UC Irvine
ICS Technical Reports

Title
Abstractions for recursive pointer data structures : improving the analysis and transformation of imperative programs

Permalink
https://escholarship.org/uc/item/1z03c1sj

Authors
Hendren, Laurie J.
Hummel, Joseph
Nicolau, Alexandru

Publication Date
1992-01-29

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

ARCHIVES
Z
699
C3
no. 92-09

Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs

TECHNICAL REPORT 92-09

Laurie J. Hendren*,
Joseph Hummel†, and
Alexandru Nicolau‡

January 29, 1992

To appear in *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation.*

*McGill University, School of Computer Science. This work supported in part by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.

†UC-Irvine, Dept. of ICS, Irvine, CA 92717-3425.

‡UC-Irvine, Dept. of ICS. This work supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215.

Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs

Laurie J. Hendren *
School of Computer Science
McGill University

Joseph Hummel
Alexandru Nicolau †
Dept. of Information and Computer Science
UC-Irvine

Correspondence to:
Joseph Hummel
Dept. of ICS
UC-Irvine
Irvine, CA 92717
Phone: (714) 725-2248
E-mail: jhummel@ics.uci.edu

Abstract

Even though impressive progress has been made in the area of optimizing and parallelizing scientific programs with arrays, the application of similar techniques to programs with general (cyclic) pointer data structures has remained difficult. In this paper we introduce a new approach that leads to improved analysis and transformation of programs with recursively-defined pointer data structures.

Our approach is based on providing a mechanism for the abstract description of data structures (ADDS), which makes explicit the important properties, such a dimensionality, of pointer data structures. As illustrated with numerous examples, the ADDS definitions are both natural to specify, and general enough to handle both simple and complex pointer data structures.

We illustrate how an abstract data structure description can improve analysis by presenting an alias analysis approach that combines an alias analysis technique, *path matrix* analysis, with information available from an ADDS declaration. Given this improved alias analysis technique, we provide a concrete example of applying a software pipelining transformation to loops involving pointer data structures.

1 Introduction and Motivation

One of the key problems facing optimizing compilers for imperative programming languages is *alias resolution*, that is, detecting when two distinct memory accesses may refer to the same physical memory location. The effectiveness of many compiler analysis techniques and code-improving transformations rely upon accurate alias resolution. Given the current trend towards vector, RISC, and superscalar architectures, where optimizing compilers are even more important for efficient execution, the need for more accurate analysis and alias resolution will grow.

Scientific codes have often been the target of optimizing and parallelizing compilers. Such codes typically use arrays for storing data, and loops with regular indexing properties to manipulate these arrays. A good deal of work has been done in the area of analysis and transformation in the presence of arrays and loops and as a result numerous techniques, such as invariant code motion, induction variable elimination, loop unrolling, vectorizing, prefetching, and various instruction scheduling strategies such as software pipelining and delay slot filling, have been developed [DH79, RG82, PW86, ASU87, ZC90, Kri90]. Unfortunately, codes that utilize dynamically-allocated pointer data structures are much more difficult to analyze, and therefore there has not been as much

*This work supported in part by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.
†This work supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215.

progress in this area. This is problematic, since numerous data structures in imperative programs—linked-lists and trees for example—are typically built using recursively-defined pointer data structures. Furthermore, such data structures are used not only in symbolic processing, but also in some classes of scientific codes (e.g. computational geometry [Sam90] and so-called *tree-codes* [App85, BH86]).

The goal of this paper is to: (1) investigate why the analysis of pointer data structures is more difficult than the analysis of array references, (2) suggest a new mechanism for describing pointer data structures, and (3) show how better descriptions of such data structures lead to improved analysis and transformation of imperative programs. To motivate the problem, let us first consider the following two code fragments[1].

```
for i = 1 to n
  a[i] = a[i] + b[j];
```

```
while p <> NULL
 { p->data = p->data + q->data;
   p = p->next;
 }
```

The left code fragment adds the element `b[j]` to each element of array `a`. Since arrays have the property that `a[i]` and `a[j]` refer to different locations if $i \neq j$, the compiler can immediately determine from this code fragment that each iteration references a different location of `a`. Further, arrays are usually statically allocated, and it is straightforward to determine that the arrays `a` and `b` are different objects, and thus any reference to `b[j]` will never refer to the same location as `a[i]`. These two observations allow the compiler to perform a number of optimizations, including (1) loading `b[j]` into a register once before the loop begins, and (2) transforming the loop by perhaps unrolling or applying software pipelining.

Now let us consider the other code fragment which operates on a linked-list. This program traverses a list `p`, and at each step adds the value `q->data` to the node currently being visited. In this case, the properties of the structure are not obvious from the code fragment. For example, unless we can guarantee that the list is acyclic, we cannot safely determine that each iteration of the loop refers to a different node `p`. This makes the application of loop transformations difficult. Secondly, since the nodes are dynamically allocated, it is much more difficult to determine if `p` and `q` ever refer to the same node. Thus, the compiler will be unable to detect if `q->data` is loop invariant.

There are currently two options for dealing with these problems: (1) assume the worst case, that the list is possibly cyclic (`p->data` of one iteration may share the same location with `p->data` of another iteration) and that at some iteration `p->data` may refer to the same location as `q->data`, or (2) analyze other parts of the program in an attempt to automatically discover the underlying structure of the list `p`, and to determine the relationship between `p` and `q`.

