In the Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, Orlando, Florida, June 20-24, 1994, pages 218-229. Copyright © 1994 ACM (see notice below).

A General Data Dependence Test for Dynamic, Pointer-Based Data Structures

Joseph Hummel*
U. of California, Irvine

Laurie J. Hendren†
McGill University

Alexandru Nicolau‡
U. of California, Irvine

Abstract

Optimizing compilers require accurate dependence testing to enable numerous, performance-enhancing transformations. However, data dependence testing is a difficult problem, particularly in the presence of pointers. Though existing approaches work well for pointers to named memory locations (i.e. other variables), they are overly conservative in the case of pointers to unnamed memory locations. The latter occurs in the context of dynamic, pointer-based data structures, used in a variety of applications ranging from system software to computational geometry to N-body and circuit simulations.

In this paper we present a new technique for performing more accurate data dependence testing in the presence of dynamic, pointer-based data structures. We will demonstrate its effectiveness by breaking false dependences that existing approaches cannot, and provide results which show that removing these dependences enables significant parallelization of a real application.

1 Introduction and Motivation

High-performance architectures rely upon powerful optimizing and parallelizing compilers to increase program performance. One of the critical features of such compilers is accurate data dependence analysis [Ken90]. A good deal of work has been done in the area of array analysis (see [PW86, ZC90, Ban93] for extensive references), with notable success. However, much less work has been done in the area of pointer analysis, and with varying degrees of success.

This paper is concerned with developing an accurate data dependence test in the presence of dynamic, pointer-based data structures. This is a problem which continues to grow in importance, for two principal reasons. Firstly, there is an increasing use of languages which support pointers, in particular C, C++, Ada, and FORTRAN90. Secondly, pointers and dynamic data structures are important tools for achieving good performance. For example, *octrees* are important data structures in computational geometry [Sam90] and N-body simulations [App85, BH86, WS92], as are *sparse matrices* in circuit simulations [Kun86, SWG91] and many other applications.

Existing techniques are either overly conservative in the presence of dynamic data structures, or work well for only a small set of structures (linked-lists and trees). In this paper we present a new, general dependence test which can yield accurate results for a wide range of data structures, from simple tree-like structures to complex cyclic structures. Our test is based on theorem proving, using axioms which describe uniform properties of the data structure. The test is general since its accuracy grows with the accuracy of the axioms, and it supports any data structure which possesses some form of regularity. The test can be used to disprove dependences between two statements in a sequence, between loop iterations, and between statement blocks. As a result, our dependence test can enable fine-grain, loop-level, and block-level transformations, potentially yielding significant improvements in performance. It is important to note that theorem proving in this context is decidable. The power and general applicability of theorem proving is well known, as is its general undecidability. Thus, a more subtle contribution of our work is the application of theorem proving in a powerful yet decidable manner.

The format of this paper is as follows. In the next section we discuss the problem in more detail, and present related work. In Section 3 we introduce our

*Please direct correspondence to jhummel@ics.uci.edu. This work supported in part by an ARPA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland.

†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.

Copyright © 1994 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM Inc., fax +1 (212) 869-0481, or (permissions@acm.org).

approach by way of example, followed by a more formal presentation of our dependence test in Section 4. We then discuss the results of applying our test to a real application in Section 5, and present our conclusions in Section 6.

2 Related Work

Before discussing related work, it will be helpful to first clarify the problem we are trying to solve.

```
        p  = &i;                    q = malloc(...);
        ...                         insert(head, q);
S:      *p = 10;                    ...
        ...                         q = head;
T:      i  = 20;                    while ... {
                                U:      q->f = fun();
                                        q = q->link;
                                    }
```

Figure 1: The two different pointer problems.

2.1 Problem Clarification

The problem of dependence testing in the presence of pointers is better understood (and attacked) if separated into two, largely-distinct subproblems. The first concerns pointers to named memory locations (typically stack-based variables), the flavor of which is captured by the left code fragment of Figure 1. In this example, there is an output dependence from statement `S` to statement `T` *iff* `p` points to `i` at `S`.

The second subproblem involves pointers to unnamed memory locations, which arise from building data structures out of dynamically-allocated, heap-based memory. The right code fragment of Figure 1 depicts a common instance. In this case, there exists a loop-carried output dependence from the statement `U` to itself *iff* `q` from one iteration points to the same memory location as a `q` from a later iteration. Note that we are unable to refer to these memory locations by name.

We shall refer to the latter subproblem as the *pointer data structure dependence problem* (PDSDP), and the former as the *pointer target dependence problem* (PTDP). This paper is concerned solely with PDSDP[1].

2.2 Solution Components

Any accurate solution to the pointer data structure dependence problem must consist of at least three components, whose relationship is shown in Figure 2. In particular, it is important to note that the dependence tester has two inputs, each of which is distinct and can be obtained in various ways. Typically this information is obtained via automatic analysis of the program; in this case, two different analyses are necessary. When appropriate, we shall refer to these as *data structure* analysis and *memory reference* analysis, respectively. It should be noted that these forms of analysis are often tightly intertwined. This becomes obvious when a data structure is under modification structurally; maintaining accurate memory references requires an understanding of the data structure and how it is being changed.

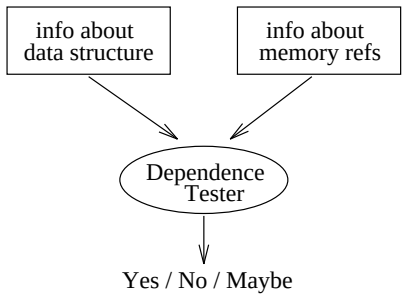


Figure 2: Solution components to PDSDP.

2.3 Extending Solutions to PTDP

There exist numerous approaches to the pointer target dependence problem [Cou86, HPR89, LMSS91, LR92, CBC93, MLR+93, EGH94], all of which follow a similar analysis and dependence testing framework: the program is analyzed (perhaps interprocedurally), and at each program point the set of aliased variables is computed. Dependence testing is then performed by simply intersecting the appropriate sets. This scheme works well in PTDP since pointers generally refer to named memory locations—i.e. variables. However, such *store-based* approaches [Deu92] do not extend well to the domain of dynamic, pointer-based data structures, for the simple reason that pointers may now refer to a seemingly infinite number of memory locations, while the dependence test is designed for a fixed number of memory locations.

