In the Proceedings of the 8th International Parallel Processing Symposium, Cancun, Mexico, April 26-29, 1994, pages 208-216. Copyright © 1994 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

A Language for Conveying the Aliasing Properties of Dynamic, Pointer-Based Data Structures

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

Abstract

High-performance architectures rely upon powerful optimizing and parallelizing compilers to maximize performance. Such compilers need accurate program analysis to enable their performance-enhancing transformations. In the domain of program analysis for parallelization, pointer analysis is a difficult and increasingly common problem. When faced with dynamic, pointer-based data structures, existing solutions are either too limited in the types of data structures they can analyze, or require too much effort on the part of the programmer. In this paper we present a powerful description language for expressing the aliasing properties of dynamic data structures. Such descriptions provide the compiler with better information during alias analysis, and require only minimal effort from the programmer. Ultimately, this enables a more accurate program analysis, and an increased application of performance-enhancing transformations.

1 Introduction

High-performance architectures rely upon powerful optimizing and parallelizing compilers to increase program performance. In turn, these compilers depend upon accurate program analysis to enable various optimizing and parallelizing transformations. A good deal of work has been done in the area of array analysis (see [ZC90] for extensive references), but much less work has been focused on pointer analysis. In particular, this paper is concerned with the problem of analyzing programs involving dynamic, pointer-based data structures.

The problem of analyzing these kinds of programs, and those using pointers in general, continues to grow in importance. This can be attributed to the increasing use of C in the parallel processing community (also C++ and F90 to a lesser extent), along with the realization that dynamic data structures are important tools for achieving high performance. For example, *octrees* are important data structures in N-body simulations [App85, BH86, WS92] and computational geometry [Sam90], as are *sparse matrices* in circuit simulations [Kun86].

The preferred method of analysis is an automatic one, in which the compiler deduces the properties of the data structure and transforms the program accordingly. However, such approaches [JM81, LH88, HPR89, CWZ90, HN90, Deu92, HA92, LR92, CBC93, PKC93] are quite limited in the types of data structures they can recognize (typically one and two-way linked-lists, and trees). A second approach is a manual one, in which the programmer annotates their program with directives, thereby directly supplying the compiler with the necessary analysis information (e.g., see [Cor91]). Such directives are typically at a very low semantic level, and must be placed throughout a program to maximize their effect.

Unhappy with either method, we recently proposed a third, compromising approach: the programmer annotates the type declarations of their data structures, and the compiler then uses this information to guide the remaining program analysis [HHN92a]. This requires only a small effort on the part of the programmer, but can result in dramatic improvements in accuracy of analysis and hence performance [HHN92b, HHN93a]. The disadvantage is that the compiler may not be able to verify the type annotations. As with program correctness, this is currently the responsibility of the programmer.

The format of this paper is as follows. In the next section we present our language for conveying the properties of data structures that are important for program analysis, namely *aliasing* properties. Then

*Dept. of Computer Science, University of California at Irvine, Irvine, CA 92717 USA. This work supported in part by a DARPA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland. Correspondence should be directed to this author (jhummel@ics.uci.edu, phone: 714-725-2248, fax: 714-856-4056).

†School of Computer Science, McGill University, Montreal, Quebec H3A 2A7 Canada. This work supported in part by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.

‡Dept. of Computer Science, University of California at Irvine, Irvine, CA 92717 USA. This work supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215.

in Section 3 we will discuss related work, followed in Section 4 by examples of using our new language. Finally, in Section 5 we present our conclusions.

2 Overview of language

First we motivate our language and its design, then we supply its definition, and lastly we discuss the problem of verification.

2.1 Motivation

One of the most critical features of optimizing and parallelizing compilers is accurate program dependence analysis [Ken90]. In the case of pointers, this requires accurate *alias* analysis, i.e. answering questions of the form: "At a program point S, what memory locations might be (are) pointed to by the pointer variable p?". When considering dynamic data structures, alias analysis most often involves two pointers p and q, and the question: "Could (do) p and q point to the same node in the data structure?". In other words, could $p = q$?