Although approach (1) is far more common, there has been significant work on approach (2). One class of solutions has been the development of advanced alias analysis techniques (also called structure estimation techniques), that attempt to statically approximate dynamically-allocated data structures with some abstraction. The most commonly used abstraction has been *k-limited* graphs [JM81], and variations on *k-limited graphs* [LH88a, LH88b, HPR89, CWZ90]. The major disadvantage of these techniques is that the approximation introduces cycles in the abstraction, and thus list-like or tree-like data structures cannot be distinguished from data structures that truly contain cycles. The work by Chase et al. [CWZ90] has addressed this problem to some degree; however, their method fails in the presence of general recursion. This is a serious drawback since many programs heavily utilize recursion. Another method, *path matrix* analysis, was designed to specifically deal with distinguishing tree-like data structures from DAG-like and graph-like structures [HN90, Hen90]. This analysis uses the special properties exhibited by tree-like structures to provide a more accurate analysis of list-like and tree-like structures even in the presence of recursion. However, it has the disadvantage that it cannot handle cyclic structures. Other approaches include one based on more traditional dependence analysis [Gua88] (this method assumes the structures do not have cycles), and abstract interpretation techniques (designed for list-like structures commonly used in Scheme programs) [Har89].

Although these methods have made some progress, there remain serious problems to address, particularly in the context of general-purpose imperative programming languages. One of the most serious problems is that such "imperative" pointer data structures often have a complex structure. For example, consider the above pointer loop once again. The linked-list formed by the `next` fields may be part of a larger, more complicated structure.

[1]Treat a, b, p, and q as local variables, where a and b are arrays of size n and p and q are pointers.

It could be, for example, a list linking together the leaves of a tree, the forward pointers of a doubly-linked list, or many other possibilities, all of which confound existing analysis techniques.

2 Our Approach

It is our belief that we can provide a more effective option by carefully studying commonly used pointer data structures, and exploiting the important regular properties exhibited by these structures. The problem then is how to capture the "important" properties of a wide range of recursive pointer data structures. Current imperative programming languages provide no mechanisms for expressing this kind of information.

Based on our successful experience of developing alias analysis techniques for tree-like structures (i.e. a subset of structures that have nice properties), and the failure of other techniques to find accurate information for general structures, we believe that a lack of appropriate data structure descriptions is the most serious impediment to the further improvement of analysis techniques. Thus, our first task was to develop mechanisms that can be used to describe a wide variety of pointer data structures commonly used in imperative programs. These mechanisms need to: (1) be simple to use for the programmer, (2) handle a wide variety of complex pointer data structures, and (3) provide information that can be effectively utilized by the compiler. The properties captured by such data structure descriptions can then be used to improve the effectiveness of existing analysis and transformation techniques, and provide a basis for the development of pointer-specific optimization techniques.

Asking the programmer to specify some properties of his or her data structures should not be considered a radical change in our way of thinking about programming in imperative programming languages. In the domain of scientific programs we do not find it strange to provide programmers with both one-dimensional and two-dimensional array data types (even though these are both implemented as one-dimensional arrays in memory). In addition, in the pointer data structures domain, programmers often convey quite a bit of implicit information about their data structures. For example, consider the following two recursive type declarations:

```
type BinTree                          type TwoWayList
  { int      data;                      { int        data;
    BinTree *left;                        TwoWayList *next;
    BinTree *right;                       TwoWayList *prev;
  };                                    };
```

Even though these type declarations appear identical to the compiler, the naming conventions imply very different structures to readers of a program. In addition, each structure has some very nice properties which the compiler could exploit. For example, a binary tree naturally subdivides into two disjoint subtrees, and a doubly-linked list has the property that a traversal in the forward direction using only the `next` field always visits a new node (likewise for traversals using only the `prev` field). The idea is simply to make this implicit information explicit to the compiler. Positive side-effects may be increased human understanding of programs, and the compiler's ability to generate run-time checks.

Note that Fortran 90 has taken a similar stance in its treatment of pointers to variables. Variables accessible through pointers must be explicitly declared as either pointers or targets [MR90]. This simple declaration greatly improves the accuracy of alias resolution in the presence of pointers.

However, the presence of such a description mechanism alone is not enough. Side-effects in imperative programs often rearrange the components of a data structure, causing a temporary but intentional invalidation of the properties we wish to exploit. Application of optimizing transformations during such a time would be incorrect, and intolerable. Hence some form of data structure analysis is still necessary, not only to ensure correctness, but to enhance debugging as well.

In the remainder of this paper we present our viewpoint that:

1. recursive pointer data structures, regardless of their overall complexity, typically contain substructures that exhibit regular properties that can be exploited for the purpose of compiler analysis and transformation, and

2. the ability to express these properties explicitly is an important first step towards the long-term goal of efficient compiler analysis and alias resolution in the presence of general (cyclic) pointer data structures.

The organization of this paper is as follows. In sections 3 and 4 we address the problem of capturing the regular properties of recursive pointer data structures. In section 3, we present a new technique for the abstract description of data structures (ADDS). We include in section 3 a wide range of examples using our technique, and we outline how the technique is applied to both simple and complex pointer data structures. In section 4 we define the properties of ADDS data structures more formally. In section 5 we provide concrete examples of how the information provided by ADDS can be used to improve the analysis and transformations of programs. Finally, in section 6 we present our conclusions.

