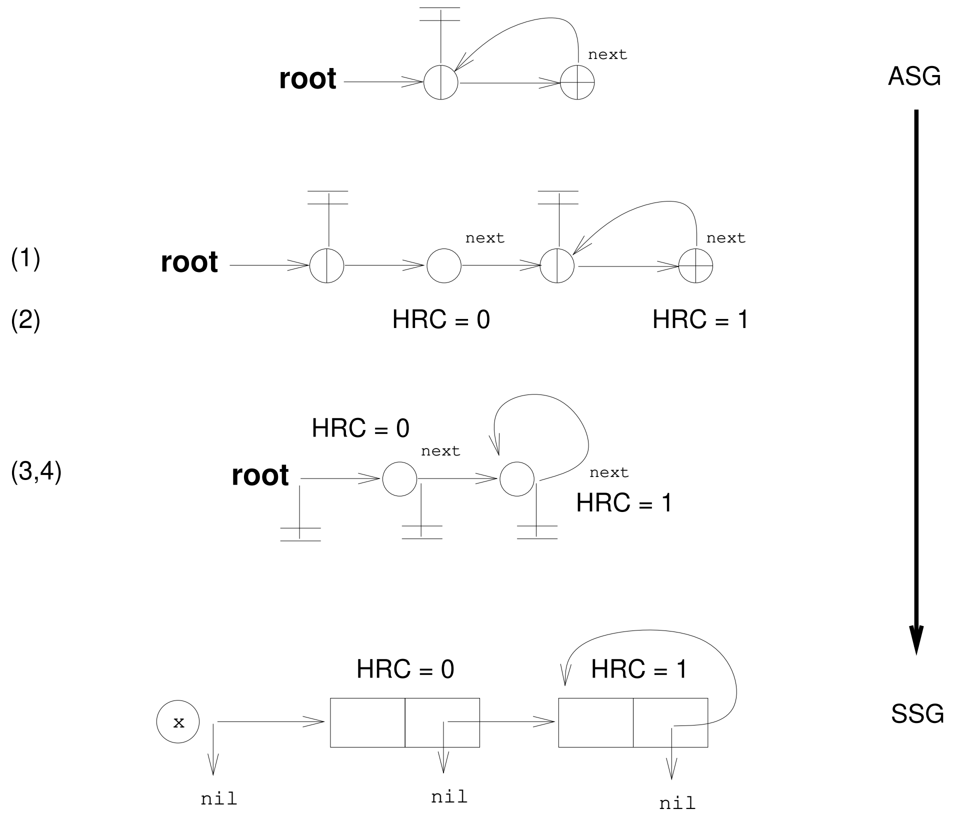

Figure 4: ASG to SSG Translation

4. Change single and summary nodes to simply storage nodes. The graph is now an SSG.
5. In order to fit the SSG algorithm requirements, the graph must be compressed such that there is only one storage node per variable and per storage allocation point.

In Figure 4 we show how the canonical ASG representation of a singly-linked list is transformed into the canonical SSG representation of the same structure. At the top is the original ASG. In the first step, the node which is reached from *root* is separated from the first summary node. Then the Heap Reference Count is calculated. Only the summary node is reached from another storage node, so it has a count of one whereas the single node is only reached by a variable and has a count of zero. In the third and fourth steps, the distinctions between the special node and reference types are forgotten, and the choice nodes are removed. The graph is now identical to the SSG; only the graph symbols are different.

4 Building ASGs

Structure analysis begins with an initial store, built from initial values of program variables (global and static data declarations). The program is then interpreted against this store, statement by statement. For our purposes, the program consists of storage allocation, assignment, conditional and loop statements. Assignments are to and from a variable or a field of an object referenced from a variable. The program is considered to be in Static Single Assignment form [CFR+91].

4.1 Driver

The overall structure of the analysis algorithm follows that of conventional iterative dataflow analysis. The solution at every program point is related to the solution at other points. Program points are program statements and the relations are defined on control flow arcs. The driver procedure appears in Figure 5.

```
COMPUTE-PROCEDURE-ASG(procedure)
  work.Enqueue(procedure.entry)
  while work.Not-Empty() do
    s ← work.Dequeue()
    s.COMPUTE-STATEMENT-ASG()
    if (s.outASG ≠ s.old-outASG)
      work.Enqueue(s.Successors())
    end if
  end while
end
```

Figure 5: Driver Function

Each statement affects the ASG as shown in Figure 6 and described below. Most of the functions in italics have the obvious definitions. The others include:

reachable-along Given an access path $a \rightarrow \mathbf{b}$, the locations labeled **b** in the nodes which are pointed to by a.

reachable-from Given an access path $a \rightarrow \mathbf{b}$, the nodes which are pointed to with label **b** from all nodes pointed to by a.

node Given a location, return the node which contains it.

type For nodes, return *single* or *summary*. For references, return *d-ref* or *n-ref*.

refs Given an access path and a node, returns all the incoming references along the path to the node.

strong-update Given a location to update, a node and its access path, if the node is a single node and all references along the access path to the node are d-references then make a d-reference from the location to the node, otherwise make an n-reference.