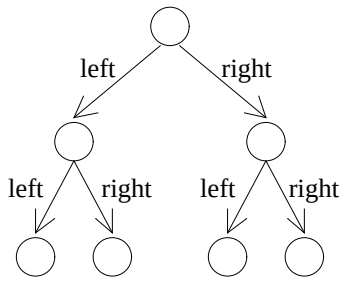

Figure 1: Example binary tree.

The idea then of our description language is to allow the specification of axioms which define under what conditions it is guaranteed that $p <> q$ — i.e. that no alias exists between p and q. It is these properties which must be expressed, since the goal of the compiler during dependence analysis is to prove that aliases are not possible. Thus, our language focuses on the communication of the (non) aliasing properties of a data structure to the compiler. Viewing a data structure as a directed graph where edges are labeled by their respective pointer fields, such aliasing properties can be expressed by stating the relationships between pointer fields. For example, one of the important properties of a binary tree (Figure 1) is that for all nodes in the tree, traversing `left` leads to a different node than traversing `right`. This can be stated by the following axiom:

$$\forall\, p,\ p.\texttt{left} <> p.\texttt{right}.$$

The same is true for distinct nodes: `left` and `right` traversals from different nodes always lead to different nodes. In fact, this holds for any combination of `left` and `right` traversals, implying the following axiom:

$$\forall\, p <> q,\ p.(\texttt{left}|\texttt{right}) <> q.(\texttt{left}|\texttt{right}).$$

Finally, we know that binary trees are acyclic; the `left` and `right` fields never form a cycle, neither independently nor together. This can be specified as follows:

$$\forall\, p,\ p.(\texttt{left}|\texttt{right})\texttt{+} <> p.\epsilon.$$

In general, axioms have the form $id_1.P_1 <> id_2.P_2$, where P_1 and P_2 are regular expressions. The alphabet of these expressions is the set containing the empty string ϵ, which denotes no traversal, and all available pointer fields (qualified by type name if name clashes arise, and with array indices if necessary). Regular expressions are a natural choice for expressing aliasing properties, since the information collected during standard compiler analyses can also be represented in this form: a sequence of statements can be expressed through concatenation, selection through alternation, and iteration through kleene star. Ultimately, this allows for a straightforward application of the axioms during dependence analysis [HHN93a]. For example, consider the following code fragment in which `p` and `root` are variables of type pointer to binary tree:

```
p = root;
if (...)
  then p = p->left;
  else p = p->right;
while (...)
  do p = p->left;
p->d = ... ;     /* stmt S */
...  = root->d;  /* stmt T */
```

Is statement `T` dependent on statement `S`? To answer this question, the compiler needs to know the relationship between `p` and `root`. When statement `S` is reached, standard analysis techniques are able to at least partially summarize their relationship with the following *access path*:

```
p = root.(left | right)left*
```

In other words, we know that `p` can be accessed from `root` by following any path in the set denoted by the regular expression `(left | right)left*`. To prove that statement `T` is not dependent on statement `S`, it

suffices to show that `p` and `root` refer to different nodes at `T`. Given the above access path for `p` (which still holds at `T`), this can be shown through a proof that `root.(left | right)left* <> root`. Since this is trivially true using the acyclicness axiom for binary trees, the compiler can conclude that no dependence exists between `S` and `T`[1].