3 ADDS - Abstract Description of Data Structures

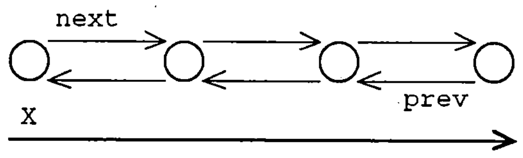
The optimization of codes involving data structures requires knowledge about the properties exhibited by that structure, e.g. shape, size, and method of element access. With arrays, these properties are readily identifiable. The shape of an array is fixed and declared at compile-time, its size is known at the time of creation (and often at compile-time), and locations are referenced using integer indices. The latter property is especially useful in optimizing loops. For example, given a statement S of the form `i = i + c` (`c` $\neq 0$), one is guaranteed that `a[i]` refers to different elements of `a` before and after the execution of S. Contrast this with user-defined pointer data structures, in which none of these properties are made explicit. Given a simple traversal statement T such as `p = p->next`, the compiler cannot readily determine whether `p` denotes a different node after the execution of T, or the same node.

Our goal in this section is to present a formalism for expressing the shape of a recursive pointer data structure, a formalism we shall call ADDS (abstract description of data structures). In developing this formalism we considered a wide variety of pointer data structures currently used in imperative programs, along with our previous experience in developing alias analysis techniques. We first give a high-level summary of how one develops an abstract description, and then we give illustrative examples that include both simple and complex data structures.

3.1 Dimensions and Directions in Pointer Data Structures

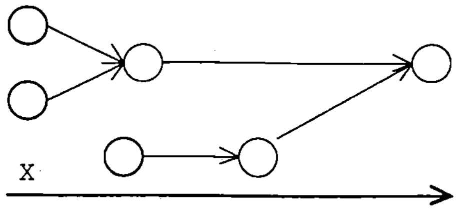
Suppose you have a recursive pointer data structure and you need to describe its shape. Consider the structure in its general form, and select a node as the "origin"—it doesn't matter which node you choose, though some choices make more sense than others (e.g. the root of a tree versus a leaf). Next, think of your structure as having "dimensions," different paths emanating from the origin, with typically one dimension per path. Finally, select a node n other than the origin, and for each recursive pointer field f in n, decide which dimension f traverses and in which "direction." The "forward" direction implies traversing f moves one unit away from the origin, and "backward" implies traversing f moves one unit back towards the origin. A field is limited to traversing one dimension in only one direction[2].

By default, a structure has one dimension D, where it is assumed that all recursive pointer fields traverse D in an "unknown" direction. The idea is to override this default and provide more specific information. For example, we can specify that a singly linked-list has one dimension **X**, and one recursive pointer field **next** that traverses **X** in the forward direction. As illustrated below, a two-way linked-list is best described as having a single dimension **X**, with **next** traversing forward along **X** and **prev** traversing backward along **X**:



To differentiate a list from a more general DAG-like list, e.g.

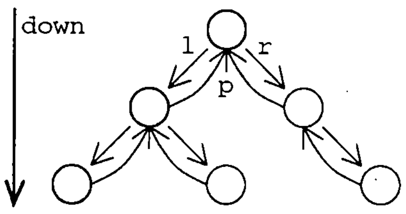
[2]This restriction can be overcome by the programmer without too much difficulty.



we introduce the notion of a field f traversing "uniquely forward," which implies that for any node n, at most one node n' ($n' \neq n$) points to n using f. Syntactically, the declaration of a two-way linked-list would then look like:

```
type TwoWayList [X]
  { int          data;
    TwoWayList *next is uniquely forward along X;
    TwoWayList *prev is backward along X;
  };
```

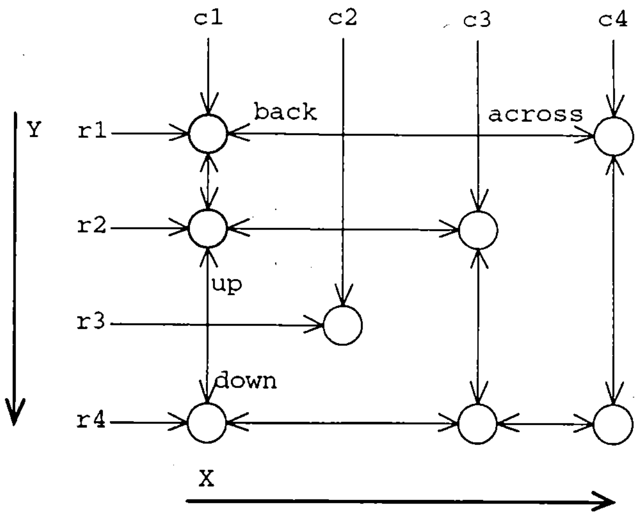
A binary tree can be thought of as having two dimensions `left` and `right`, but for reasons soon to be apparent, we shall consider a binary tree as having only one dimension, `down`. The important property of a binary tree (and of trees in general) is that for any node n, all subtrees of n are disjoint. This information can be expressed by saying that the `left` and `right` fields "combine" to traverse `down` in a uniquely forward manner. More formally, `left` and `right` exhibit the property that at most one `left` *or* one `right` points to any node n, but not both. The motivation for choosing one dimension to describe a binary tree is to support the notion of parent pointers, a field that refers from either a left or right child back to its parent. Thus, pictorially we view such a binary tree as:



Its ADDS declaration would be:

```
type PBinTree [down]
  { int        data;
    PBinTree *left, *right is uniquely forward along down;
    PBinTree *parent       is backward along down;
  };
```

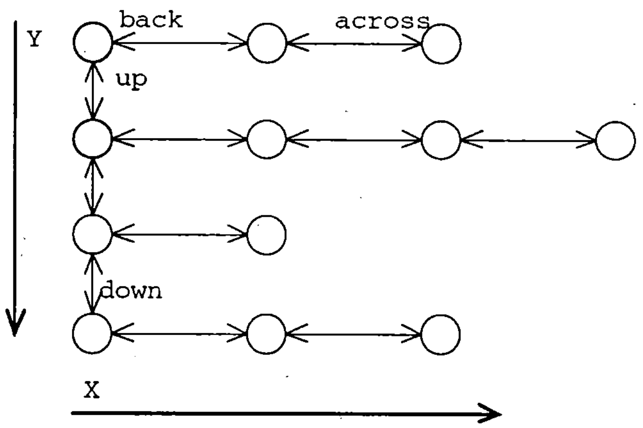
The flexibility of the ADDS technique is illustrated by more exotic recursive pointer data structures. Typically such structures exhibit multiple dimensions, where dimensions are either "independent" (disjoint) or "dependent." For example, an *orthogonal list* [Sta80], used to implement sparse matrices, has two dependent dimensions `X` and `Y` (much like the two-dimensional array it represents):



We say X and Y are dependent since one traversal along X and another traversal along Y may lead to a common substructure. For example, traversing along X from r4 and along Y from c3 may lead to the same node. One can also think of this property at the node level. Each node n may be accessed by traversing from some node along the X dimension, **and** from a different node along the Y dimension. This indicates that there is a dependency between the two dimensions. However, even though the dimensions are dependent, notice that *orthogonal lists* still possess regular properties. For example, traversing forward along X, or forward along Y, is always guaranteed to visit a new node. Further, each row is disjoint, so that parallel traversals of different rows along X will never visit the same node (likewise for columns and the Y dimension). These properties are captured by the following ADDS declaration, which specifies that the fields **across** and **down** are uniquely forward[3]:

```
type OrthList [X][Y]
  { int        data;
    OrthList  *across is uniquely forward along X;
    OrthList  *back   is backward along X;
    OrthList  *down   is uniquely forward along Y;
    OrthList  *up     is backward along Y;
  };
```

A data structure that has two independent dimensions is a list of lists. Consider the following which illustrates a list of lists, including back pointers along each dimension:

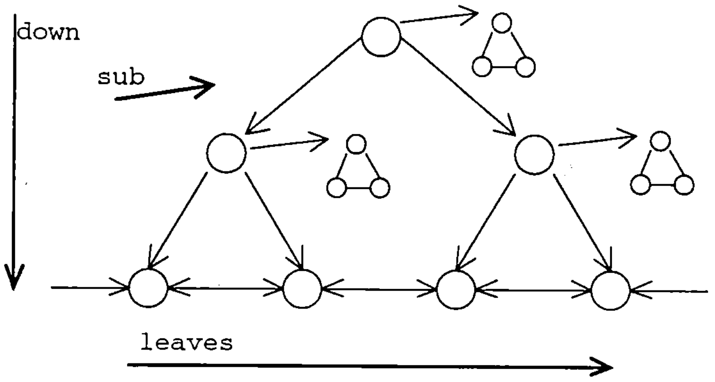


Note that each node may be accessed by traversing along **either** the X dimension **or** the Y dimension, but not both. Thus, any node that can be reached by a forward traversal along X cannot be reached by a forward traversal along Y, and vice versa. Hence X and Y are independent, which is conveyed using || in the following ADDS declaration:

[3]Unless otherwise stated (via ||), dimensions are assumed to be dependent. Such conservative nature is intentional.

```
type ListOfLists [X][Y] where X || Y
  { int          data;
    ListOfLists  *across is uniquely forward along X;
    ListOfLists  *back   is backward along X;
    ListOfLists  *down   is uniquely forward along Y;
    ListOfLists  *up     is backward along Y;
  };
```

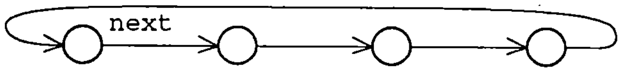
An interesting three-dimensional structure that has both dependent and independent dimensions is the *two-dimensional range tree* [Sam90], used to answer queries such as "find all points within the interval x1...x2" or "find all points within the bounding rectangle (x1,y1) and (x2,y2)." As illustrated below, it is a binary tree of binary trees, where the leaves of each tree are linked together into a two-way linked list:



The dimensions **down** and **leaves** are dependent (each leaf node can be reached by traversing along the **down** dimension and along the **leaves** dimension). However, observe that **sub** is independent of both **down** and **leaves**. That is, any node that can be accessed by a forward traversal along **sub**, cannot be accessed by a forward traversal along **down** or along **leaves**. This leads to the following declaration[4]:

```
type TwoDRangeTree [down][sub][leaves] where sub||down, sub||leaves
 {  int            data;
    TwoDRangeTree  *left, *right is uniquely forward along down;
    TwoDRangeTree  *subtree      is uniquely forward along sub;
    TwoDRangeTree  *next         is uniquely forward along leaves;
    TwoDRangeTree  *prev         is backward along leaves;
 };
```