4.2 Allocation

When a new node is allocated and assigned to a variable, a new single node of the appropriate type is created, and the variable is set to d-reference that node.

4.3 Assignment

In order to analyze programs which manipulate structures (as opposed to simply building them), we must remove as many references as we create. This follows from the observation that the graph before the manipulation must match the resulting graph. We therefore differentiate between two types of updates — *strong* and *weak*. When one reference replaces another it is called a *strong* update [CWZ90] and corresponds to the killing rules of standard dataflow techniques [ASU87]. Strong updates can only occur when a location is definitely updated. When a location may or may not be changed this results in a *weak* update. Not only is the new reference nondeterministic, but any existing references must be weakened (made nondeterministic) as well.

```
allocation::COMPUTE-STATEMENT-ASG()
  n ← new-single-node(struct-type(rhs))
  outASG.make-ref(d-ref,lhs,n)
end

cfg-diverge::COMPUTE-STATEMENT-ASG()
  outASG-1 ← inASG
  outASG-2 ← inASG
end

cfg-converge::COMPUTE-STATEMENT-ASG()
  s-vars ← inASG-1.vars()∩inASG-2.vars()
  ∀v ∈ s-vars
    c ← new-choice-node()
    outASG.make-ref(ref,v,c)
    outASG.make-ref(inASG-1.ref-type(v),c,
                    inASG-1.dest(v))
    outASG.make-ref(inASG-2.ref-type(v),c,
                    inASG-2.dest(v))
  outASG.COMPRESS()
end
```

```
assignment::COMPUTE-STATEMENT-ASG()
  lvals ← reachable-along(lhs)
  rvals ← reachable-from(rhs)
  ∀v ∈ node(lvals),type(v) = summary,
      type(ref-from-to(lhs,v) = d-ref
    inASG.DECONSTRUCT(lhs,v)
  ∀v ∈ rvals,type(v) = summary,
      type(ref-from-to(rhs,v) = d-ref
    inASG.DECONSTRUCT(rhs,v)
  ∀r ∈ lvals
    c ← new-choice-node()
    if (type(node(r)) ≠ summary and
        ∀v ∈ refs(lhs,node(r)),type(v) = d-ref)
      ∀m ∈ rvals
        outASG.strong-update(c,m,rhs)
    else
      ∀k reachable from node(r) along r
        outASG.make-ref(n-ref,c,k)
      ∀m ∈ rvals
        outASG.make-ref(n-ref,c,m)
    end if
    outASG.make-label-ref(label(r),ref,node(r),c)
end
```

Figure 6: Basic Functions

Manipulating summary nodes directly results in weak updates introducing nondeterminism into the graph. We present a technique called *deconstruction* in Section 5.2 which separates out a single node from a summary node making strong updates possible. With this facility, we discuss the steps to interpret a general assignment of and through a field ($a \rightarrow \mathbf{b} = c \rightarrow \mathbf{d}$).

First, any summary nodes pointed to by a, c and $c \rightarrow \mathbf{d}$ are deconstructed. This step is not necessary to preserve a safe approximation, but it improves precision by enabling strong updates of nodes which have been summarized. Deconstructing the nodes in which the update will occur allows single locations to be updated and deconstructing the right hand side make it possible to assign a pointer to a particular node.

Second, for each node on the left hand side (pointed to by a) we decide if we can make a strong update. If the node is reached from a by only d-references then, if the node exists, a definitely points to it. If that node is also a single node, then we have a single location to update and can do a strong update, otherwise we do a weak update. The pseudocode for assignment appears on the right side of Figure 6.

4.4 Conditionals and Loops

The effect of the conditional can be safely approximated by applying the effects of each branch separately, and then taking a safe merger of the results. This is accomplished by starting each branch of the conditional with the ASG at the conditional.

At any point where control flow converges, for instance at the back-edge of a loop, a merge is taken (see Section 4.5). Loops are automatically handled by the worklist approach where transfer functions for statement successors are evaluated if the incoming ASG changes. However, for the data flow algorithm to terminate the fixed-point of the iteration must be detected.

Detection of the fixed point requires a node labeling which allows nodes in different ASGs to be matched. The nodes are labeled on the basis of their creation points as represented by (*i*) the program statement which caused their creation, and (*ii*) a timestamp based on the flow of ASGs through the program (the order in which program statements are required to be evaluated). When two summary nodes are combined, a total order over creation points determines the label of the resulting node. Nodes match when they either have identical labels, or the same label relative to the current timestamp. The fixed-point is detected if all the edges match for the matched nodes. Since matching is deterministic, the size of the graph is bounded and the references increase monotonically, all nodes will be matched at the fixed point.

4.5 A Safe Merge

At points of control confluence, such as the exits of loops or the point below a conditional, the ASGs from each path must be combined. Merging combines two ASGs, producing a safe approximation to the reference patterns of both. Any number of paths may merge at a program point, but such cases can be handled as a sequence of pairwise merges.