2.2 Definition

We shall refer to our language as **ASAP**, since it supports the abstract specification of aliasing properties. The syntax of ASAP is straightforward: a valid ASAP specification consists of one or more axioms, of which there are three forms. These are defined by the extended BNF shown in Figure 2. For simplicity of presentation, we assume that all field names are unique (even across types), and that a field denotes a single pointer value (not an array of pointers); these limitations are easily lifted.

```
<spec>     ::= <axiom> +
<axiom>    ::= <same-NA> | <disj-NA> | <def-A>
<same-NA>  ::= ∀ ID ',' 
                 ID '.' <ref1> <> ID '.' <ref2>
<disj-NA>  ::= ∀ ID1 <> ID2 ','
                 ID1 '.' <ref1> <> ID2 '.' <ref2>
<def-A>    ::= ∀ ID ','
                 ID '.' <ref1> = ID '.' <ref2>
<ref1>     ::= <re>
<ref2>     ::= <re>
<re>       ::= <field> + | '(' <s> ')' [<kleene>]
<s>        ::= <re> | <re> '|' <re>
<field>    ::= ID | ϵ
<kleene>   ::= '*' | '+'
```

Figure 2: Extended BNF definition of ASAP.

The semantics of a valid ASAP specification S are likewise straightforward. For all data structures DS which will be built, each of the axioms in S is guaranteed to hold. An axiom of the form `<same-NA>` specifies that $\forall$ nodes p in DS, the set of nodes[2] reachable from p along any path in the set `<ref1>` will be disjoint from the set of nodes reachable from p along any path in the set `<ref2>`. The meaning of a `<disj-NA>` axiom is defined similarly. However, a `<def-A>` axiom specifies the opposite (a definite alias): $\forall$ nodes p in DS, the set of nodes reachable from p along any path in the set `<ref1>` and the set of nodes reachable from p along any path in the set `<ref2>` all in fact refer to the same node n. The motivation for this type of axiom will become clear in Section 4.

2.3 Verification

Given the expressive power of ASAP, it is unlikely that a compiler will be able to verify an ASAP specification in the context of all possible programs. Thus, as with program correctness, the correctness of an ASAP specification must ultimately be the responsibility of the programmer. One workable solution is to limit the use of ASAP to abstract data types, which are small enough to be proven correct (by hand if necessary). Another is the availability of ASAP-aware tools for code analysis, debugging, run-time checking, and visualization. However, regardless of the approach taken, the problem of verification should not necessarily deter the use of ASAP, much the same way it does not deter programming in general.

3 Related work

First we will discuss the work from which ASAP grew, and then other related work.

3.1 ADDS

Our work on ASAP is a generalization of our earlier work on **ADDS**, a data structure description language based on a notion of *directions* and *dimensions* [HHN92a]. ADDS is a simple yet powerful language for describing pointer-based data structures. However, as the complexity of a data structure grows, the accuracy of ADDS can diminish. As we tried to increase the accuracy of ADDS, and in general enlarge the class of describable data structures, the language quickly became more complicated and much less intuitive. As a result, instead of trying to extend (and possibly ruin) ADDS, we studied the underlying information that ADDS was conveying to the compiler, and designed a language that allowed this information to be expressed directly. This led to the development of ASAP. ASAP has more descriptive power than ADDS, which we will demonstrate in Section 4 by presenting several important data structures that ADDS is unable to describe accurately. The apparent disadvantage in using ASAP over ADDS is the lower-level nature of ASAP's axioms, making the language less intuitive. However, since an ADDS description can automatically be translated into an equivalent ASAP

[1] Details on how access paths are collected, and how general dependence testing is then performed, can be found in [HHN93a].

[2] A *node* is a memory location with a non-NULL address.

specification [HHN93b], a programmer can use ADDS when s/he wishes, and use ASAP only when necessary. Thus, the use of ASAP does not necessarily imply a loss of intuitiveness.

3.2 Graph types

Graph types [KS93] and ASAP take a similar approach to describing data structures, namely using regular-like expressions to specify the relationships between pointer fields. With graph types, pointer fields are separated into two types, *tree* and *routing* fields. The tree fields must create a spanning tree for the data structure; the routing fields are then defined in terms of the tree fields and the underlying spanning tree.