The typical solution is to retain the dependence test from PTDP, and adopt a *k-limited* data structure analysis [JM82]. Given a set of dynamically-allocated memory locations, this has the effect of assigning k unique names to the first k memory locations, and a single *summary* name to all remaining memory locations. This quickly becomes overly conservative—consider a loop, for example. Even though each iteration may write to a different memory location (suppose the loop is updating a linked-list), at best the dependence test will prove that only the first k iterations are indepen-

[1] In C one must also consider pointers to dynamic arrays and multi-dimensional arrays constructed using pointers. Since the programmer's intent is a well-understood data structure (the array) for which numerous dependence tests already exist, we view this as a special case of PDSDP and feel it should be handled as such.

dent. It should be noted that the conservativeness of such k-limited approaches has been addressed to some degree by the work of Chase et. al. [CWZ90], and later improved upon by Plevyak et. al. [PCK94]. However, these techniques do not propose a more accurate dependence test, but rather a more accurate data structure analysis which extends the k-limited scheme to provide better information in the case of linked-lists and trees.

2.4 PDSDP Only

These approaches are based on a more powerful naming scheme, designed specifically for the pointer data structure dependence problem. The general idea is to name memory locations by *relation* to one another, and then test for intersecting relations. Given the potential complexity and number of such relations, the difficulty is designing an accurate dependence test. These approaches are sometimes referred to as *storeless* approaches [Deu92].

Larus et. al. [LH88] presented a technique for naming memory locations using *path expressions*. Viewing a data structure as a directed graph where edges are labeled with the names of their respective pointer fields, each vertex is labeled with a regular expression denoting a path through the data structure starting from a particular vertex v. Dependence testing is performed by intersecting path expressions, in particular the languages denoted by the corresponding regular expressions. If the intersection is empty, a dependence does not exist. Path expressions are computed during data structure analysis and stored in *alias graphs*. Memory reference analysis, however, collects *access paths*, which denote actual traversals of the data structure by the program. Thus, the dependence tester is employed by mapping the given access paths to their appropriate path expressions, using the alias graphs.

For trees, the dependence test of Larus et. al. is a precise one. However, for DAGS and graphs the results are overly conservative. The problem lies in the use of intersection, which forces an overly conservative mapping from access path to path expression. For example, consider the leaf-linked binary tree shown in Figure 3. Since the access paths $root.LLNN$ and $root.LRN$ lead to the same vertex, their corresponding path expressions must produce a non-empty intersection (note that the intersection of $LLNN$ and LRN would be empty, which is incorrect). Thus, to obtain correct results, these access paths are mapped to more conservative path expressions (e.g. $(L|R)^+N^+$). The result is that very similar access paths, such as $root.LLN$ and $root.LRN$, are likewise mapped, resulting in a non-empty intersection even though it can be proven that these access paths will never lead to the same vertex (see Section 3.3). It should be noted that more accurate mapping strategies may in fact exist; Larus et. al. did not address this (we explored the idea, but found our solutions to be inadequate).

Hendren et. al. [HN90] discussed a similar technique for naming memory locations, in this case using a simpler form of *paths*. Paths are collected in a *path matrix* such that all paths between memory locations of interest are known. This approach is potentially less expensive than that of Larus, yet also precise for trees. However, it fails to present a general dependence test, and does not handle cyclic data structures. The notion of access paths is also used by Deutsch [Deu92]. He discusses a method of alias analysis based on a new, potentially more powerful form of data structure analysis. However, a general dependence test is not presented. Finally, Guarna [Gua88] took a different approach and used syntax trees as a naming scheme. These trees are then intersected to detect dependences. Though his technique was applicable to both PTDP and PDSDP, in the latter domain it is limited to tree-like structures only.

2.5 Other Work

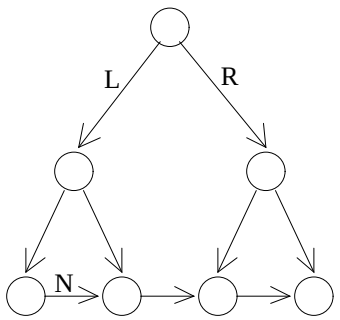
There has been substantial work done on program analysis in regards to dynamic allocation. Such analysis is useful in many related problems, e.g. reference counting and memory lifetimes [Hud86, ISY88, RM88, Har89, Bak90, WH92] as well as memory placement [Har89, HA93]. In particular, Harrison [Har89] performs extensive analysis of dynamically allocated memory; this work has been extended in [HA93]. From this analysis it is possible to discover various properties of a data structure (e.g. its treeness), and thus prove that certain dependences are not possible. However, a general technique of dependence testing which exploits this information is not presented.

Neirynck et. al. [NPD89] developed an abstract interpretation approach capable of coarse-grain dependence testing on higher-order applicative programs. The *effect* system [LG88] is a language-based approach in which the effect of a statement must be explicitly associated with a region of memory; this enables the compiler e.g. to perform coarse-grain dependence testing. Klappholz et. al. [KKK90] discuss another language-based approach in which dependence testing relies upon programmer-supplied *partition* statements and *software tagging*.

3 Overview of Our Approach

Our approach is most similar to that of Larus et. al. [LH88], since we also base our naming scheme on regular expressions. Regular expressions allow more accurate naming than k-limited schemes, possess important theoretical properties such as decidability, and fit naturally into a standard analysis framework (the effect of a statement sequence can be summarized using concatenation, selection through alternation, and

```
type LLBinaryTree_t
{ integer          d;
  LLBinaryTree_t  *L;
  LLBinaryTree_t  *R;
  LLBinaryTree_t  *N; }
```



where AXIOMS are
A1: $\forall$ p, p.L <> p.R,
A2: $\forall$ p <> q, p.$(L|R)$ <> q.$(L|R)$,
A3: $\forall$ p <> q, p.N <> q.N,
A4: $\forall$ p, p.$(L|R|N)^+$ <> p.ε;

Figure 3: A leaf-linked binary tree: type declaration, example data structure.

iteration via kleene star). Unlike Larus et. al., our dependence tester is based on theorem proving, providing a more powerful and accurate test.