A safe merge is achieved by inserting choice nodes in front of each program variable pointing to the alternatives from each graph. Such a merge is safe because each variable points to everything it pointed to in either graph. The result, however, is large since it includes all the nodes of both graphs. Consequently, while merging ASGs is logically the result of compressing the result of the safe merge above, a more efficient mechanism should be used in practice. In Figure 6, a safe merge occurs in the function `cfg-converge::COMPUTE-STATEMENT-ASG`.

5 Bounding the Size of ASGs

While the preceding description yields an algorithm which is safe, it is not efficient. Since we have presented no mechanism for summarization, the ASG will continue to expand (in most programs) forever. In order to prevent this, the ASG at a program point is compressed to a size proportional to the number of program variables. This is done by combining storage nodes into summary nodes. Unfortunately, summarization can result in the loss of critical information. ASGs provide two additional mechanisms for dealing with this problem:

- **Identity paths** to preserve the internal structure of summary nodes.
- **Deconstruction** to enable the extraction of a single node from a summary node.

The pseudocode for these functions is given in Figure 7 with subfunctions:

ref-along-label Given two nodes and a label, return *d-ref* if all the references from the first node with the label to the second node are d-references, otherwise if a reference exists return *n-ref* otherwise return *no-ref*.

ref-from-to Given a location and a target node, returns the *ref-along-label* of the node of the location, target node, and the label of the location.

ref-along Given a node and a label, return the reference from the node along the label.

locations Given an access path, return all the locations which store the terminal reference.

5.1 Identity Paths

Identity paths are pairs of labels which define circular relationships following d-references between pairs of storage nodes. They preserve the reference pattern internal to summary nodes. This information is used during deconstruction to retrieve the infinite structure of the data structure. If this information were not stored, self references would always imply a potential cycle.

Identity paths are initially created and attached to single nodes where the circular relationship is explicit in the d-references. They are preserved when storage nodes are combined, resulting in a summary node which inherits the intersection of the identity paths *compatible* with both its constituents. An identity path **(a b)** is compatible if it either holds for the node or if the outgoing reference labeled **a** is NIL.

```
DECONSTRUCT(src, n)
  m ← new-single-node(struct-type(n))
  ∀r ∈ in-refs(n)
    make-ref(type(r),src(r),m)
  ∀r ∈ out-refs(n)
    make-ref(type(r),m,dest(r))
  ∀s ∈ locations(src)
    c ← new-choice-node()
    make-ref(ref,s,c)
    ∀x ∈ reachable-from(s), x ≠ n
      make-ref(type(ref-from-to(s,x)),c,x)
    end for
  end for
  ∀k ∈ {n,m}
    ∀p ∈ id-paths(k),p = a.b
      if (∀s ∈ reachable-from-label(k,a),
          k ∉ reachable-from-label(s,b)
        remove-ref(ref-along(k,a))
      end if
end

CHECK-VALIDITY(n,p)
  if ∀m ∈ reachable-from-label(n,p.out-label)
      (type(ref-along-label(n,m,p.out-label))
          = d-ref and
        type(m) = single or p^-1 ∈ id-paths(m)
        and
        type(ref-along-label(m,n,p.in-label))
          = d-ref and
        type(n) = single or p ∈ id-paths(n))
      return TRUE
    else
      return FALSE
end
```

```
UPDATE-IDENTITY-PATHS(r)
  ∀n ∈ back-reachable-from(r)
    ∀p ∈ id-paths(n),∃s ∈ out-refs-to(n,r),
        p.out-label = label(s)
      if (CHECK-VALIDITY(n,p) = FALSE)
        remove-path(n,p)
  ∀n ∈ reachable-from(r)
    ∀p ∈ id-paths(n),∃s ∈ in-refs-from(n,r),
        p.in-label = label(s)
      if (CHECK-VALIDITY(n,p) = FALSE)
        remove-path(n,p)
  ∀n ∈ back-reachable-from(r)
    ∀m ∈ reachable-from(r)
      ∀p ∈ paths(label(out-refs-to(n,r)),
                 label(out-refs(m)))
        if (CHECK-VALIDITY(n,p) = TRUE)
          add-path(n,p)
      ∀p ∈ paths(label(out-refs(n)),
                 label(out-refs-to(m,r)))
        if (CHECK-VALIDITY(n,p) = TRUE)
          add-path(n,p)
end
```

Figure 7: Identity Path and Deconstruction Functions

After adding or deleting a reference, storage nodes may fall into or out of identity relationships. The pseudocode for updating identity paths whenever a reference is added or deleted appears in Figure 7. Identity paths are determined locally by examining the nodes which are either sources or targets of the changed reference. All identity paths which are effected by the change are checked for validity (using function `CHECK-VALIDITY`), and if the nodes are both single nodes, any new valid identity paths are created.

An identity path **(a b)** is valid for a storage node if all nodes reachable along the reference **a** are d-referenced, and the node itself is d-referenced with label **b** starting from these nodes. If the node in question is a summary node, we assume that the current identity paths attached to the summary node are valid. This assumption is safe since the only d-references incident on summary nodes were created between single nodes.