The primary disadvantage of graph types is that they only support data structures with a spanning tree backbone. However, the class of describable structures is even smaller, since a structure must also be *deterministic*—the relationship between a routing field and the underlying backbone must be precisely known at compile-time. For example, a *skiplist* [Pug90] is a data structure in which some pointers skip ahead x number of nodes in order to reduce search time. Since x cannot be predicted at compile-time, this type of structure cannot be described using a graph type. The same is true for *orthogonal lists* [Sta80], an important data structure used to implement sparse matrices; we will discuss sparse matrices further in Section 4.4.

It should be noted however that there are some cases in which graph types can yield a more accurate description than ASAP. For example, graph types can accurately describe a post-order threaded tree; ASAP is unable to express such threading as precisely. Another nice feature of graph types is that the compiler is able to guarantee the well-formedness of the regular-like expressions. However, this does not guarantee that the structure described is actually the same one the programmer had in mind—only that the description is deterministic. Furthermore, the compiler is not guaranteeing the correctness of imperative-style code which might build this kind of data structure. In fact, graph types are intended for functional languages in which the compiler is responsible for correctly maintaining the routing fields.

3.3 Other related work

IDL, the Interface Description Language, allows the programmer to define various properties of a data structure by way of assertions supplied in its declaration [Sno89]. Unlike ASAP however, these assertions do not aid analysis since the compiler does not *understand* their implication; the purpose of these assertions is to ensure validity at run-time (in particular, as the structure is passed from one running program to another). Larus [Lar89] discussed a technique for describing acyclic structures in LISP; these were code annotations which conveyed the acyclicness of a data structure to the compiler. Another code annotation approach is that taken by Klappholz et al. in Refined-C [KKK90], where explicit statements such as `distinct` are inserted into the program to guide analysis. Finally, a very different but related language-based approach is the *effect* system of FX [LG88], in which the effect of a statement must be explicitly associated with a region of memory; this enables the compiler e.g. to perform a coarse-grain alias analysis.

4 Examples

When describing simple data structures such as linked-lists and trees, the programmer will likely find it easier to use ADDS or graph types. In this case we view ASAP as a lower-level language into which ADDS or graph type descriptions can be "compiled" and the resulting axioms fed to a dependence analysis framework [HHN93a]; this separates the description language of choice from the underlying analysis algorithms. However, as this section will demonstrate, there exist important data structures for which neither ADDS nor graph types suffice. In these situations the programmer can use ASAP directly to provide a more accurate description, thus enabling more accurate program analysis.

In this section we shall present three data structures which pose problems for both ADDS and graph types. In the case of ADDS, all three of the structures can be described to some degree, just not as accurately as ASAP. With graph types, the first structure can be described as accurately as ASAP, but the remaining two cannot be described at all.

4.1 General trees

A general tree is one in which nodes may have any number of children. As shown in Figure 3, this can be done by forming a linked-list out of the children nodes; the `child` field points to the first child, and the rest of the children are reached by traversing along the `sib` field. A third field, `parent`, allows direct access to the parent from the last child.

It is the cyclic relationship involving the `child`, `sib`, and `parent` fields which prevents an accurate ADDS description. The problem is that ADDS is

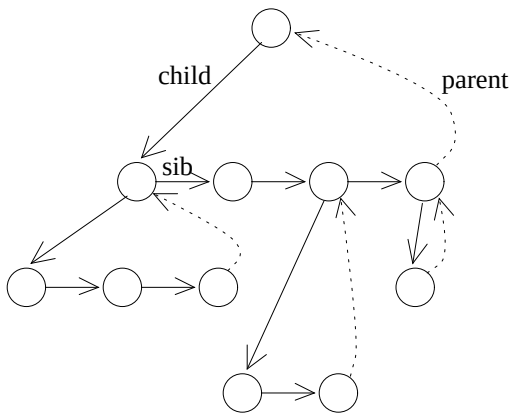

Figure 3: A general tree.

only able to express simple cycles involving two fields. ASAP does not share this limitation.