The critical inputs to our dependence tester are twofold: (1) axioms which define various aliasing properties of the data structure, and (2) access paths for the two memory locations being referenced. The dependence tester then applies the axioms directly to the supplied access paths in an attempt to prove that no dependence is possible. If a proof exists, the dependence tester will find it and return `No`. Otherwise, the tester halts and either returns `Yes` (a dependence definitely exists) or `Maybe` (a dependence possibly exists). Theorem proving is applied in a completely general manner, allowing more accurate results given more accurate inputs.

In this first part of this section we define our notions of aliasing axioms and access paths. We then briefly overview the framework necessary to support our dependence tester. Finally, we present our test by way of example. A more formal discussion of our work follows in Section 4.

3.1 Aliasing Axioms and Access Paths

Aliasing axioms define aliasing properties which hold uniformly throughout a given data structure. An axiom can take one of three forms:

1. $\forall$ p, p.$RE1$ <> p.$RE2$,
2. $\forall$ p <> q, p.$RE1$ <> q.$RE2$, and
3. $\forall$ p, p.$RE1$ = p.$RE2$.

Viewing a data structure as a directed graph where edges are labeled with their corresponding pointer field names, the variables p and q refer to any vertex in the graph. The regular expressions $RE1$ and $RE2$ denote sets of paths through the data structure. Thus, the *access path* p.$RE1$ denotes the set of vertices reached by starting at vertex p and traversing any path in $RE1$.

The semantics of an aliasing axiom are straightforward. Given an axiom $\forall$ p, p.$RE1$ <> p.$RE2$, this states that $\forall$ vertices p, the access path p.$RE1$ never refers to the same vertex as the access path p.$RE2$. In other words, p.$RE1 \cap$ p.$RE2 = \emptyset$. Axioms of the form $\forall$ p <> q, p.$RE1$ <> q.$RE2$ are defined similarly. Finally, axioms of the form $\forall$ p, p.$RE1$ = p.$RE2$ imply the opposite: $\forall$ vertices p, the sets p.$RE1$ and p.$RE2$ are equivalent. This latter form is useful for describing cycles in a cyclic data structure.

As an example, consider the leaf-linked binary tree shown in Figure 3. Four axioms are supplied as part of the type declaration and define the critical properties of this data structure. The first two axioms define the treeness of the substructure consisting of `L` and `R` fields: `A1` states that `L` and `R` lead to different vertices from the same vertex, while `A2` adds that these fields never lead back to the same vertex from different vertices. Note these axioms alone do not completely describe a tree however, since they allow the possibility of a cyclic edge from a leaf back to the root. `A3` conveys that the substructure formed by the `N` fields is a linked-list, since it states that different vertices will never lead to the same node via `N`—forcing `N`-labeled edges into a linear ordering. Note that this axiom does not completely describe a linked-list, since it allows a cyclic edge from the last node in the list to the first. Finally, `A4` defines the acyclicness of the data structure, in particular stating that `L` and `R` form a true tree and that `N` forms an acyclic linked-list. For completeness, note that since the axioms do not say otherwise, it is assumed that the `L`, `R`, and `N` fields form a DAG, which they do.

Though exceedingly simple in nature, a set of aliasing axioms is able to describe quite complicated data structures, such as sparse matrices (see Section 5) and two-dimensional range trees (a leaf-linked tree of leaf-linked trees, used in computational geometry [PS85]). See [HHN94] for a more complete discussion.

3.2 Supporting Framework

As pictured in Figure 2 (and discussed in Section 2.2), an accurate dependence test for PDSDP requires two types of information: information about the data structure, and information about the referenced memory locations. In the case of our dependence test, the

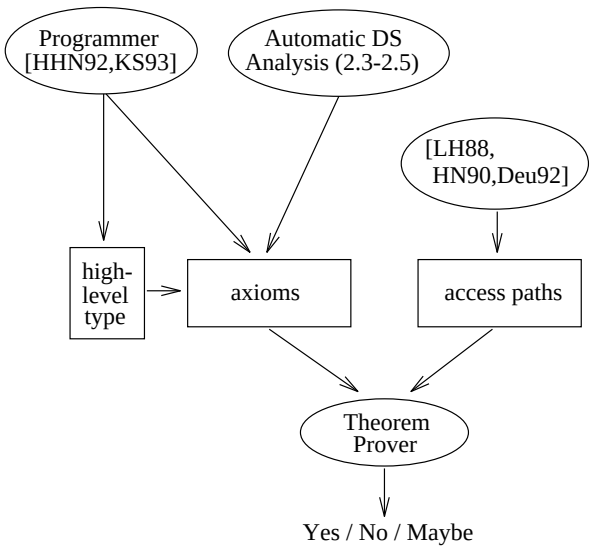


Figure 4: Our solution components.

former is supplied by way of the axioms, and the latter through access paths. This information can be collected in various ways, as depicted in Figure 4. Axioms can be collected automatically (using many of the techniques discussed in Section 2), or supplied by the programmer (and perhaps automatically verified). In the latter case, note that axioms can be specified indirectly using a higher level of abstraction, e.g. the ADDS data structure description language [HHN92] or *graph types* [KS93]. Access paths are straightforward to collect, since standard flow analysis techniques map nicely (and accurately) into regular expressions. Various forms of path collection are discussed in [LH88, HN90, Deu92].