5.2 Deconstruction

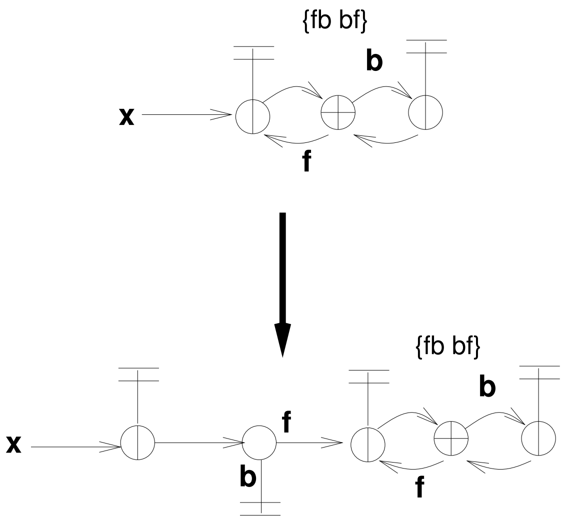

Figure 8: Deconstruction

To preserve as much information as possible, variables used in updates must refer to storage nodes not summary nodes. As necessary, summary nodes are *deconstructed* into a storage node and a summary node, isolating the node to be assigned to or from. An example of deconstruction is shown in Figure 8.

Deconstruction is done on all summary nodes which are d-referenced from a location. A new storage node is created with all the references of the summary node, except those that can be eliminated by exploiting disjunction information or by considering identity paths. In addition, since the new single node is defined to be the one which is reached along the d-reference, other references which refer to the new node along the d-reference are removed. The pseudocode for deconstruction appears in Figure 7.

Consider the graphs in Figure 8. We have eliminated the **f** reference to the new storage node because the reference from x and an **f** reference are disjoint alternatives. Identity paths also constrain the new node's references. In the example, the **b** reference for the new storage node must be NIL since the identity path would require it to point to the source of an **f** reference which does not exist. Lastly, the NIL reference from the **b** field of the summary node is also removed since some incoming d-reference must have been generated only by an **f** field reference. Figure 11, Step 4 shows how the deconstructed storage nodes form a doubly-linked list as required by the identity path.

```
COMPRESS()
  pair-nodes-by-combine-criteria()
  while (|ASG.nodes()| > c*|ASG.vars()|)
    REMOVE-REDUNDANT-NODES()
    COMBINE-DUPLICATE-EDGES()
    n-pair ← remove-first-pair()
    COMBINE-NODES(n-pair.1,n-pair.2)
  end while
end

REMOVE-REDUNDANT-NODES()
  compute-connectivity()
  ∀x,y ∈ ASG.nodes()
    if (type(x) = type(y) = choice and
        connects(x) = connects(y))
      fuse(x,y)
    end if
    if (type(x) = type(y) = storage and
        struct-type(x) = struct-type(y) and
        connects(x)/y = connects(y)/x)
      fuse(x,y)
    end if
end
```

```
COMBINE-DUPLICATE-EDGES()
  ∀x ∈ ASG.nodes(),type(x) = choice
    if (∃u,v ∈ x.edges(),dest(u) = dest(v))
      make-ref(n-ref,x,dest(u))
      remove-ref(u)
      remove-ref(v)
    end if
end

COMBINE-NODES(x,y)
  n ← new-node(comb-type(x,y),struct-type(x))
  ∀r ∈ in-refs(x),∃s ∈ in-refs(y),
      type(r) = type(s) = d-ref and
      src(r) = src(s)
    make-ref(d-ref,src(s),n)
  ∀r ∈ in-refs(x),choose unadded s ∈ in-refs(y)
    c ← new-choice-node()
    make-ref(ref,src(s),c)
    make-ref(ref,src(r),c)
    make-ref(d-ref,c,n)
  ∀r ∈ in-refs(x) ∪ in-refs(y),
      r has not been added yet
    make-ref(n-ref,src(r),n)
  ∀r ∈ out-refs(x)
    c ← new-choice-node()
    make-label-ref(label(r),ref,src(r),c)
    make-ref(type(r),c,dest(r))
    s ← ref-along(y,label(r))
    make-ref(type(s),c,dest(s))
  id-paths(n) = valid-path-union(x,y)
end
```

Figure 9: Compression Functions

5.3 Compression

Approximating the store in finite space and detecting termination requires an upper bound on the size of the ASG along with a deterministic compression function. The compression function is composed of the following subroutines which are each discussed in turn in the following sections:

1. Remove redundant choice nodes, references and storage nodes.
2. Select storage nodes to be combined.
3. Combine storage nodes.