The first step is to express the treeness of the data structure; we shall mirror the approach discussed earlier in Section 2. For all nodes in the tree, we know that traversing `child` and `sib` will lead to different nodes. We state this with the following axiom:

$$\forall\ p,\ p.\texttt{child} <> p.\texttt{sib}.$$

Likewise, traversing these fields from distinct nodes will never lead to the same node. In fact, traversing any combination of `child` and `sib` fields from different nodes will never lead to the same node. Thus:

$$\forall\ p <> q,\ p.(\texttt{child}|\texttt{sib}) <> q.(\texttt{child}|\texttt{sib}).$$

Finally, we specify the acyclicness of the tree:

$$\forall\ p,\ p.(\texttt{child}|\texttt{sib})+ <> p.\epsilon.$$

The next step is to consider the `parent` field, and its relationship to the other fields `child` and `sib`. We know that from a given node, traversing `parent` will lead to a different node than traversing either `child` or `sib`:

$$\forall\ p,\ p.\texttt{parent} <> p.(\texttt{child}|\texttt{sib}).$$

However, when traversing from different nodes it *is* possible for a `parent` field to lead to the same node as a `child` or `sib` field, thus no axiom is appropriate. On the other hand, if we consider only the `parent` field, it is true that different nodes cannot reach the same node. Hence:

$$\forall\ p <> q,\ p.\texttt{parent} <> q.\texttt{parent}.$$

Next, we state the acyclicness of the `parent` field:

$$\forall\ p,\ p.(\texttt{parent})+ <> p.\epsilon.$$

Finally, and perhaps most importantly, we want to convey the cyclic relationship that `parent` forms with the `child` and `sib` fields. This can be stated as follows:

$$\forall\ p,\ p.\texttt{child}(\texttt{sib})*\texttt{parent} = p.\epsilon.$$

This ASAP specification describes a general tree more accurately than an ADDS declaration. In this case it is also true that a graph type could be used to achieve the same accuracy as our ASAP specification. However, this will not be true for the examples that follow.

4.2 Developing an ASAP specification

Before proceeding, it might be helpful to summarize a technique for developing ASAP specifications. Firstly, one should consider the relationships between pointer fields when originating from a single node. Then repeat this process, except consider the relationships when originating from distinct nodes; in this case consider not only how a given field relates to other fields, but also how this field relates to itself. Next, state which fields (and combinations of fields) are acyclic. Finally, express any cyclic relationships.

The low-level nature of ASAP axioms can lead to somewhat lengthy specifications. If this becomes an impediment, note that higher-level languages such as ADDS can be used and the description translated into ASAP; the result can then be supplemented with ASAP axioms as necessary. Also, note that it can usually be assumed that pointer fields of different types cannot lead to the same node[3]; thus there is no need to restate the obvious.

4.3 Union-Find trees

Union-find trees are a data structure used to support efficient *find* and *union* operations on data elements [AHU74, Man89]. The find operation searches for the group in which an element resides, while the union operation unions two existing groups into a new group. Initially, each element resides in a group by themselves. Various forms of search trees are well-known to allow efficient searching; the problem is to support efficient unions as well. The union-find tree shown in Figure 4 highlights one approach for fast unions; note that this is not a tree in the typical sense, since the pointers flow from the "leaves" to the

[3]Though this is a safe assumption in ANSI C, older C code will require special checks before this can be assumed.

```
type Group_t {
  GName_t   g;
  Group_t  *ngroup;
};

type Element_t {
  EName_t     e;
  Group_t    *fgroup;
  Element_t  *next;
};
```

