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

TECHNICAL REPORT 93-43

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

September 1993

[*]This work supported in part by an ARPA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland. Correspondence should be directed to this author via email to jhummel@ics.uci.edu, or via post to UC-Irvine, Dept. of ICS, Irvine, CA 92717, USA.

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

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

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

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

September 20, 1993

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.

Recently we proposed a compromising approach, in which the programmer supplies some small amount of information, and the compiler performs the remaining analysis [HHN92a]. However, the language for conveying this information is unable to accurately describe a number of important data structures, and does not easily support the compiler's underlying alias analysis. In this paper we present a more powerful description language that also directly supports alias analysis. Ultimately, this will lead to more accurate program analysis for a larger class of programs, and hence the increased application of performance-enhancing transformations on these programs.

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, Ban93] 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

[*]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.

[App85, BH86, HHN92b, WS92] and computational geometry [Sam90], as are *sparse matrices* in circuit simulations [Kun86, SWG91].

The preferred method of analysis is an automatic one, in which the compiler deduces the properties of the data structures and transforms the program accordingly. However, such approaches [JM81, LH88, ISY88, Har89, HPR89, CWZ90, HN90, Deu92, HA92, LR92, CBC93, PKC93] require interprocedural analysis and are currently quite limited in the types of data structures they can recognize (typically one and two-way linked-lists, and trees). A second analysis approach is the manual one, in which the programmer must annotate their program with directives, thereby directly supplying the compiler with analysis information (e.g., see [Cor91, McN93]). These directives are typically at a very low semantic level, and must be placed throughout the program to maximize effect.

Unhappy with either method, we recently proposed a third, compromising approach: the programmer annotates only the type declarations for 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, HHN93]. 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 **ADDS** description language we proposed in [HHN92a] is intuitive and fairly powerful in its descriptive abilities. However, it is unable to accurately describe a number of important data structures, and does not easily support a direct method of alias analysis testing. In this paper we present a new description language called **ASAP** that is both more powerful and provides direct support for alias analysis. Furthermore, ASAP's flexibility allows ADDS descriptions to be directly translated into equivalent ASAP descriptions, thereby giving programmers the flexibility to use either language (or both) as appropriate or preferred.

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 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. For the interested reader, Appendix A outlines the process of translating ADDS descriptions into ASAP descriptions.

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 $p = q$?

```
type BinaryTree_t {
  Data_t        d;
  BinaryTree_t *left;
  BinaryTree_t *right;
};
```

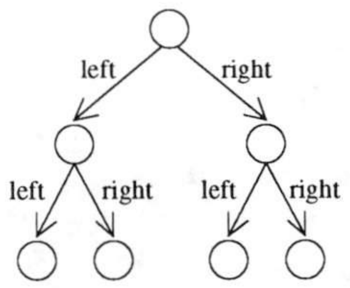

Figure 1: A binary tree: type declaration and example data structure.

The idea then of our description language is to allow the specification of axioms which define under what conditions it is guaranteed[1] 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 nodes p, p.left <> p.right.
```

The same is true for distinct nodes; `left` and `right` traversals from different nodes always lead to different nodes (as well as `left` and `left`, or `right` and `right`, traversals). This implies the following axiom:

```
forall distinct nodes p and q, p.(left|right) <> q.(left|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 nodes p, p.(left|right)+ <> p.
```

In general, axioms are of the form $var_1.path_1 <> var_2.path_2$, where $path_1$ and $path_2$ 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 mapped into this form: a sequence of statements

[1] Assuming that the axioms are true of course.

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 [HHN93]. For example, consider the following code fragment in which p and root are variables of type pointer to BinaryTree_t:

```
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 must be shown 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[2].