3.3 An Example

We consider an example involving leaf-linked trees, a data structure used e.g. in N-body simulations [BH86]. In our case we consider binary trees, an instance of which is shown in Figure 3. Also shown in the figure are a set of axioms which we assume hold at the start of our example. Access paths will be collected and presented in the form of *access path matrices* (APM). There exists an APM at each program point, where each entry in an APM denotes a path (or set of paths) which may have been traversed up to (but not including) that point in the program. Note that an APM does *not* summarize all possible paths between vertices in a data structure, only paths explicitly traversed by the program. The key observation is that whenever possible, access paths should be collected in reference to fixed vertices in the data structure (Larus et. al. [LH88] collected access paths in this manner). We will refer to these vertices as *handles*. Handles are associated with pointer variables, and a new handle *_hp* is created each time its associated pointer variable p is assigned to. The one exception is when p is assigned a value relative to itself, in which case a new handle is not created; this is important e.g. when p is an induction variable for a loop (as will be the case in Section 5). Existing handles are destroyed when they are no longer needed, i.e. they no longer anchor *any* access path.

```
      subr(LLBinaryTree_t *root)
      { LLBinaryTree_t  *p, *q;

        root = root->L;
        p = root->L;
        p = p->N;
S:      p->d = 100;
        p = root;
I:      q = root->R
        q = q->N;
T:      return q->d;
      }
```

Consider the code fragment shown above. The question is whether or not statement `T` is dependent on statement `S`. When the analysis reaches `S`, we have the following APM:

APM	root	p
_hroot	L	LLN
_hp		N

There are two handles and three access paths; for example, the vertex denoted by `p` can be reached via the handle-based access path _hroot.LLN. Continuing, we obtain the following APM at statement `I`:

APM	root	p
_hroot	L	L
_hp		
_hp2		ε

Notice that a new handle _hp2 was added due to the assignment of `root` to `p`, and that the handle _hp is no longer of use and can be destroyed. Eventually, we obtain the following APM at statement `T`:

APM	root	p	q
_hroot	L	L	LRN
_hp2		ε	
_hq			N

Is there a dependence from `S` to `T`? We scan the APMs at statements `S` and `T`, looking for a handle common to both `p` and `q` (though a common handle is not required by the dependence test, it generally leads to more accurate results). Finding one in _hroot, we replace `p` at `S` with the access path _hroot.LLN, and `q` at `T` with _hroot.LRN. Note that since none of the pointer fields in the data structure have been modified between `S` and `T`, we know that `p`'s access path is still valid at `T`. Assuming that all four axioms of Figure 3

were valid at the start of `subr`, we also know that these axioms are valid when `T` is reached.

There is no dependence if it can be proven that for all possible roots, and across all possible data structures under which these axioms hold, the access paths can never lead to the same vertex. Thus, the theorem prover constructs the following theorem of no dependence, and then attempts to prove this theorem:

> Theorem: T is not dependent on S if $\forall$ vertices `_hroot`, `_hroot`.LLN <> `_hroot`.LRN.

The following paraphrased proof is derived automatically:

> Proof:
>
> - Applying `A3`, theorem is true if `_hroot`.LL <> `_hroot`.LR.
> - Since both paths start from the same vertex and begin with L, reduces to showing that `_hroot`$'$.L <> `_hroot`$'$.R.
> - Applying `A1`, this holds. □

The dependence test thus returns No, and we can conclude that no dependence exists from `S` to `T`.

3.4 Impact of Structural Modifications

When a data structure undergoes structural modification, i.e. one or more of its pointer fields is updated, this can invalidate both access paths and axioms. For example, inserting a new vertex may lengthen an access path, while the insertion process itself may temporarily break any "treeness" axioms. It is the job of the analyzer to identify and understand such structural changes, and adjust its information accordingly. This is a difficult and common problem in all forms of pointer analysis. On a more positive note however, we have found that many applications contain large portions of structurally read-only code, since the code which performs structural modifications is often localized for maintainability.

In our case, suppose we have a statement M which modifies a pointer field. If a dependence test is performed across M, then the access path moved over M may need to be adjusted. Likewise, the set of valid axioms may also need to be updated. As access paths are collected, the set of axioms valid after M must be recorded. If a dependence test is then given an access path whose construction occurs across M, the set of axioms to supply is thus the intersection of the axiom sets valid before and after M.

4 An Axiom-based Pointer Test

First we present the algorithm behind our dependence test, then a discussion of its time and space complexity.

4.1 Algorithm

APT, an Axiom-based Pointer Test, is a theorem proving system involving axioms specified using regular expressions. Pointer values are expressed as access paths, which are also based on regular expressions. APT is most appropriate for tests involving data structures, since the axioms can describe properties of the data structure, while the access paths can denote paths through the structure.

We assume the existence of two statement executions S and T, where S precedes T and each accesses a single field relative to some pointer (we assume that expressions involving multiple fields have already been simplified into this format [HDE+93]):

S: ... `p->`f ... ;
⋮
T: ... `q->`g ... ;

Furthermore, either S performs a write to `p->`f, T performs a write to `q->`g, or both, with no intervening write to the memory location denoted by `p->`f. There exists a data dependence from S to T, denoted $S \rightarrow T$, *iff* `p->`f and `q->`g denote overlapping memory regions.

The input to the dependence test consists of a set $\mathcal{A}$ of applicable axioms, two expressions E_S and E_T (corresponding to `p->`f and `q->`g respectively), and two valid access paths AP_p and AP_q (corresponding to $H_p.Path_p$ and $H_q.Path_q$ respectively). The dependence test returns `No` if it can be proven that no data dependence exists from S to T (with respect to the given expressions), or `Yes` if it can be proven that a data dependence definitely exists. Otherwise, the dependence test returns `Maybe`.

```
deptest(A, E_S, E_T, AP_p, AP_q)
begin
  if p and q are different types
    then return No;
  if f and g do not overlap
    then return No;
  assert(H_p = H_q);

  if Path_p = Path_q and |Path_p| = 1
    then return Yes;

  if proveDisj(A, Path_p, Path_q, ε, ε)
    then return No;

  return Maybe;
end deptest;
```

The first step is to check for the possibility of a data dependence. We assume that pointers are not cast freely between data structure types, and that pointers to a vertex v point to the start of v in memory and not elsewhere. These are quite reasonable assumptions, since a compiler will assume such when generating code

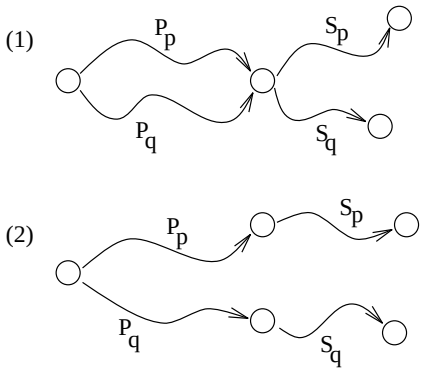


Figure 5: The two cases of `proveDisj`.

to access fields relative to a pointer. Though safe in ANSI C, these assumptions are not necessarily true in older K&R C. In this case special checks are needed to guarantee validity of the above assumptions. If these checks fail, then the first two tests in the dependence algorithm should be skipped. We also assume that access paths share a common handle; the test for different handles is nearly identical, although its accuracy depends on knowing the relationship between the two handles as well.