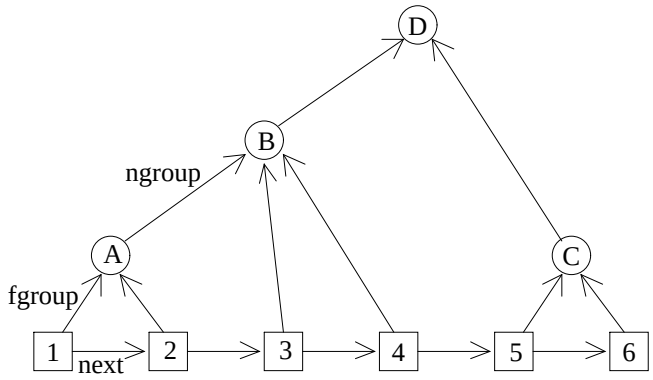

Figure 4: A union-find tree: type declaration and example data structure.

"root", but we shall remain consistent with the literature. The find of element x is accomplished by traversing from x to the root of its tree; e.g., *Find*(5) is D. Unions are thus performed by simply linking trees (i.e. root nodes) together. For example, D = *Union*(B, C) is brought about by simply creating a new node D and having the nodes B and C point to D. By adding just two links, six elements have been unioned together.

Graph types cannot be used to declare this type of data structure for the simple reason that no subset of the pointer fields forms a spanning tree backbone. A union-find tree is actually a DAG, not a tree, with multiple pointers converging on both its interior nodes and the "root." An ADDS description can capture portions of the data structure, but overall the result is not very accurate. The problem is that ADDS is unable to express the relationship between `fgroup` and `ngroup`—the only choices are tree or graph. The fact that these fields form a DAG cannot be expressed.

ASAP, however, can be used to generate an accurate description. We start by first considering the simpler group nodes. There is only one pointer field in the node, so axioms of the form `forall nodes p...` do not make sense. Considering disjoint nodes, it *is* possible for disjoint nodes to lead to the same node by traversing `ngroup` fields (consider nodes B and C in Figure 4), so an axiom of this form is not appropriate. Lastly, we consider acyclicness, which we postpone since it can be more accurately stated at the end.

Next we consider the element nodes. Since `next` and `fgroup` are of different types, their leading to different nodes is a given (from the same node and from distinct nodes). However, we know that from any two distinct nodes, `next` fields will not lead to the same node. We specify this with the following axiom:

$$\forall\ p <> q,\ p.\texttt{next} <> q.\texttt{next}.$$

Note that the same does not hold for `fgroup` fields (consider nodes 1 and 2). Next, since `fgroup` and `ngroup` are of the same type, we consider their relationship in the case of distinct nodes (traversing from a single node does not make sense, since these fields will never be part of the same node). However, from distinct nodes it is possible for these to lead to the same node (consider nodes A and 3), so once again an axiom is not appropriate.

Finally, we consider the acyclicness of the element nodes, and in fact the group nodes as well. This can be stated by the following single axiom:

$$\forall\ p,\ p.(\texttt{next}\,|\,\texttt{fgroup}\,|\,\texttt{ngroup})\texttt{+} <> p.\epsilon.$$

Note this axiom states that all substructures are acyclic as well, e.g. the list of elements, traversing from an element to a group node, etc.

Thus, an ASAP specification consisting of only two axioms describes the union-find tree more accurately than ADDS, and describes a structure which graph types cannot describe at all.

4.4 Sparse matrices

Sparse matrices, an example of which is shown in Figure 5, are implemented using orthogonal lists [Sta80]. Sparse matrices are an important data structure in scientific computing, and are commonly used e.g. in circuit simulations [Kun86]. The elements of the matrix form linked-lists by row and by column, with a header node at the front of each such list. These header nodes are in turn linked into one of two lists, depending on whether they head a row or a column. Finally, the pointers to these row and column lists are stored in a single header node, which serves to denote the entire sparse matrix.

```
type SparseMatrix_t {
  RowHdr_t  *rows;
  ColHdr_t  *cols;
};
type RowHdr_t {
  integer     row;
  Element_t  *relems;
  RowHdr_t   *nrowH;
};
type ColHdr_t {
  integer     col;
  Element_t  *celems;
  ColHdr_t   *ncolH;
};
type Element_t {
  Data_t      d;
  integer     r, c;
  Element_t  *nrowE;
  Element_t  *ncolE;
};
```