2.2 Definition

Since our language supports the abstract specification of aliasing properties, we shall refer to it as **ASAP**. 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 following BNF:

```
<spec>          ::= <axioms>
<axioms>        ::= <axiom> <axioms> | <axiom>
<axiom>         ::= <same-noalias> | <disj-noalias> | <same-alias>
<same-noalias> ::= forall nodes <var> , <var> . <re1> <> <var> . <re2>
<disj-noalias> ::= forall distinct nodes <var> and <var> ,
                        <var> . <re1> <> <var> . <re2>
<same-alias>    ::= forall nodes <var> , <var> . <re1> = <var> . <re2>
<var>           ::= IDENTIFIER
<re1>           ::= <re>
```

[2]Details on how access paths are collected, and how general dependence testing is then performed, can be found in [HHN93].

```
<re2>           ::= <re>
<re>            ::= <fields> | ( <re> ) | ( <re> ) <kleene> | <re> '|' <re>
<fields>        ::= <field> <fields> | <field>
<field>         ::= IDENTIFIER | _E_
<kleene>        ::= * | +
```

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. The `_E_` is used to denote the empty string ϵ.

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-noalias>` specifies that *forall* nodes p in DS, the set of nodes[3] reachable from p along any path in the set `<re1>` will be disjoint from the set of nodes reachable from p along any path in the set `<re2>`. The meaning of a `<disj-noalias>` axiom is defined similarly. However, a `<same-alias>` axiom specifies the opposite: $\forall$ nodes p in DS, the set of nodes reachable from p along any path in the set `<re1>` and the set of nodes reachable from p along any path in the set `<re2>` all in fact refer to the same node n. The motivation for this last form of axiom will become clear in Section 4.

2.3 Verification

Given the expressive power of ASAP, it is likely that a compiler will be unable 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.

Note that visualization tools can also help ensure that a given ASAP specification accurately reflects what the programmer had in mind, as can compiler support for predefined collections of axioms (e.g. **`left`** and **`right`** define a "tree") or a higher-level language such as ADDS.

3 Related Work

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

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

```
type TofTs_t [Down][Across][Sub] {
  Data_t    d;
  TofTs_t  *l, *r are uniquely forward along Down;
  TofTs_t  *n is uniquely forward along Across;
  TofTs_t  *p is backward along Across;
  TofTs_t  *s is uniquely forward along Sub;
} where Sub is independent;
```

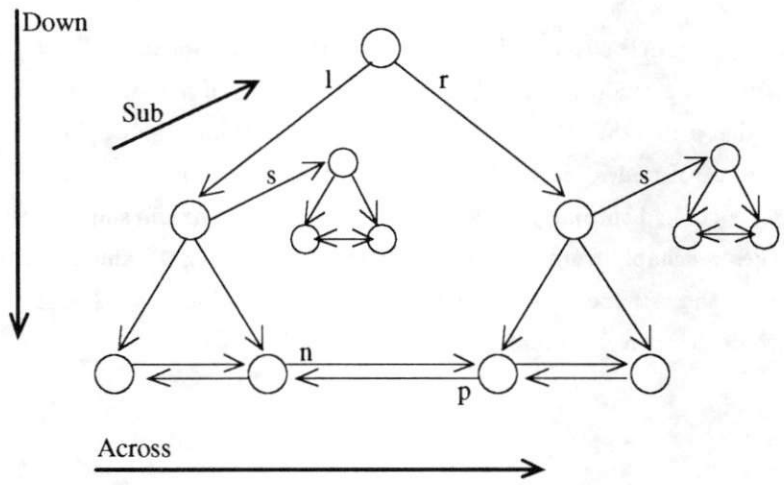

Figure 2: Tree of trees: ADDS declaration and example data structure.

3.1 ADDS

Our work on ASAP is a generalization of our earlier work on **ADDS** [HHN92a], a data structure description language based on a notion of *directions* and *dimensions*. In ADDS, pointer fields can be declared as traversing in one of five directions—**forward**, **uniquely forward**, **combined uniquely forward**, **backward**, and **unknown**—as well as along a programmer-defined dimension. Dimensions can be declared as either **independent**, or by default, **dependent**. A quick example should suffice to define the meaning of these terms; for more information, the reader is referred to [HHN92c].