Lastly, let us consider a common cyclic data structure, the circular linked-list:



This type of structure can be problematic, since a single field (**next**) is used for what are essentially two purposes, (1) traversing uniquely forward and (2) circling around. One declaration is thus:

```
type CircularList [X]
  { int           data;
    CircularList  *next is unknown along X;
  };
```

This is equivalent to saying nothing at all (the default), and the unknown nature of **next** prevents the compiler from performing possible optimizations (e.g. given the statement `p = q->next` the compiler must conservatively assume that p and q are aliases). One solution is the use of a more explicit declaration that more accurately reflects the properties of a circular list:

[4]Note that the field pairs (left, right) and (next, prev) could be overloaded to save storage; this is accomplished using a variant record (union), without loss of generality.

```
type CircularList [X]
  { int   data;
    int   un_type;
    union { CircularList  *next   is uniquely forward along X;
            CircularList  *around is unknown along X;
          } un;
  };
```

Obviously, this may require some rather clumsy coding practices. A second solution is to declare **next** as traversing **X** in a "circular" direction:

```
type CircularList [X]
  { int           data;
    CircularList  *next is circular along X;
  };
```

This solution appears more desirable, since the declaration does not require a change in the coding of the program. However, one needs to know the length of the list in order to perform accurate alias analysis. This implies information must be collected and maintained at run-time, and so is beyond the scope of this paper. Hence the circular direction is equivalent to the unknown direction, and thus is not present in ADDS.

3.2 Speculative Traversability

In all cases, a data structure described using ADDS is required to be *speculatively traversable* [HG91]. This property allows one to traverse past the "end" of a data structure without causing a run-time error. It can be automatically supported by the compiler, and places no additional burden on the programmer (except good programming practices—they must use the name NULL and not an arbitrary integer). This property is analogous to *computing* an array index outside the bounds of an array, but not actually using it. It is often important for performing loop transformations.

3.3 Summary of ADDS

In summary, ADDS is a technique for abstractly describing the important properties of a large class of useful data structures. Once known, these properties can be exploited by the compiler for analysis and transformation purposes. By default, a structure has one dimension D, and all recursive pointer fields traverse D in an unknown fashion. The programmer may refine this by describing (1) additional dimensions, (2) the interaction between dimensions, and (3) how the various fields traverse these dimensions. As illustrated with our examples, the choice of dimensions and directions is quite intuitive. Even though predefined types such as "list" or "tree" would be easier to use, these lack the flexibility to describe such data structures as the TwoDRangeTree. Instead, ADDS should be used to build such higher-level data structures into a programming language. In the next section we give a more formal definition of the properties of ADDS data structures.

4 Properties of ADDS Data Structures

In this section we present an overview of the formal properties of ADDS data structures. These definitions are not crucial to the understanding of the ideas, but they do show that our ADDS approach has some well-specified properties. For each definition we summarize the importance of the property being defined. If the reader is comfortable with the intuitive definitions presented in section 3, this section can be safely skipped.

Let N be a data structure declaration with n recursive pointer fields, $n \geq 1$. Let k be the number of programmer-specified dimensions over N, $1 \leq k \leq n$. The dimensions of N are denoted by $d_1, \ldots, d_k$, and the recursive pointer fields of N are denoted by $f_1, \ldots, f_n$. A recursive pointer field f may traverse along only one dimension d, and in only one of three directions: forward, backward, or unknown. pN denotes a dynamically-allocated node of type N, and $pN.f$ is either NULL[5] or denotes the dynamically-allocated node of type N reached

[5]Think of NULL as denoting a typeless node that is dynamically-allocated by the system at startup.

by traversing the pointer stored in field f of pN. Traversing a series of non-NULL fields is denoted using a regular expression notation, e.g.

- $pN(.f)^2$ denotes the node $pN.f.f$,
- $pN(.f)^+$ denotes any one of the nodes $pN.f$, $pN.f.f$, $\ldots$,
- $pN((.f)+(.g))^*$ denotes any one of the nodes pN, $pN.f$, $pN.g$, $pN.f.f$, $pN.f.g$, $pN.g.f$, $pN.g.g$, $\ldots$.

For any such regular expression R, R denotes a list of nodes which may or may not contain duplicates. If this list is **finite** (i.e. for all fields f in R, there exists a node pN' denoted by R such that $pN'.f$ is NULL), then there are no duplicates. Otherwise R denotes an **infinite** list of nodes, in which case there must be duplicates.

All nodes pN can be thought of as forming a single data structure pDS of type N. pDS is considered **well-behaved** if it adheres to the abstraction defined for N. In cases where pDS is not well-behaved (e.g. pDS is under modification) the abstraction must be ignored at these points in the program[6]. The definitions presented here assume a well-behaved data structure.

Def 4.1: if N is **speculatively traversable**, then for all fields f and for all nodes pN, if $pN.f$ = NULL then $pN.f.f$ is a valid traversal and also yields NULL.
[Implication: legal to traverse past what would normally be thought of as the "end" of a structure.]

Def 4.2: if f traverses d in the **forward** direction, then for all nodes pN, the list of nodes denoted by $pN(.f)^*$ is finite.
[Implication: traversing f never visits a node twice (f is acyclic).]