There is a definite dependence if the access paths are guaranteed to reach the same vertex; this is true if the paths $Path_p$ and $Path_q$ are identical **and** the cardinality of these sets is one. In this case the dependence test returns `Yes`. Otherwise, we attempt to prove the opposite, that a dependence is impossible.

The core of the dependence test is a general theorem proving system. The axioms are applied in all possible combinations in an attempt to prove that the access paths cannot lead to the same vertex. The idea is to apply the axioms to ever-increasing suffixes of $Path_p$ and $Path_q$, in an attempt to prove that these suffixes cannot lead to the same vertex. Since the distinctness of the vertices visited along each access path is unknown, a proof must consider two cases: (1) suffixes may originate from the same vertex, and (2) suffixes may originate from distinct vertices. The situation is shown in Figure 5, where S_p and S_q denote the suffixes and P_p and P_q the resulting prefixes. Note the direct correlation between the two principal forms of axioms ("$\forall$ `p` ..." and "$\forall$ `p` <> `q` ...") and the format of these two cases.

The core of our dependence test, `proveDisj`, thus begins as follows:

```
proveDisj(A, P_p', P_q', S_p', S_q')
begin
  if new suffixes do not exist
    then return False;
  let S_p = a longer suffix of path P_p' + S_p';
  let P_p = new prefix given choice of S_p;
  let S_q = a longer suffix of path P_q' + S_q';
  let P_q = new prefix given choice of S_q;

  A: let T1 = result of trying to prove:
       ∀ vertices x, x.S_p <> x.S_q;
  B: let T2 = result of trying to prove:
       ∀ vertices x <> y, x.S_p <> y.S_q;

  if T1 = True and T2 = True
    then return True;

  < remainder of proveDisj to follow; >
```

The results $T1$ and $T2$ are obtained by direct application of a single axiom a from $\mathcal{A}$. For example, consider the calculation of $T1$. For all axioms a of the form "$\forall$ `p`, `p`.$RE1$ <> `p`.$RE2$," the dependence tester tries to show that either $S_p \subseteq$ `RE1` and $S_q \subseteq$ `RE2`, or vice versa. If either case is true, then the proof of step `A` succeeds and $T1$ becomes $True$. Otherwise the proof fails and $T1$ is set to $False$. The question of subset can be answered using some general theory of regular expressions. Given two regular expressions $R1$ and $R2$, $R1 \subseteq R2$ if $M1 \cap$ `complement`$(M2) = \emptyset$, where $M1$ and $M2$ are the DFAs constructed from $R1$ and $R2$, respectively. The algorithmic details are discussed in [HU79, DDQ78].

However, if only $T1$ or $T2$ is true, it may still be possible to prove no dependence. Consider Figure 5 once again:

```
  C: if T1 and can prove H_p.P_p = H_q.P_q
       then return True;
  D: if T2 and can prove H_p.P_p <> H_q.P_q
       then return True;

  E: < alternative, kleene star processing; >

  F: return proveDisj(A, P_p, P_q, S_p, S_q);
end proveDisj;
```

Step `C` corresponds to case (1) and is true if the access paths $H_p.P_p$ and $H_q.P_q$ denote a definite dependence. Step `D` corresponds to case (2) and can be answered recursively: `proveDisj`$(\mathcal{A}, P_p, P_q, \varepsilon, \varepsilon)$. If these attempts at a proof fail, the algorithm continues recursively in search of new suffixes and a successful proof (steps `E` and `F`). Eventually, the algorithm will either find a proof and succeed, or if no such proof exists, halt and fail. This completes the core of the dependence testing algorithm.

Two issues remain, however. Firstly is the algorithm behind suffix generation. A regular expression consists of zero or more components, where each component is either ε, a single field name[2] f, two alternative components $a|b$, a kleene star component a^*, or a parenthesized component (a). Suffixes are selected by starting with the original paths $Path_p$ and $Path_q$ and generating three sets of suffixes:

(1,1): S_p = last component of $Path_p$ and
S_q = last component of $Path_q$,
(1,0): S_p = last component of $Path_p$ and $S_q = \varepsilon$,
(0,1): $S_p = \varepsilon$ and S_q = last component of $Path_q$.

This process is then repeated recursively for case (1,1). For cases (1,0) and (0,1), only cases (1,0) and (0,1), respectively, are repeated. This approach generates the exact set of all possible suffixes.

Secondly is the handling of parenthesized, alternative, and kleene star components: if the current prefix ends in one of these components, how is a new suffix generated? In the case of parenthesized components, the parentheses are simply stripped before suffix generation. For an alternative component $a|b$, it is first treated as a single component (in the hope of finding a proof more quickly) and processed as before by `proveDisj`. However, if the proof fails, we then split the alternatives and attempt to prove each case separately. The difference is that both alternatives must result in a successful proof in order for the original proof to succeed.

Finally, given a kleene star component, at first it is treated as a single component. If the proof fails, then induction is employed. Suppose that only one of the prefixes ends in a kleene component, denoted by a^*. An inductive proof requires three cases, each emphasizing a replacement for a^*:

1. replace with ε,
2. replace with a,
3. assume a^*a and replace with a^*aa.

If the inductive step (3) cannot be proven directly from the inductive hypothesis, the theorem prover proceeds recursively in search of a proof (note that the replacement string contains a^*). This will require the proof of additional base cases, and may even require further recursion. Eventually however, the process terminates since the original proof will either succeed (and thus halt) or fail on some base case due to the finite number of axioms (and thus halt). If both prefixes end in kleene components, the resulting proof contains four major cases, summarized as follows (we use kleene '+' to simplify the presentation, and we assume that P_p ends in a^* and P_q ends in b^*):

1. replace with $(\varepsilon, \varepsilon)$,
2. replace with (ε, b^+),
3. replace with (a^+, ε),
4. replace with (a^+, b^+).