Figure 2 illustrates a tree of trees, in which all trees are independent from one another. However, within a given tree, note that the leaves are linked together in a two-way fashion. An appropriate ADDS-based type declaration is also shown in Figure 2. The type **TofTs_t** is declared as having 3 dimensions: **Down**, **Across**, and **Sub**. The **Down** and **Across** dimensions are dependent since it is possible for a node to be reached by traversing along either dimension; the Sub dimension is independent, however, since any substructure reached by traversing along Sub is disjoint from all others. The direction forward declares acyclicness, uniquely forward declares that at most one pointer labeled with this field ever points to a node (implying a linked-list), and combined uniquely forward generalizes the notion of uniquely forward to hold for two or more fields (thereby implying a tree). Hence, looking at Figure 2, we see that the field n traverses uniquely forward along **Across**, s traverses uniquely forward along Sub, and l and r traverse combined uniquely forward along

Down. Finally, the field p is said to traverse backward along Across, given the cyclic relationship between n and p and the fact that n traverses forward.

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. For example, the fields l, r, and n of Figure 2 form an acyclic substructure, but such acyclicness cannot be conveyed by this (or any other) ADDS declaration. 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. So, 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 lead 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 specification (see Appendix A), 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 to 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[4]. 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 and thus commonly found in circuit simulations [Kun86]. We will discuss sparse matrices, and other data structures that graph types are unable to describe, in more detail in Section 4; these same structures will be described accurately using ASAP.

There are some cases, however, in which graph types can yield a more accurate description than ASAP. For example, with graphs types it is possible to accurately describe a post-order threaded tree; one can predict, at compile-time and for all nodes, exactly which node will be reached after traversal of the threading field. In ASAP, one can accurately describe the tree, but at best can only convey that the threading field may reach *some* node in the tree, not precisely which one. Though it is not clear for just how many data structures this holds, both languages can describe common structures such as one-way linked lists, two-way

[4] In fact, not even all deterministic relationships can be specified, see [KS93].

linked-lists, and binary trees equally well.

It should be noted that graph types typically involve more complicated regular expressions than does ASAP, which one would expect since the programmer must state exactly which node will be reached after a given traversal. A nice feature of graph types is that the compiler is able to guarantee the well-formedness of the regular-like expressions. However, the latter does not guarantee that the structure described is the same one the programmer had in mind (a problem shared by all languages, not just graph types). It should also be pointed out that the compiler is not guaranteeing the correctness of imperative-style code which might build this kind of data structure. Graph types are intended for functional languages, i.e. languages without assignment statements. This guarantees the treeness of the backbone, since without assignment one can only build trees. The compiler is thus responsible for generating the code to compute and maintain the values of the routing fields, which is possible (though potentially expensive) using the regular-like expressions from the graph type.

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 [HHN93]; 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.

```
type Tree_t {
  Data_t   d;
  Tree_t  *child;
  Tree_t  *sib;
  Tree_t  *parent;
};
```

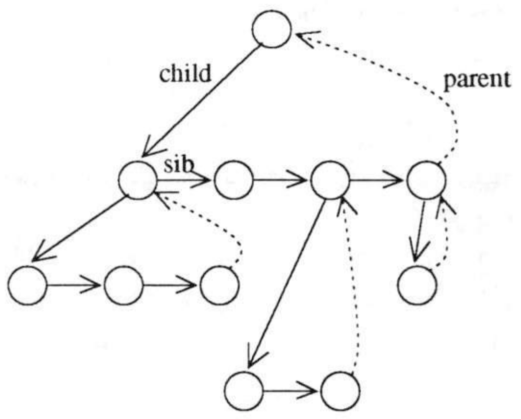

Figure 3: A general tree: type declaration and example data structure.