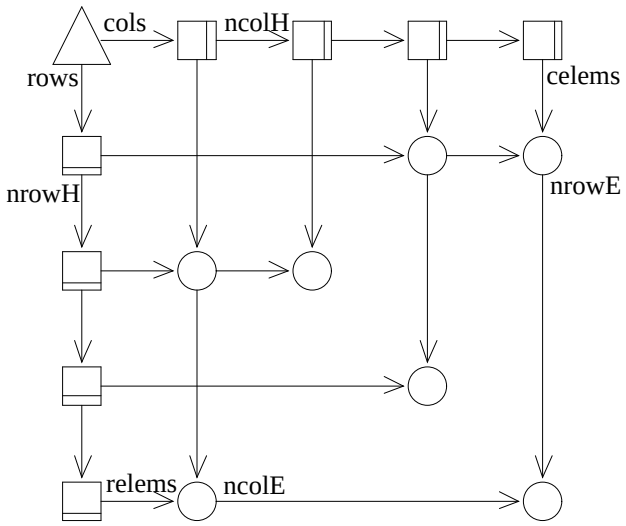

Figure 5: A sparse matrix: type declaration and example data structure.

The key distinguishing characteristic of this data structure is that the row and column element lists are intertwined, allowing elements to be accessed by row and column. However, equally important is the fact that individual rows are disjoint; the same is true for individual columns. This fact is crucial for parallelization of many sparse matrix algorithms, since such algorithms frequently operate row by row or column by column. Thus, any description of a sparse matrix should include this information.

The sparsity of the data structure prevents the use of graph types since the programmer cannot predict at compile-time exactly which node a given node will refer to via its `nrowE` (or `ncolE`) field; this depends on the state of the matrix at run-time. With ADDS, a sparse matrix can at least be partially described. The key problem however is that ADDS is unable to express the disjointness of the rows (and likewise the columns)—the very property often needed for parallelization.

ASAP has the necessary flexibility to allow a more accurate description of a sparse matrix. Since an ADDS declaration can be automatically translated into an ASAP specification [HHN93b], we know ASAP can do no worse than ADDS. We can however do much better. The idea is to take a bottom-up approach, beginning with simple axioms and developing more complicated ones as necessary. Starting with the element nodes, we see that the rows and columns are intertwined into a DAG. Thus, we follow the specification of a tree (as done in Section 2), relaxing this specification where appropriate. As with a tree, traversing an `nrowE` field leads to a different node than traversing an `ncolE` field:

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

Unlike a tree however, it is possible for these fields to lead to the same node from different nodes (hence the DAG). But rows of elements, as well as columns, are list-like, yielding the following two axioms:

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

Likewise, rows of elements are disjoint, as are columns. This can be stated as follows:

$$\forall\, p,\ p.(\texttt{ncolE})* <> p.(\texttt{nrowE})+(\texttt{ncolE})*.$$
$$\forall\, p,\ p.(\texttt{nrowE})* <> p.(\texttt{ncolE})+(\texttt{nrowE})*.$$

Lastly, the acyclicness of the element nodes should be specified; this is better postponed however, since it can be done more accurately at the end.