This last case is handled inductively via four sub-cases:

4.1 replace with (a, b),
4.2 replace with (a^+, b),
4.3 replace with (a, b^+),
4.4 assume (a^+, b^+) and replace with (a^+a, b^+b).

Thus, a total of seven cases are required when both prefixes end in kleene components.

4.2 Complexity

The algorithm will either find a proof (if one exists), or halt with failure (if one does not)[3]. However, this may require an exponential amount of time in the worst case. Suppose the original paths $Path_p$ and $Path_q$ contain n components. There exist $O(n^2)$ sets of suffixes which must be checked, and thus $O(n^2)$ different proofs to be explored (this assumes that the results of intermediate proofs are cached so that a proof attempt is never repeated, and that the cache is maintained and searched efficiently). We shall refer to this set of proofs as $\mathcal{P}$.

Now we consider the cost of exploring one proof in $\mathcal{P}$, which corresponds to an execution of `proveDisj`. If the paths do not contain alternative nor kleene star components, the cost of this proof is based on two factors. Firstly, step `D` may cause an intermediate proof. However, this intermediate proof is in $\mathcal{P}$, so we pay this cost only once—either now or later. Hence we ignore it here. Secondly, steps `A` and `B` require searching $\mathcal{A}$ for an applicable axiom to use in the proof. Each such search requires a subset operation, which is dominated by the time it takes to convert the two regular expressions into equivalent DFAs. In the worst case, this conversion process may produce DFAs of size $O(2^n)$. With m axioms, the time cost of steps `A` and `B` becomes $O(m2^n)$. Therefore, in the worst case, the cost of exploring a single proof is exponential in terms of both time and space.

If a proof in $\mathcal{P}$ involves a kleene component and initially fails (i.e. steps `A-D` fail), the cost of induction (step `E`) is a constant number of intermediate proofs (most of which are not in $\mathcal{P}$). This constant depends on the number of base cases that are ultimately required, and is a function of the axioms. However, in the worst case, each intermediate proof may in turn involve kleene star components. This has a multiplicative

[2] For simplicity we assume that arrays of pointers do not exist, and that field names are unique across type declarations. Lifting these restrictions is simply more detail.

[3] Note that the problem of static analysis in the presence of pointers has been shown to be undecidable [Lan92]. In relation to our work, this undecidability result corresponds to the problem of performing accurate data structure and memory reference analysis (see Figure 2).

effect, resulting in a proof with a worst-case exponential time complexity of $O(c^n)$. Worst-case space complexity is also exponential due to the DFA construction process.

The same holds true if a proof in $\mathcal{P}$ involves an alternative component. In this case however, an exponential number of axioms is required to force worst-case exponential time.

In practice we have found that paths are relatively short (n is on the order of ten) and relatively simple (few kleene and alternative components). This is a consequence of both the analysis (with its numerous handles) and the nature of typical applications (which feature well-structured code and highly-regular data structures). Proof times thus become dominated by the RE to DFA conversion process, which generally runs in $O(n^2)$ time and space complexity. Thus, we have found that in practice, our dependence algorithm requires $O(n^4)$ time and $O(n^2)$ space. Of course, the proof process can be pruned heuristically and cutoff points set, allowing a tradeoff between accuracy and efficiency. This may even be user controllable, e.g. via a compiler option.

5 Results

To demonstrate the effectiveness of our dependence test, a prototype was implemented and applied in a realistic situation—sparse matrices as used e.g. in circuit simulations [Kun86].

Sparse matrices are often implemented using *orthogonal lists* [Sta80], an example of which is shown in Figure 6 (along with suitable type declarations). The fundamental operations performed on sparse matrices are scaling, factoring, and solving; in terms of execution time, scaling and solving are linear in the size of the data structure, whereas factoring is quadratic. We shall focus on factorization here.

Gaussian elimination is used to factor the matrix, where the crucial consideration is minimizing *fillins* (elements added to the matrix as a direct result of the factorization process). Good pivot selection is one of the keys to reducing the number of fillins, and thus considerable effort is spent in selecting the best possible pivot element at each factorization step. Here is a high-level overview of `factor`:

```
factor(M)
begin
  for each successive row R in M
  { let SM = submatrix[R+1..N, R+1..N];
    compute fillin heuristic for each elem in SM;
    search SM for best pivot p;
    adjust M to bring p into pivot position;
    add fillins to SM;
    perform elimination on each row of SM; }
end factor;
```

Notice that for each row of the matrix, four of the five steps operate on the entire submatrix. Thus, each such step executes in a row-by-row (or column-by-column) fashion, something akin to:

```
      let r := p->nrowE;
L1:   while r != NULL
      { let e := r->ncolE;
L2:     while e != NULL
        { S: ...
          let e := e->ncolE; }
        let r := r->nrowE;
      }
```

It turns out that in each step, the effect of statement `S` is either local to the element e or the row r, revealing a good deal of parallelism in both the outer and inner loops.

The problem is exploiting this parallelism in the presence of a complex, pointer-based data structure. We shall consider parallelizing the the outer loop `L1`; similar logic can be applied to the inner loop `L2`. Consider the first iteration of `L1`. The elements accessed are summarized by the path expression $\texttt{_hr}.ncolE(ncolE)^*$. The elements accessed in subsequent iterations are thus summarized by $\texttt{_hr}.(nrowE)^+ncolE(ncolE)^*$. In fact, since r is the induction variable for `L1`, these two paths expressions hold for any two iteration executions i and j ($i < j$). This leads to the following theorem of no dependence:

> Theorem T: `S` is not loop-carried dependent on itself at the level of `L1` if $\forall$ vertices `_hr`, $\texttt{_hr}.ncolE^+ <> \texttt{_hr}.nrowE^+ncolE^+$.

Since the sub-structure formed by the fields `ncolE` and `nrowE` is a DAG and not a tree, T cannot be proven by simply intersecting the given path expressions (see Section 2.4). Instead, the intersection of two, more conservative path expressions must be performed, resulting in a non-empty intersection and thus an unsuccessful proof. K-limited approaches will likewise fail to find a proof.