The pseudocode for the compress function appears in Figure 9 with subfunctions:

pair-nodes-by-combine-criteria This is defined as part of the **Combine** operation later in this section.

compute-connectivity Compute for each choice node, the set of incoming references which are reachable from the choice node. This is a transitive closure calculation.

connects Given a choice node, the set of incoming references reachable from that choice node.

in-refs Given a node, all the incoming references.

out-refs Given a node, all the outgoing references.

valid-path-union Given two nodes, all the identity paths attached to either of them that are valid for both of them.

fuse Given two nodes with identical connectivity and identity paths, combine them into a summary node.

comb-type Given two nodes, returns *single* if the two nodes can be combined into a single node, otherwise returns *summary*.

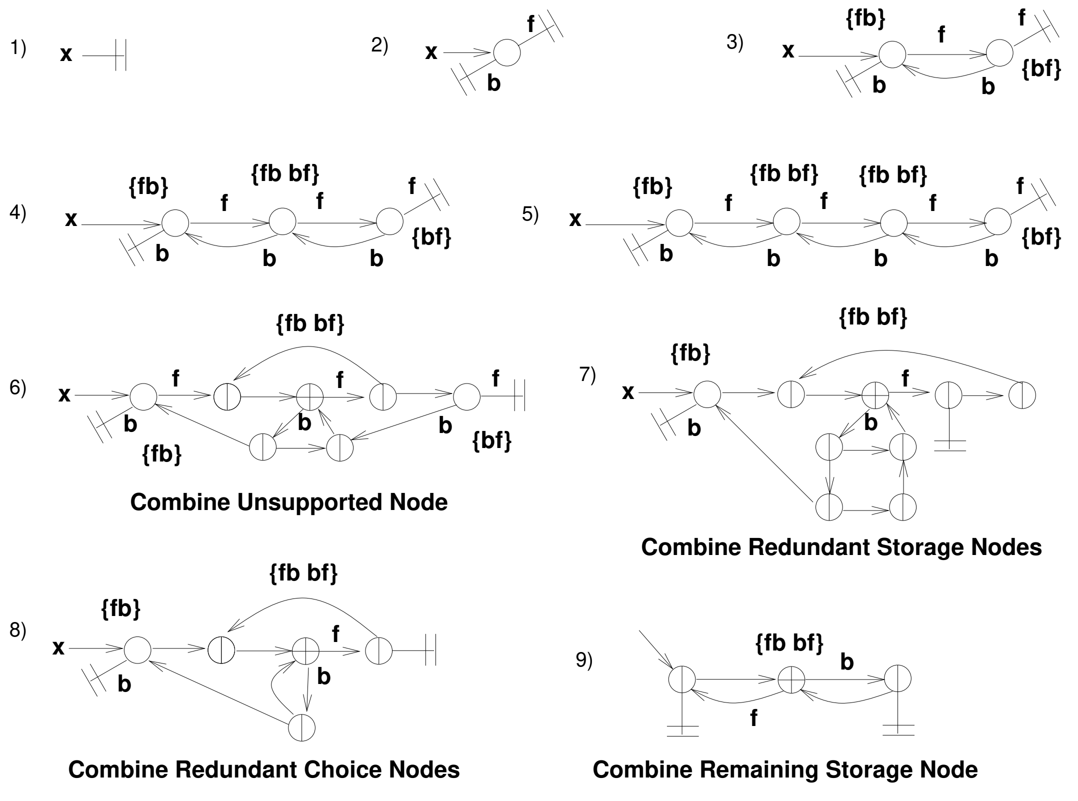

Figure 10: Doubly-linked List Loop ASG

5.3.1 Redundancies

Redundancies in the ASG arise from merge points. An edge is redundant if removing it does not change the alias pattern between variables. Any two edges from a choice node to a storage node (one of which must be an n-reference by definition) can be combined into a single n-reference since the reference pattern through the choice node to the storage node is already nondeterministic.

Choice nodes are redundant when merging them would not change the ASG's connectivity. Connectivity must be determined independently for the d- and n-references emanating from a group of choice nodes. Any pair of choice nodes which reach the same outputs may be merged without changing the connectivity. Connectivity is determined by a transitive closure algorithm followed by pairwise comparison of directly connected nodes.

A storage node is redundant if combining it with a summary node does not change the connectivity of the summarized nodes (deconstruction would result in recovery of the original graph).

For example, in Figure 10, Step 6, the storage node to the far right is redundant, and combining it with the summary node in the center results in the addition of a reference to NIL to the choice node referenced by the **f** field in Step 8. In Step 7, a set of choice nodes are all connected to the summary node and the storage node pointed to by x, so they are merged. In Step 9, the last storage node with a compatible identity path is combined because no variable directly references the storage node. The resulting graph matches that for the doubly-linked list in Figure 2.

5.3.2 Combine Criteria

The combine criteria determines which nodes to combine such that the size of the ASG is within a constant factor of the number of program variables and as much deterministic reference information is preserved as possible. The following heuristics are used to identify nodes to be combined and are used to implement the subfunction *pair-nodes-by-combine-criteria*.

1. *Similar Nodes*: two nodes which were the same node before a control flow divergence and are now in the graph being compressed.
2. *Directly-referenced Nodes*: two nodes which are d-referenced from the same variable.
3. *Distant Nodes*: nodes which are "farthest" from variables using the number of intermediate storage nodes as a measure of distance.