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. In this example a third field, **parent**, allows easy access to the parent from any child.

In ADDS parlance (see Section 3 for a quick overview of ADDS), a general tree would be declared with a single **Down** dimension, where the fields **child** and **sib** traverse **Down** in a combined uniquely forward direction (i.e. in a tree-like manner). The problem lies in describing the behavior of the **parent** field. Since this field does not form a direct cycle with each field traversing in the forward direction—the cycle actually involves both the **child** and **sib** fields—this cyclic relationship cannot be expressed using the ADDS backward keyword. Since this is the only available means in ADDS for accurately describing cycles, we conclude that ADDS cannot accurately describe this type of data structure.

A general tree can, however, be accurately described using ASAP. 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 nodes p, p.child <> p.sib.
```

Likewise, traversing these fields from distinct nodes will never lead to the same node. In fact, notice this holds for each field individually as well; traversing **child** and **child** from different nodes will not lead to the same node, as is the case for **sib** and **sib** traversals. Thus:

```
forall distinct nodes p and q, p.(child|sib) <> q.(child|sib).
```

Finally, we specify the acyclicness of the tree:

```
forall nodes p, p.(child|sib)+ <> p.
```

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 nodes p, p.parent <> p.(child|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 distinct nodes p and q, p.parent <> q.parent.
```

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

```
forall nodes p, p.(parent)+ <> p.
```

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 nodes p, p.child(sib)*parent = p.
```

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 a determent, 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[5]; thus there is no need to restate the obvious.

[5] 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;
};
```

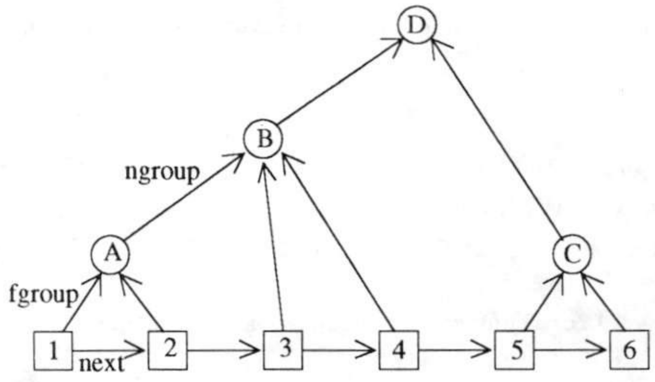

Figure 4: A union-find tree: type declaration and example data structure.

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[6]; note that this is not a tree in the typical sense, since the pointers flow from the "leaves" to the "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. One likely ADDS declaration would declare two dimensions `Across` and `Up`, where `next` traverses uniquely forward along `Across` and the group pointers `fgroup` and `ngroup` traverse forward along `Up`. The problem is that ADDS is unable to express the relationship between `fgroup` and `ngroup`—the only choices are tree (via combined uniquely forward) or graph (the default). The fact that these fields form a DAG cannot be expressed. Thus, given such an ADDS description, the compiler must conservatively assume that the relationship between `fgroup` and `ngroup` could be cyclic. Note that declaring additional ADDS dimensions will not help, since none of these dimensions will be independent, and dependent dimensions are likewise conservatively assumed to be cyclically related.

ASAP, however, can be used to generate an accurate description. We start by first considering the

[6]For simplicity, we show the elements in a linked-list form, even though a more efficient structure is needed by the find operation in order to locate elements quickly. Replacing the linked-list by a more efficient structure can easily be accommodated, and does not change the outcome of this discussion.