Def 4.3: if f traverses d in a **uniquely forward** direction, then Def 4.2 holds for f, and for all distinct nodes pN and pN', either

1. $pN.f = pN'.f =$ NULL, or
2. $pN.f \neq pN'.f$.

[Implication: traversing f from different nodes never visits the same node (f is acyclic and list-like).]

Def 4.4: if f traverses d in an **unknown** direction, then there may exist a node pN such that the list of nodes denoted by $pN(.f)^*$ is infinite.
[Implication: traversing f is unpredictable, could visit a node twice (f is potentially cyclic).]

Def 4.5: if a field b traverses d in the **backward** direction, then there exists a field f that traverses d in the forward direction.

Def 4.6: if f traverses d in a uniquely forward direction and b traverses d in the backward direction, then for all nodes pN, either:

1. $pN.f =$ NULL,
2. $pN.f.b =$ NULL, or
3. $pN.f.b = pN$.

[Implication: traversing f then b forms a cycle, otherwise f or b is NULL.]

Def 4.7: if f and g ($f \neq g$) traverse d in a **combined uniquely forward** direction, then Def 4.3 holds for f, Def 4.3 holds for g, and for all distinct nodes pN and pN',

1. $pN.f \neq pN'.g$ or $pN.f = pN'.g =$ NULL, and
2. $pN.g \neq pN'.f$ or $pN.g = pN'.f =$ NULL.

[Implication: f and g separate a structure into disjoint substructures (tree-like).]

Def 4.8: if fields $f_1, f_2, \ldots, f_m$ $(2 < m \leq n)$ traverse d in a combined uniquely forward direction, then for all f_i and f_j, $1 \leq i, j \leq m$ and $i \neq j$, Def 4.7 holds for f_i and f_j.
[Implication: generalization of Def 4.7—the m fields separate a structure into m disjoint substructures.]

[6] The problem of detection is discussed in the next section.

Def 4.9: (a) let d_i and d_j be dimensions of N $(i \neq j)$. If d_i and d_j are **independent**, then for any field f traversing d_i in a uniquely forward direction, for any field g traversing d_j in the forward direction, and for all distinct nodes pN and pN',

1. $pN.f \neq pN'.g$ or $pN.f = pN'.g =$ NULL, and
2. $pN.g \neq pN'.f$ or $pN.g = pN'.f =$ NULL.

(b) Further, for any field b traversing d_i in the backward direction, either

3. $pN.f =$ NULL, or
4. for all pN'' in the set denoted by $pN.f(.g)^*$, $pN''.b = pN$ or $pN''.b =$ NULL.

[Implication: (a) d_i and d_j separate a structure into disjoint substructures, (b) uniquely forward/backward cycles hold across an independent dimension.]

Def 4.10: let d_i and d_j be dimensions of N $(i \neq j)$. If d_i and d_j are not independent, then they are **dependent** and Def 4.9 does not hold for d_i and d_j.
[Implication: d_i and d_j could refer to a common substructure.]

5 Analysis and Transformations using ADDS

In this section we show how the properties of ADDS data structures can be used for improving both data structure analysis and optimizing/parallelizing transformations. The major difference between our approach, and previous analysis approaches, is that we are using information available in the ADDS declarations to guide the analysis. This synergy between the abstract data structure descriptions and the analysis technique provides a more general and more accurate approach. For example, by using information about the dimensionality and direction of field traversals, the abstraction approximations are freed from estimating needless cycles (such as those formed by the forward and backward directions along the same dimension), and can therefore avoid making needless conservative approximations.

In section 5.1 we present a new approach to alias analysis that combines existing techniques with information provided by an ADDS declaration, and in section 5.2 we present a new application of software pipelining that is made possible by our improved analysis.

5.1 Data Structure Analysis

As discussed earlier, a description mechanism such as ADDS is simply not enough by itself. Imperative programs routinely rearrange components of a data structure, and it is during these points in a program that the abstraction must be ignored to ensure correctness. Our approach then is a combined one, in which safe analysis techniques are used in conjunction with ADDS declarations. In particular, we are developing an approach to the static analysis of ADDS data structures that is an extension of *path matrix* analysis [HN90, Hen90], called *general path matrix* analysis.

Path matrix analysis was originally designed to automatically discover and exploit the properties of acyclic data structures. General path matrix analysis computes, for each program point, a path matrix PM which estimates the relationship between every pair of live pointer variables. The entry $PM(r, s)$ denotes an explicit path or alias, if any, from the node pointed to by r to the node pointed to by s. The analysis does not attempt to express all possible paths between two nodes, since cyclic data structures would soon overwhelm the matrix. Instead, the paths explicitly traversed by the program are captured in the PM, while the remaining paths and aliases are deduced from the current state of the matrix and the ADDS declarations.

Each new path matrix is generated from the current path matrix. The process is controlled by "pointer rules," which are applied on the basis of the program statement under analysis. For example, there is a rule to handle pointer statements of the form `A = B->f`. This is one of the more complex rules, since it must handle DAG and cyclic structures. However, the ADDS declaration can be used to simplify this rule, thus making the analysis more efficient. For example, if f is an acyclic field (i.e. Def 4.2 or 4.5 applies to f), then the rule developed in [Hen90] can be used, where detection of a cycle denotes a break in the abstraction.