The first heuristic pairs up natural combinations, such as nodes which have not changed. This can be accomplished logically as follows: starting from the set of variables which are not listed in the ϕ-functions at the merge point [CWZ90] (those that were not assigned in the region of the program for which the merge point is on the post-dominator front), select all nodes referenced only by these variables whose outgoing references have not changed. Applying this criteria recursively will result in the combining of untouched portions of the graph. In practice, storing the graph as edge difference lists attached to variables and nodes allows the untouched portion of the graph to be determined quickly.

The second heuristic chooses to pair nodes which have similar reference patterns. The rationale for this is that fewer n-references are created as a result of combining two such nodes.

The third heuristic chooses nodes which are distant from variables since updates occur in the vicinity of variables. Such nodes are less likely to contain information which will be needed by the algorithm in succeeding steps; consequently, these nodes can be summarized. One way of identifying such nodes is by distributing a unit weight among nodes within a neighborhood of all variables, such that distant storage nodes get a lower weight.

Nodes selected through the latter two heuristics are matched with a node of the same type selected through a limited breath first search since neighboring nodes are likely to share reference patterns. Should the search fail, the node will be combined with a random node of the same type, but it will be combined after all those for whom the search succeeded since, as was noted in [CWZ90], combining two unrelated nodes will not decrease space usage.

It is important to note that the combine criteria must be deterministic. A total order over the nodes can be used to ensure this.

5.3.3 Combine

Given two nodes selected according to the combine criteria, a decision needs to be made as to whether the combined node will be a single node or a summary node. Any two single nodes which cannot simultaneously exist can be combined into a single node. This condition can be determined for unchanged portions of the graph at merge points by using difference lists. It can also be determined locally by checking if the two nodes are both d-referenced from a single location (either a labeled reference from a single node or a variable). All other pairs of nodes are combined into a summary node. The pseudocode for the combine function (`COMBINE-NODES`) is shown in Figure 9. The combine operation results in a new node of the appropriate type with incoming and outgoing references as described below.