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 distinct nodes p and q, p.next <> q.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 nodes p, p.(next|fgroup|ngroup)+ <> p.
```

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, SWG91]. 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.

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

```
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;
};
```

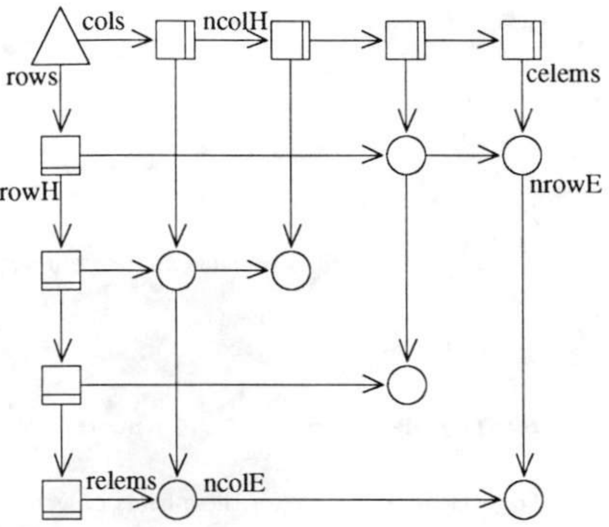

Figure 5: A sparse matrix: type declaration and example data structure.

state of the matrix at run-time. With ADDS, a sparse matrix can at least be partially described. The idea would be to first declare two dimensions, `Down` and `Across`. The field `ncolE` would traverse uniquely forward along `Across`, while `nrowE` would traverse uniquely forward along `Down`. These dimensions are dependent, since a node can be reached along either dimension. The problem is that such an ADDS declaration loses the important information that the row lists are disjoint (likewise for the column lists). This is a result of the fact that one must conservatively assume that the situation shown in Figure 6 is possible since the ADDS declaration does not explicitly prevent it; an element may have both `nrowE` and `ncolE` fields coming into its node (due to dependent dimensions), and an `ncolE` field can link two rows (dashed line) and still remain uniquely forward (at most one `ncolE` into any node). Thus, since a better ADDS declaration does not exist, ADDS is unable to accurately describe a sparse matrix, in particular the very property 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 (see Appendix A), 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:

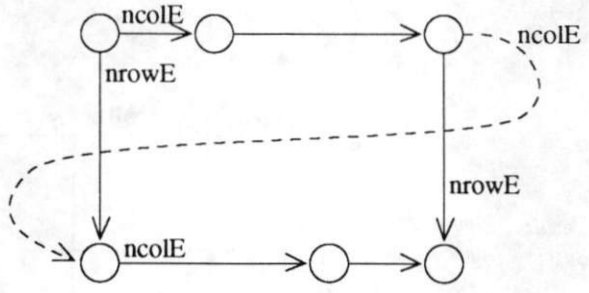

Figure 6: A possible aliasing allowed by ADDS declaration.

```
forall nodes p, p.nrowE <> p.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 distinct nodes p and q, p.nrowE <> q.nrowE.
forall distinct nodes p and q, p.ncolE <> q.ncolE.
```

Observe that these axioms do not prevent the situation shown in Figure 6. To correct this, we specifically need to state that rows of elements are disjoint, as are columns. This can be done as follows:

```
forall nodes p, p.(ncolE)* <> p.(nrowE)+(ncolE)*.
forall nodes p, p.(nrowE)* <> p.(ncolE)+(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 distinct nodes p and q, p.nrowH <> q.nrowH.
forall distinct nodes p and q, p.ncolH <> q.ncolH.
```

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