As a more concrete example, consider the following code fragment. The pointer variable **head** denotes a two-way linked-list of points, where each point contains its x and y coordinates. The code below shifts the origin from (0,0) to (**head->x**,0) by subtracting the x-coordinate of this first point from all remaining points.

```
p = head->next;
while p <> NULL
 { p->x = p->x - head->x;
   p = p->next;
 }
```

	head	*p*	*p'*
head	=	=?	=?
p	=?	=	=?
p'	=?	=?	=

If the compiler fails to discover that **next** traverses a list in the forward direction (i.e. in an acyclic manner, see Def 4.2), then its analysis of the above code will be overly conservative—the compiler will assume that **head** and all values of p are potential aliases for the same node. This is expressed above in path matrix form (p' is used to summarize the iterative values of p).

Suppose the programmer declared their two-way linked-list using the ADDS declaration TwoWayList, as shown in section 3. If general path matrix analysis agrees that the structure is well-behaved at this point, the compiler can use the acyclic nature of the **next** field (Def 4.2) to infer that the statement p = p->**next** always traverses to a distinct node. The result are the following path matrices, which denote (from left to right): just before the loop, after one iteration, and after the loop analysis has reached a fixed point.

	head	*p*
head	=	*next*
p		=

	head	*p*	*p'*
head	=	$next^2$	*next*
p		=	
p'		*next*	=

	head	*p*	*p'*
head	=	$next^+$	$next^+$
p		=	
p'		*next*	=

An empty entry does not necessarily mean there is no path; it does however guarantee that the two pointers are not aliases. Thus the ADDS declaration and the general path matrix analysis have captured the desired property in PM, namely that **head** and p are never aliases.

5.2 Transformations

Optimizing and parallelizing transformations come in many forms, including:

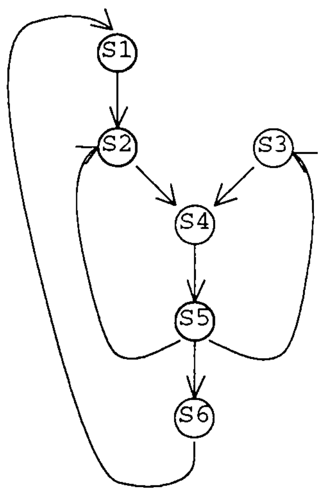
- fine-grain transformations (e.g. improved instruction scheduling),
- loop transformations (e.g loop unrolling or software pipelining), and
- coarse-grain transformations (e.g. parallel execution of code blocks).

The application of such transformations typically requires accurate alias resolution. As we discussed in the previous section, the ADDS approach allows us to develop more accurate alias analysis techniques. This will clearly aid in fine-grain transformations where dependency analysis is crucial. In addition, loop transformations typically require the data structure to exhibit list-like properties, and coarse-grain transformations typically require tree-like properties. Such properties are all expressible using ADDS declarations.

Earlier work [HG91] has shown the applicability of loop unrolling [DH79] on scalar architectures. For example, a simple loop to initialize every node of a linked-list showed 47% speedup on the MIPS architecture for a list of size 100 with 3-unrolling (see [HG91] for more details and timings). In this section we present an example of a more powerful loop transformation, software pipelining [RG82, AN88a, AN88b, Lam88, EN89]. Given the current trend towards machines supporting higher degrees of parallelism, e.g. VLIW and superscalar, this type of optimization will offer greater speedups.

Once again, consider the code fragment from the previous section. In pseudo-assembly code, the loop looks as follows:

```
S1   if p==NULL goto done
S2   load   p->x,R1
S3   load   head->x,R2
S4   sub    R1,R2,R3
S5   store  R3,p->x
S6   load   p->next,p
S7   goto   S1
```



As we saw earlier, if the compiler's analysis is overly conservative, then the compiler will incorrectly assume that **head** and all values of p are potential aliases for the same node. In this case the data dependency graph shown above is constructed, with false loop-carried dependencies from S5 to S2 and S3. The result is that the loop appears to exhibit very little parallelism.

However, if the structure is declared as a TwoWayList, general path matrix analysis will be able to determine that in fact **head** and p are never aliases. This will eliminate the false dependencies, and the loop now appears as a sequence of nearly independent iterations.

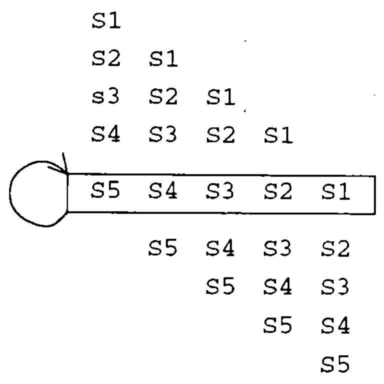
To exploit this parallelism, we now apply software pipelining. First we need to minimize the effect of the loop-carried dependence from S6 to S1. By renaming, we can move S6 above S2, calling it S1.6, and then replace S6 with a copy statement. The result is a semantically equivalent loop (shown on the left).

```
S1   if p==NULL goto done
S1.6 load   p->next,p'
S2   load   p->x,R1
S3   load   head->x,R2
S4   sub    R1,R2,R3
S5   store  R3,p->x
S6   move   p',p
S7   goto   S1
```

```
S0   load   head->x,R2
S1   load   p->next,p'
S2   if p==NULL goto done
S3   load   p->x,R1
S4   sub    R1,R2,R3
S5   store  R3,p->x
S6   move   p',p
S7   goto   S1
```

Note the next iteration of the loop can begin as soon as S1.6 of the current iteration completes, allowing the loop bodies to nearly overlap in execution. However, the result of the analysis enables the application of additional optimizing transformations. Since **head** and p are never aliases, **head->x** is loop invariant, and S3 can be moved outside and above the loop. Also, since the list is speculatively traversable, it is safe to swap S1 and S1.6 (Def 4.1), further increasing the available parallelism[7]. The code on the right incorporates these transformations (producing yet another semantically equivalent loop). This loop can now be pipelined as follows (columns denoted different iterations, rows denotes statements executing in parallel):

[7]If the architecture supports delayed branch slots, it would be wiser to leave S1.6 where it is, and use it to fill such a slot.