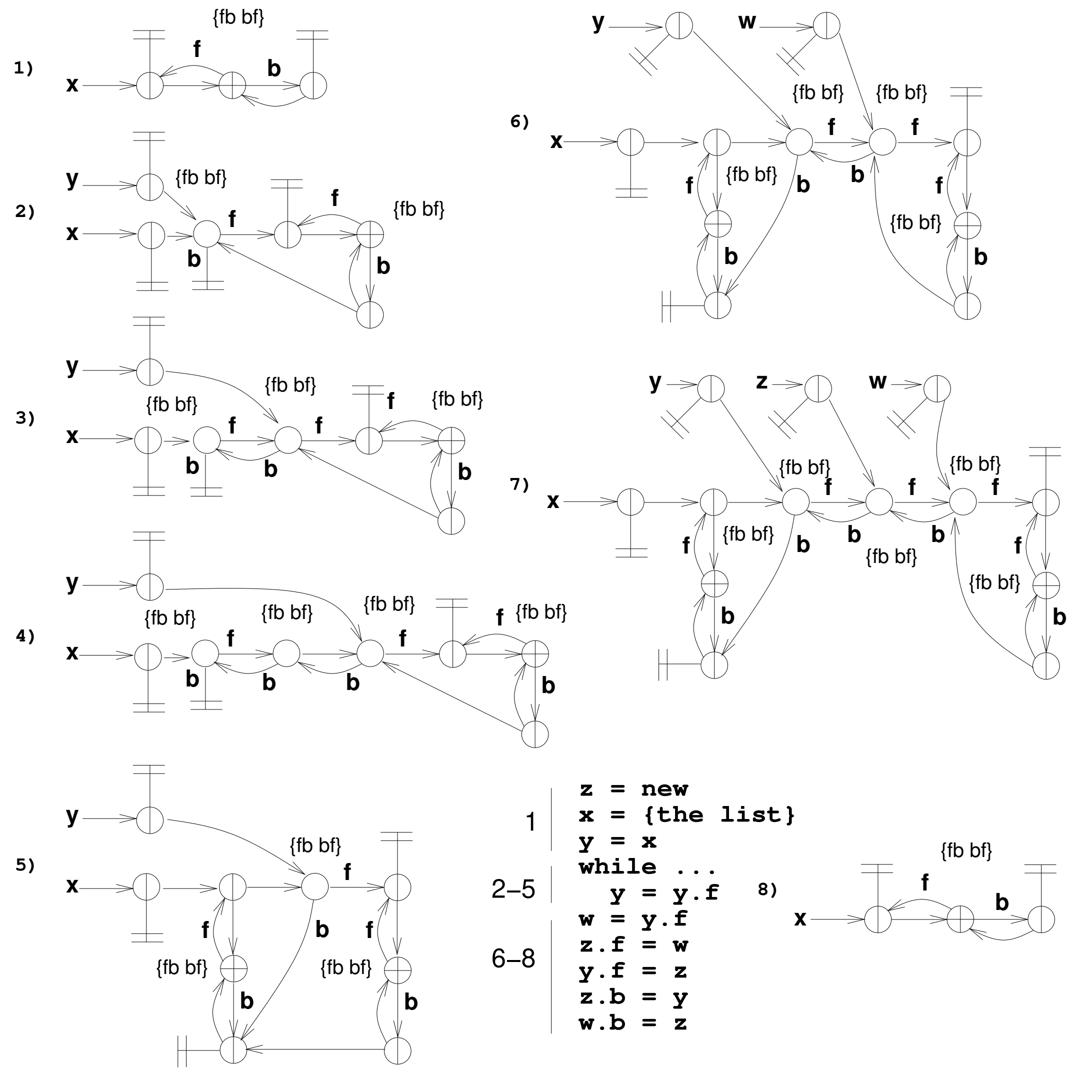

Figure 11: Insertion in a Doubly-linked List

To make strong updates to the ASG, we need to preserve incoming d-references, so we apply heuristics designed to preserve as many d-reference alternatives as possible. For each d-reference, if it is possible to find a corresponding d-reference incident on the other node with identical source, the two d-references can be combined into one. If it is not possible to find a corresponding d-reference with identical source, the d-reference is paired heuristically with a d-reference incident on the other node giving priority to references originating at variables, and to references sharing the same field name. For each pair of d-references a choice node is created to join the d-references from the two sources into one d-reference. All other incoming references become n-references. For outgoing references choice nodes are simply inserted which point to the two alternatives for the different nodes. These heuristics attempt to prevent the introduction of n-references since their introduction results in a loss of deterministic reference information.

In Figure 10, Steps 5 and 6 show the combination of two nodes in a doubly-linked list. First, the two central nodes in Step 5 are selected for combining since they are (i) not pointed to directly by a variable and (ii) have the same number of incoming d-references. Then new choice nodes are inserted with edges attached according to the rules above (see Step 6).

6 Safety

The ASG's approximation of the structure of the program store is contingent on the safety of the initial approximation and the safety of the functions which manipulate it. The abstract interpretation of program statements and the safe merge have been shown to be safe in their respective sections.

The identity path operations are safe since (i) identity paths are only created when the identity relation is explicit in the graph and (ii) combining of nodes results in the combined node getting identity paths which are compatible with both the constituent nodes.

Compress is safe since the resulting nodes have at least the connectivity of the constituent nodes and at least as many potentially incoming edges from any source. It follows that the resulting graph has at least the connectivity of the source graph.

Deconstruction is safe since the newly created node has the same connectivity as the parent summary node which is logically an infinite set of single nodes of like connectivity. The references which are removed are the result of the definition of the deconstructed node as the node which is reached along the incoming d-reference. Since such a node must exist, and since the identity path operations are safe, removing these edges is safe as well.

7 Complexity Analysis

The analysis below assumes that the maximum number of nodes in the ASG is bounded from above by a constant times the number of distinct program objects (variables, etc.) (V). Note that this condition can be easily ensured in any implementation of the algorithm.

Since at the end of the intraprocedural algorithm, each program statement stores $O(1)$ copies of the ASG, the worst-case space complexity of the intraprocedural algorithm is $O(S * V^2)$, where S is the number of program statements.

The time complexity is obtained by noting that the dominant cost-contributors in the construction (update) of the ASG for any program statement are the identity-path update, summary-node deconstruction, and the ASG compression steps. Each call to `UPDATE-IDENTITY-PATHS` involves $O(V)$ invocations of `CHECK-VALIDITY`, each of which requires a transitive-closure computation on the ASG, and results in an overall complexity of $O(V^4)$. `DECONSTRUCT` also requires a transitive-closure computation and has complexity $O(V^3)$. The four constituent operations of `COMPRESS` each have complexity $O(V^3)$: the dominant operation in `REMOVE-REDUNDANT-NODES` is *compute-connectivity* which involves a transitive-closure operation, while in case of `COMBINE-NODES` it is *valid-path-union* which requires $O(1)$ calls to `CHECK-VALIDITY`. The overall complexity of `COMPRESS` is $O(V^4)$ because the number of iterations before the ASG gets compressed to the required size is $O(V)$ since in each iteration the number of storage nodes are being reduced by at least one, and the number of choice nodes in the graph is at most a constant factor of storage-node pairs. Thus, the worst-case time complexity of modifying the ASG due to a program statement is $O(V^4)$.

To bound the number of iterations required for the algorithm to reach fixed point, we consider a round-robin evaluation of the program statements. This is more inefficient than the worklist approach but does not change the asymptotic complexity. Since the longest cycle in a control-flow graph for a procedure with S statements can have length $O(S)$, propagating structure information to all statements of the procedure requires $O(S)$ iterations. This corresponds to $O(S^2)$ statement evaluations; consequently, the overall complexity of the iterative algorithm is $O(S^2V^4)$. This is a pessimistic measure of the algorithm complexity; in practice, we expect the asymptotic complexity to be considerably less since the improved precision of the algorithm should result in sparse connectivity.