```
forall distinct nodes p and q, p.relems <> q.relems.
forall distinct nodes p and q, p.celems <> q.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 distinct nodes p and q, p.relems(ncolE)* <> q.relems(ncolE)*.
forall distinct nodes p and q, p.celems(nrowE)* <> q.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 distinct nodes p and q, p.rows <> q.nrowH.
forall distinct nodes p and q, p.cols <> q.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 nodes p, p.(rows|cols|relems|celems|nrowH|ncolH|nrowE|ncolE)+ <> p.
```

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 an 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.

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.

[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.

[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.

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

[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.

[HHN92c] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, 1(3):243–260, September 1992.

[HHN93] 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. Submitted for publication.

[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.

[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.

[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.

[McN93] D. McNamara. Pointer analysis in VAST-C, March 1993. Private communication. D. McNamara is Product Manager at Pacific-Sierra Research.

[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.

[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.

[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: ADDS to ASAP Translation

ADDS is a data structure description language based on a notion of *directions* and *dimensions* [HHN92a]. Given an ADDS-based type declaration, pointer fields can be declared as traversing in one of five directions: **forward**, **uniquely forward**, **combined uniquely forward**, **backward**, and **unknown**. If a pointer field is declared as traversing a direction, then it must also be declared as traversing a programmer-defined dimension; the exception is the **unknown** direction, in which a dimension may not be specified. If a pointer field is not annotated in any way, it is assumed to traverse in the **unknown** direction. Finally, dimensions can be declared as either **independent**, or by default, **dependent**.

The translation from an ADDS declaration to an ASAP specification proceeds as follows. Suppose $F1$, $F2$, ..., Fn represent distinct pointer fields and D denotes any programmer-defined dimension, then:

1. If Fi $(1 \leq i \leq n)$ is declared as traversing **forward** along D, then the following axiom holds (Fi is acyclic):

   ```
   forall nodes p, p.(Fi)+ <> p.
   ```

2. If Fi $(1 \leq i \leq n)$ is declared as traversing **uniquely forward** along D, then the following axioms hold (Fi is acyclic and at most only one Fi ever points to a given node):

   ```
   (a) forall nodes p, p.(Fi)+ <> p.
   (b) forall distinct nodes p and q, p.Fi <> q.Fi.
   ```

3. If Fi, ..., Fj $(1 \leq i < j \leq n)$ are declared as traversing **combined uniquely forward** along D, then the following axioms hold where f and g denote all distinct pairs of $Fi \ldots Fj$ ($Fi \ldots Fj$ form a tree):

   ```
   (a) forall nodes p, p.f <> p.g.
   (b) forall distinct nodes p and q, p.(Fi|...|Fj) <> q.(Fi|...|Fj).
   (c) forall nodes p, p.(Fi|...|Fj)+ <> p.
   ```

4. If Fi $(1 \leq i \leq n)$ traverses **backward** along D, then for all Fj, ..., Fk $(1 \leq j \leq k \leq n)$ which traverse in some **forward** direction along D, the following axioms hold (Fi is acyclic, and has a cyclic relationship with $Fj \ldots Fk$):

   ```
   (a) forall nodes p, p.(Fi)+ <> p.
   (b) forall nodes p, p.(Fj|...|Fk)Fi = p.
   ```

   Furthermore, if $j = k$ and Fj traverses **uniquely forward** along D, then the following additional axioms hold (Fi is uniquely backward, and cyclic relationship holds in both directions):

   ```
   (a) forall distinct nodes p and q, p.Fi <> q.Fi.
   (b) forall nodes p, p.FiFj = p.
   ```

5. If D is an **independent** dimension, where for all Fi, ..., Fj $(1 \leq i \leq j \leq n)$ which traverse forward along D and for all remaining fields Fk, ..., Fl $(1 \leq k \leq l \leq n)$ which traverse along other dimensions, the following axioms hold (traversing any field of an independent dimension separates a data structure into disjoint substructures):

   ```
   (a) forall nodes p, p.(Fi|...|Fj)(Fk|...|Fl)* <> p.(Fk|...|Fl)*.
   (b) forall distinct nodes p and q,
         p.(Fi|...|Fj)(Fk|...|Fl)* <> q.(Fi|...|Fj)(Fk|...|Fl)*.
   ```

Finally, if a field Fi is said to traverse in the **unknown** direction, no axioms are specified since Fi can potentially alias any node in the data structure. Likewise, if a dimension D is said to be **dependent**, no axioms are specified since the behavior of this dimension with respect to other dimensions is unknown.