However, the theorem prover we have outlined in Section 4 will be able to derive a proof automatically, assuming it is given enough information. At the very least, the theorem prover must know that: (1) rows form linked-lists (not DAGs), (2) the end of a row or column does not lead to the start of a different row or column, and (3) the sub-structure is acyclic. This information can be conveyed by the following three axioms, respectively:

`A1`: $\forall$ `p` <> `q`, `p`.$ncolE$ <> `q`.$ncolE$,
`A2`: $\forall$ `p`, `p`.$ncolE^+$ <> `p`.$nrowE^+$, and
`A3`: $\forall$ `p`, `p`.$(ncolE|nrowE)^+$ <> `p`.ε.

These axioms can either be supplied by the programmer (note that a single axiom along the lines of The-

```
type SparseMatrix_t
{ RowHdr_t  *rows;
  ColHdr_t  *cols; };

type RowHdr_t
{ Element_t  *relems;
  RowHdr_t   *nrowH; };

type ColHdr_t
{ Element_t  *celems;
  ColHdr_t   *ncolH; };

type Element_t
{ real        value;
  Element_t  *nrowE;
  Element_t  *ncolE; };
```

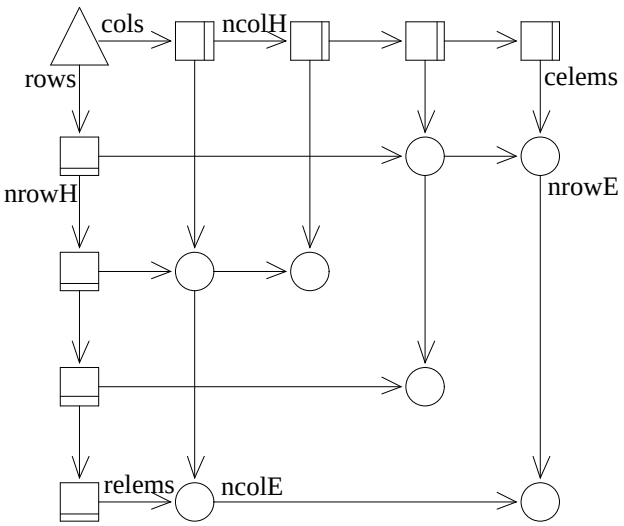


Figure 6: A sparse matrix: type declarations and example data structure.

orem T will also suffice), or obtained using automatic data structure analysis techniques (although current approaches appear too limited to derive these axioms automatically). Regardless, these axioms are sufficient for our theorem prover to prove T, thus breaking a critical false dependence and revealing a significant amount of parallelism to the compiler. The proof has been omitted due to its length (there are four initial cases since each access path ends in '+', and many of these contain multiple sub-cases); the complete set of axioms for a sparse matrix are given in Appendix A.

The importance of breaking these false dependences is demonstrated by the speedup figures shown in Figure 7. To collect these results, we manually applied loop-level transformations and ran the resulting code on an 8-PE Sequent multiprocessor. Given the discussion in Section 3.4, we collected two types of results. Firstly, we assumed a simplistic analysis which only collected access paths for structurally read-only portions of the code. These are the *partially* parallel results. Secondly, we assumed a more sophisticated analysis capable of handling modifications to the structure of the sparse matrices. These are the *fully* parallel results. In the partial case, the speedups are good but not linear. The fully parallel case comes much closer to linear speedup. However, it remains sub-linear since one of the factorization steps ("adjust M to bring p into pivot position") is inherently sequential.

1000x1000, N=10,000	2 PEs	4 PEs	7 PEs
Factor only (partial)	1.7	2.5	3.1
Scale, Factor, Solve (partial)	1.7	2.4	3.0
Factor only (full)	1.8	3.3	5.2
Scale, Factor, Solve (full)	1.8	3.3	5.2

Figure 7: Sparse matrix speedup results.

6 Conclusion

We have presented a new dependence test (**APT**) appropriate for dynamic, pointer-based data structures. It is more accurate than existing approaches, and supports any data structure which possesses some form of regularity. Our test is a general one based on theorem proving, allowing it to produce more accurate results given more accurate information. This information is supplied in the form of axioms, which describe uniform properties of the data structure, and access paths, which denote the memory locations in question. In essence, the dependence tester applies the axioms to the access paths in an attempt to prove that these paths will never refer to the same memory location. Access paths are easily collected using standard analysis techniques, and axioms can be obtained in a number of ways. Hence APT is not dependent on any particular form of data structure analysis.

For the expected case, our approach is practical. Its use can break false dependences between statements in a sequence, iterations of a loop, or blocks of statements. This in turn can enable the application of performance-enhancing transformations at the statement, loop, and block level. The effectiveness of our test was demonstrated using a real example, namely sparse matrices.

References

[App85] Andrew W. Appel. An efficient program for many-body simulation. *SIAM J. Sci. Stat. Comput.*, 6(1):85–103, 1985.

[Bak90] H. Baker. Unify and conquer (garbage, updating, aliasing, ...) in functional languages. In *Pro-*

ceedings of the '90 ACM Conference on LISP and Functional Programming, June 1990.

[Ban93] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer, 1993.

[BH86] Josh Barnes and Piet Hut. A hierarchical O(NlogN) force-calculation algorithm. *Nature*, 324:446–449, 4 December 1986. The code can be obtained from Prof. Barnes at the University of Hawaii, or from jhummel@ics.uci.edu.

[CBC93] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In *Proceedings of the ACM 20th Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[Cou86] D. Coutant. Retargetable high-level alias analysis. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 110–118, January 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.

[DDQ78] P. Denning, J. Dennis, and J. Qualitz. *Machines, Languages, and Computation.* Prentice-Hall, 1978.

[Deu92] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the IEEE 1992 International Conference on Computer Languages*, pages 2–13, April 1992.

[EGH94] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1994.

[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.

[HA93] W. Ludwell Harrison III and Z. Ammarguellat. A program's eye view of Miprac. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Fifth International Workshop on Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 512–537. Springer-Verlag, 1993.

[Har89] W. Ludwell Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, 1989.

[HDE+93] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Fifth International Workshop on Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 406–420. Springer-Verlag, 1993.