The boxed "statement" represents the new parallel pipelined body of the loop. As a result, the code exhibits a theoretical speedup of 5. Note that the copy statement S6 is removed as part of the pipelining process, via (enhanced) copy propagation [NPW91].

In general, software pipelining can lead to even larger speedups, depending on the characteristics of the loop body. However, the actual speedup also depends heavily on the underlying machine architecture.

6 Conclusions

As we have demonstrated with numerous examples in this paper, many recursively-defined pointer data structures exhibit important properties which compilers can exploit for optimization purposes. These properties are known to the programmer, and in imperative programming languages are often conveyed implicitly via appropriate identifiers. We have proposed an abstract description technique, ADDS, which allows the programmer to state such properties explicitly. The description of a recursive pointer data structure using ADDS is quite intuitive, and does not place an excessive burden on the programmer.

Combined with an extended form of path matrix analysis, called *general path matrix* analysis, this approach enables accurate analysis and alias resolution, and hence the application of numerous optimizations. As we have seen, software pipelining is just one such optimization.

ADDS represents an important first step in our long-term goal of efficient compiler analysis of codes involving general, cyclic pointer data structures. ADDS represents a simple extension of most any imperative programming language.

References

[AN88a] A. Aiken and A. Nicolau. Optimal Loop Parallelization. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 308–317, June 1988.

[AN88b] A. Aiken and A. Nicolau. Perfect Pipelining: A new loop parallelization technique. In *Proceedings of the 1988 European Symposium on Programming*. Springer Verlag Lecture Notes in Computer Science No.300, March 1988.

[App85] Andrew W. Appel. An Efficient Program for Many-Body Simulation. *SIAM J. Sci. Stat. Comput.*, 6(1):85–103, 1985.

[ASU87] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1987.

[BH86] Josh Barnes and Piet Hut. A Hierarchical O(NlogN) Force-Calculation Algorithm. *Nature*, 324:446–449, 4 December 1986.

[CWZ90] D.R. Chase, M. Wegman, and F.K. Zadek. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, 1990.

[DH79] J.J. Dongarra and A.R. Hinds. Unrolling loops in FORTRAN. *Software-Practice and Experience*, 9:219–226, 1979.

[EN89] Kemal Ebcioglu and Toshio Nakatani. A New Compilation Technique for Parallelizing Loops Loops with Unpredictable Branches on a VLIW Architecture. In *Proceedings of the Second Workshop on Programming Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing. MIT-Press, 1989.

[Gua88] Vincent A. Guarna Jr. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 212–220, 1988.

[Har89] W. Ludwell Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. Technical Report CSRD Rpt. No. 860, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, February 1989.

[Hen90] Laurie J. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell University, April 1990. TR 90-1114.

[HG91] Laurie J. Hendren and Guang R. Gao. Designing Programming Languages for Analyzability: A Fresh Look at Pointer Data Structures. In *Proceedings of the 4th IEEE International Conference on Computer Languages*, 1991.

[HN90] Laurie J. Hendren and Alexandru Nicolau. Parallelizing Programs with Recursive Data Structures. *IEEE Trans. on Parallel and Distributed Computing*, 1(1):35–47, January 1990.

[HPR89] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 28–40, June 1989.

[JM81] N. D. Jones and S. S. Muchnick. *Program Flow Analysis, Theory and Applications*, chapter 4, Flow Analysis and Optimization of LISP-like Structures, pages 102–131. Prentice-Hall, 1981.

[Kri90] S. M. Krishnamurthy. A Brief Survey of Papers on Scheduling for Pipelined Processors. *SIGPLAN Notices*, 25(7):97–106, 1990.

[Lam88] Monica Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[LH88a] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, June 1988.

[LH88b] James R. Larus and Paul N. Hilfinger. Restructuring Lisp programs for concurrent execution. In *Proceedings of the ACM/SIGPLAN PPEALS 1988 - Parallel Programming: Experience with Applications, Languages and Systems*, pages 100–110, July 1988.

[MR90] Michael Metcalf and John Reid. *Fortran 90 Explained.* Oxford University Press, 1990.

[NPW91] A. Nicolau, R. Potasman, and H. Wang. Register Allocation, Renaming and their impact on Fine-grain Parallelism. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[PW86] David A. Padua and Michael J. Wolfe. Advanced compiler optimization for supercomputers. *Communications of the ACM*, 29(12), December 1986.

[RG82] B. R. Rau and C. D. Glaeser. Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support. In *Proceedings of the 9th Symposium on Computer Architecture*, April 1982.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[Sta80] Thomas A. Standish. *Data Structure Techniques.* Addison-Wesley, 1980.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.