The next step is to consider the row and column header nodes, and their accompanying element lists as appropriate. Since the pointer fields in these nodes are of different types, many of the axioms can be inferred. We need only consider the cases where fields of the same type are involved. Firstly, the row and column header nodes each form a linear linked-list, implying

that the `nrowH` and `ncolH` fields never lead to the same node from different nodes:

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

Likewise, the `relems` and `celems` fields never lead to the same node from differing nodes:

$$\forall\ p <> q,\ p.\texttt{relems} <> q.\texttt{relems}.$$
$$\forall\ p <> q,\ p.\texttt{celems} <> q.\texttt{celems}.$$

It is possible however for the `relems` and `celems` fields to lead to the same node from different header nodes; consider row two and column one in Figure 5. Thus, an axiom is not appropriate. On the other hand, the row and column headers provide another, slightly different opportunity for specifying row or column disjointness:

$$\forall\ p <> q,\ p.\texttt{relems(ncolE)*} <> q.\texttt{relems(ncolE)*}.$$
$$\forall\ p <> q,\ p.\texttt{celems(nrowE)*} <> q.\texttt{celems(nrowE)*}.$$

Note that these axioms subsume the previous two. The last step is to state something about the acyclicness of these fields, which we once again postpone.

The final step is to consider the main header node. Since the `rows` and `cols` fields are of different types, nothing must be stated concerning only these fields. But the main header node does have fields of the same type as the row and column header nodes, and so these relationships should be considered. Given that the main header node always points to the first row and column header node in each list, we can state the following:

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

Finally, we need to consider the acyclicness of the fields in the main header node. Since we have reached the end however, we consider the entire sparse matrix, which we know is acyclic. This implies the following axiom:

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

Note that this states that all substructures are acyclic as well, e.g. a list of elements, the row header list, etc.

The above ASAP specification accurately describes a sparse matrix, in particular conveying the important properties of row and column disjointness necessary for parallelization. This specification is more accurate than any ADDS declaration, and describes a data structure that graph types cannot.

5 Conclusion

ASAP is a powerful language for conveying the aliasing properties of dynamic, pointer-based data structures. It is more powerful than existing data structure description languages, and the format of its axioms provides direct support for alias analysis. Furthermore, ADDS declarations can be automatically translated into ASAP specifications, allowing the programmer to use ASAP or ADDS (or both), depending on what is appropriate or preferred.

With the ability to describe a larger class of data structures, the use of ASAP will allow more accurate program analysis of a larger class of programs. Ultimately, this will enable the increased application of compiler-based, performance-enhancing transformations on such programs. A dependence testing framework based on ASAP is currently being implemented as part of an optimizing and parallelizing C compiler.

Acknowledgements

The authors wish to thank the anonymous referees for their helpful comments; the result is a better paper.

References

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

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

[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.

[Cor91] CONVEX Computer Corporation. CONVEX C optimization guide, April 1991.

[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.

[Deu92] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the IEEE '92 International Conference on Computer Languages*, April 1992.

[HA92] W. Ludwell Harrison III and Z. Ammarguellat. A program's eye view of Miprac. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, pages 339–353, August 1992. Available as Yale Tech Report YALEU/DCS/RR-915.

[HHN92a] Laurie J. Hendren, Joseph Hummel, and Alexandru 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.

[HHN92b] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. Applying an abstract data structure description approach to parallelizing scientific pointer programs. *Proceedings of the 21st Annual International Conference on Parallel Processing, Volume II*, pages 100–104, August 1992.

[HHN93a] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general dependence test for dynamic, pointer-based data structures. Technical Report ICS 93-44, UC-Irvine, September 1993. To appear in the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation.

[HHN93b] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. Technical Report ICS 93-43, UC-Irvine, September 1993. To appear in 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.

[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.

[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.

[Lar89] James R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, University of California, Berkeley, 1989.

[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.

[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.

[Man89] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.

[PKC93] J. Plevyak, V. Karamcheti, and A. Chien. Analysis of dynamic structures for efficient parallel execution. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the 6th Annual Workshop on Languages and Compilers for Parallel Computing*, pages c1–c20, August 1993.

[Pug90] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *CACM*, 33(6):668–676, June 1990. An implementation can be FTPed from mimsy.cs.umd.edu.

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

[Sno89] R. Snodgrass. *The Interface Description Language: Definition and Use*. Computer Science Press, 1989.

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

[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.