8 Current Status

We are implementing this algorithm as part of the *Concert* compiler. The resulting information will be used to enable transformations of runtime data structures such as fusion, clustering and tiling as well as a host of other traditional and novel optimizations. Structure analysis is so critical to attaining efficient use of distributed memory parallel machines that we believe the cost of such analysis is justified. Uninformed compilation of such languages can result in programs which are several orders of magnitude slower than corresponding fast serial implementations.

The *Concert* system includes a compiler for an extended version of Concurrent Aggregates [CD90] and a runtime which exposes the cost hierarchy among the basic operations required for fine grained reactive computation [KC93]. Programs compiled with the system execute on uniprocessor workstations and the CM5 [Thi91].

9 Summary and Future Work

Efficient execution of irregular computations containing dynamic structures on parallel architectures requires that the compiler know the shape of the runtime store. The ease of general purpose parallel programming depends on future compilers being able to determine this information without programmer assistance. We present an algorithm for determining structure information.

We have described a new structure graph, the ASG, which can model multiply-linked infinite structures in a finite representation without loss of information. We have also described an algorithm for computing the ASG which can analyze programs with such structures, even in the presence of mutation. The key to this analysis is to preserve deterministic reference information. This is accomplished through the novel features of the ASG — deterministic references, choice nodes and identity paths, and the novel features of the algorithm including deconstruction of summarized nodes.

Successful structure analysis results when the functions which manipulate the real data structures can be be shown to preserve the ASG which models the data structures. The key issues are:

1. The ability to precisely model recursive structures in finite space.

2. The ability to *deconstruct* the finite model revealing part of the infinite structure.

3. The ability to make strong updates to the revealed portion.

While we are pursuing a much needed implementation and empirical verification of the algorithm, manual application on small examples has encouraged us to believe that our algorithm meets these criteria. In Figure 11, we show the analysis of a code segment which inserts a node into a doubly linked list. A variable y traverses the list starting in step 2. Values of the graph within the loop are given in steps 3 and 4 with the fixed point being reached in 5. In steps 6 and 7, strong updates are done to the graph to insert the new node. The temporary variables fall out of scope, and the graph is compressed in step 8, resulting in the same graph that we started with.

In the future, we plan to enhance the algorithm for incremental flow-sensitive analysis by storing information about distinct call paths and their entry ASGs. If the ASGs entering a procedure are sufficiently distinct, the path through the procedure can be split incrementally resulting in flow-sensitive analysis. We are considering *path nodes* as a means to reduce the cost of a splitting operation [CU91].

References

[ASU87] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Computer Science. Addison-Wesley, Reading, Massachusetts, 1987.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensative interprocedural computation of pointer-induced aliases and side effects. In *Twentieth Symposium on Principles of Programming Languages*, pages 232–245. ACM SIGPLAN, 1993.

[CD90] A. A. Chien and W. J. Dally. Concurrent Aggregates (CA). In *Proceedings of Second Symposium on Principles and Practice of Parallel Programming.* ACM, March 1990.

[CFKP92] A. A. Chien, W. Feng, V. Karamcheti, and J. Plevyak. Techniques for efficient execution of fine-grained concurrent programs. In *Proceedings of the Fifth Workshop on Compilers and Languages for Parallel Computing*, pages 103–13, New Haven, Connecticut, 1992. YALEU/DCS/RR-915, Springer-Verlag Lecture Notes in Computer Science, 1993.

[CFR+91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CKP93] Andrew Chien, Vijay Karamcheti, and John Plevyak. The concert system – compiler and runtime support for efficient fine-grained concurrent object-oriented programs. DCS Technical Report UIUCDCS-R-93-1815, University of Illinois, Department of Computer Science, 1304 W. Springfield Avenue, Urbana, Illinois, June 1993.

[CU91] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *OOPSLA '91 Conference Proceedings*, 1991.

[CWZ90] D. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[HN90] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[HNH92] L. Hendren, A. Nicolau, and J. Hummel. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–260. ACM SIGPLAN, ACM Press, June 1992.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 28–40. ACM SIGPLAN, ACM Press, 1989.

[JM81] N. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice-Hall, 1981.

[KC93] Vijay Karamcheti and Andrew Chien. Concert – efficient runtime support for concurrent object-oriented programming languages on stock hardware. To appear in the Proceedings of SUPERCOMPUTING'93, 1993.

[Lar89] James Richard Larus. Restructuring symbolic programs for concurrent execution on multiprocessors. Technical Report UCB/CSD 89/502, University of California at Berkeley, 1989.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 21–33. ACM, June 1988.

[Mye81] E. Myers. A precise interprocedural data flow algorithm. In *Eighth Symposium on Principles of Programming Languages*, pages 219–30, 1981.

[Thi91] Thinking Machines Corporation, Cambridge, Massachusets. *CM5 Technical Summary*, October 1991.