[HHN92] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249–260, June 1992.

[HHN94] J. Hummel, L. Hendren, and A. Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *Proceedings of the 8th International Parallel Processing Symposium*, April 1994.

[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.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Hud86] P. Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, 1986.

[ISY88] K. Inoue, H. Seki, and H. Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM TOPLAS*, 10(4):555–578, October 1988.

[JM82] N. D. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *9th ACM Symposium on Principles of Programming Languages*, pages 66–74, 1982.

[Ken90] K. Kennedy. Foreword of *Supercompilers for Parallel and Vector Computers*, 1990. The text is written by Hans Zima with Barbara Chapman, available from the ACM Press.

[KKK90] David Klappholz, Apostolos D. Kallis, and Xiangyun Kang. Refined C: An update. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 331–357. The MIT Press, 1990.

[KS93] N. Klarlund and M. Schwartzbach. Graph types. In *Proceedings of the ACM 20th Symposium on Principles of Programming Languages*, pages 196–205, January 1993.

[Kun86] K. Kundert. Sparse matrix techniques. In A. Ruehli, editor, *Circuit Analysis, Simulation and Design*, pages 281–324. Elsevier Science Publishers B.V. (North-Holland), 186.

[Lan92] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4), December 1992.

[LG88] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings 15th ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[LH88] 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.

[LMSS91] J. Loeliger, R. Metzger, M. Seligman, and S. Stroud. Pointer target tracking - an empirical study. In *Proceedings of Supercomputing '91*, pages 14–23, November 1991.

[LR92] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[MLR+93] T. Marlowe, W. Landi, B. Ryder, J. Choi, M. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *ACM SIGPLAN Notices*, 28(9):67–70, September 1993.

[NPD89] A. Neirynck, P. Panangaden, and A. J. Demers. Effect analysis in higher-order languages. *International Journal of Parallel Programming*, 18(1):1–37, 1989.

[PCK94] J. Plevyak, A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Sixth International Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 37–56. Springer-Verlag, 1994.

[PS85] F. Preparata and M. Shamos. *Computational Geometry: An Introdution.* Springer-Verlag, 1985.

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

[RM88] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 285–293, 1988.

[Sam90] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, 1990.

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

[SWG91] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, 1991. FTP to mojave.stanford.edu.

[WH92] E. Wang and P. Hilfinger. Analysis of recursive types in LISP-like languages. In *Proceedings of the '92 ACM Conference on LISP and Functional Programming*, pages 216–225, June 1992.

[WS92] M. Warren and J. Salmon. Astrophysical n-body simulations using hierarchical tree data structures. In *Proceedings of Supercomputing 1992*, pages 570–576, November 1992.

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

Appendix A: Sparse Matrix Axioms

The following twelve axioms can be used to accurately describe a sparse matrix. Due to space limitations we merely present the axioms here; for a more detailed discussion see [HHN94]. Note that some axioms are inferred since pointer fields of different types should lead to different vertices.

We develop the axioms bottom-up, starting from the perspective of the matrix elements and finishing with the root of the sparse matrix. Firstly, we convey that the rows and columns form linked-lists, and that from any element of the matrix the next row and column elements are distinct:

$$\forall\ \mathtt{p}\mathtt{<>}\mathtt{q},\ \mathtt{p}.nrowE \mathtt{<>} \mathtt{q}.nrowE,$$
$$\forall\ \mathtt{p}\mathtt{<>}\mathtt{q},\ \mathtt{p}.ncolE \mathtt{<>} \mathtt{q}.ncolE,$$
$$\forall\ \mathtt{p},\ \mathtt{p}.nrowE \mathtt{<>} \mathtt{p}.ncolE.$$

Next, we state directly that rows are disjoint, likewise for columns:

$$\forall\ \mathtt{p},\ \mathtt{p}.ncolE^* \mathtt{<>} \mathtt{p}.nrowE^+ ncolE^*,$$
$$\forall\ \mathtt{p},\ \mathtt{p}.nrowE^* \mathtt{<>} \mathtt{p}.ncolE^+ nrowE^*.$$

We also say that row and column headers form linked-lists:

$$\forall\ \mathtt{p}\mathtt{<>}\mathtt{q},\ \mathtt{p}.nrowH \mathtt{<>} \mathtt{q}.nrowH,$$
$$\forall\ \mathtt{p}\mathtt{<>}\mathtt{q},\ \mathtt{p}.ncolH \mathtt{<>} \mathtt{q}.ncolH.$$

Once again we state the disjointness of the rows (columns), this time from the perspective of the row (column) headers:

$$\forall\ \mathtt{p}\mathtt{<>}\mathtt{q},\ \mathtt{p}.relem(ncolE)^* \mathtt{<>} \mathtt{q}.relem(ncolE)^*,$$
$$\forall\ \mathtt{p}\mathtt{<>}\mathtt{q},\ \mathtt{p}.celem(nrowE)^* \mathtt{<>} \mathtt{q}.celem(nrowE)^*.$$

Since the root vertex always refers to the first row (column) header, we view the root vertex as part of the row (column) header linked-list:

$$\forall\ \mathtt{p}\mathtt{<>}\mathtt{q},\ \mathtt{p}.rows \mathtt{<>} \mathtt{q}.nrowH,$$
$$\forall\ \mathtt{p}\mathtt{<>}\mathtt{q},\ \mathtt{p}.cols \mathtt{<>} \mathtt{q}.ncolH.$$

Finally, we state that a sparse matrix is acyclic:

$$\forall\ \mathtt{p},\ \mathtt{p}.(rows|cols|relems|celems|nrowH|ncolH|nrowE|ncolE)^+ \mathtt{<>} \mathtt{p}.\varepsilon.$$

As an aside, note that since a sparse matrix has only one root vertex, we can easily state the disjointness of sparse matrices:

$$\forall\ \mathtt{p}\mathtt{<>}\mathtt{q},\ \mathtt{p}.(rows|cols)(relems|celems|nrowH|ncolH|nrowE|ncolE)^* \mathtt{<>} \mathtt{q}.(rows|cols)(relems|celems|nrowH|ncolH|nrowE|ncolE)^*.$$