Spring 2015

Recursive tree traversal dependence analysis

Yusheng Weijiang
Purdue University

Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Yusheng Weijiang

Entitled
Recursive Tree Traversal Dependence Analysis

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

MILIND KULKARNI

MITHUNA S. THOTTETHODI

VIJAY RAGHUNATHAN

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): MILIND KULKARNI

Approved by: Michael R. Melloch 04/21/2015

Head of the Department Graduate Program Date

RECURSIVE TREE TRAVERSAL DEPENDENCE ANALYSIS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Yusheng Weijiang

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2015

Purdue University

West Lafayette, Indiana

Dedicated to my parents.

ACKNOWLEDGMENTS

Many thanks to Dr. Milind Kulkarni for advising me throughout my research. He has made my graduate school experience here extremely valuable and helped me get through the process of research by taking things one step at a time. Also, thanks to my committee members, Dr. Mithuna Thottethodi and Dr. Vijay Raghunathan, who offered guidance and support.

I would also like to thank Youngjoon Jo, the person whose work my research is primarily built upon. He has helped me immensely in understanding and expanding on his past project to get me to where I am now.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

Point a data structure that represents a single traversal of a tree structure, often containing local variables used within the traversal

Block a set of points in point blocking, containing local information for each traversal

ABSTRACT

Weijiang, Yusheng M.S.E.C.E., Purdue University, May 2015. Recursive Tree Traversal Dependence Analysis. Major Professor: Milind Kulkarni.

While there has been much work done on analyzing and transforming regular programs that operate over linear arrays and dense matrices, comparatively little has been done to try to carry these optimizations over to programs that operate over heap-based data structures using pointers. Previous work has shown that *point blocking*, a technique similar to loop tiling in regular programs, can help increase the temporal locality of repeated tree traversals. Point blocking, however, has only been shown to work on tree traversals where each traversal is fully independent and would allow parallelization, greatly limiting the types of applications that this transformation could be applied to.

The purpose of this study is to develop a new framework for analyzing recursive methods that perform traversals over trees, called *tree dependence analysis*. This analysis translates dependence analysis techniques for regular programs to the irregular space, identifying the structure of dependences within a recursive method that traverses trees. In this study, a dependence test that exploits the dependence structure of such programs is developed, and is shown to be able to prove the legality of several locality- and parallelism-enhancing transformations, including point blocking. In addition, the analysis is extended with a novel path-dependent, conditional analysis to refine the dependence test and prove the legality of transformations for a wider range of algorithms. These analyses are then used to show that several common algorithms that manipulate trees recursively are amenable to several locality- and parallelism-enhancing transformations. This work shows that classical dependence analysis techniques, which have largely been confined to nested loops over array data

structures, can be extended and translated to work for complex, recursive programs that operate over pointer-based data structures.

1. INTRODUCTION

1.1 Problem Context

Over the past three decades, a tremendous number of *loop transformations*, such as loop interchange, fusion, and tiling, have been designed to improve locality in programs that use loop nests to manipulate arrays [1]. A number of powerful *dependence analysis* techniques and frameworks have been developed to determine when applying these various transformations to *regular* programs—array programs with affine loop bounds and index expressions—is legal [2–9]. While there have been many attempts to extend these transformations to handle more sophisticated programs, including those that have non-affine loop bounds and index expressions [10–12], these tools have largely been confined to the class of array programs using nested loops. However, these conditions are quite restrictive on the types of programs that can be optimized using these transformations, and many programs, notably ones that allocate data structures on the heap, can not be analyzed using these frameworks.

One such class of programs that prior analyses cannot handle are programs that perform traversals on pointer-based tree structures. Pointer-based tree structures are commonly used for many applications, such as the Barnes-Hut n-body simulation and binary search trees. These structures are often accessed through a series of recursive traversals, which is a pattern that admits a high degree of possible parallelism.

1.1.1 Point Blocking

In recent work, Jo and Kulkarni [13] developed an optimization called *point blocking* that performs loop tiling–like transformations not on nested loops, but instead on *repeated recursive traversals* of pointer-based tree structures. Point blocking works

by grouping together multiple traversals of a tree into a *block* and performing a single traversal of the tree. Within each block, each of the original traversals (called *points*), performs all required computation while visiting a particular node, then the whole block moves forward to the next node. In essence, the computations performed by multiple traversals are reordered to promote locality in the tree.

Unfortunately, while this transformation resembles loop tiling (see Section 3.2), existing dependence analyses cannot be applied, as point blocking targets pointer-based, recursive programs. Instead, Jo and Kulkarni establish the legality of their transformations through a simple, sufficient condition: their transformations can be applied when the traversals over the tree structure are independent of each other. This condition can be established using existing shape analysis techniques [14–16].

However, this sufficient condition misses many optimization opportunities. Consider inserting a set of points into a binary search tree, as shown in Figure 1.1(a). Point blocking can be correctly applied to the code, as shown in Figure 1.1(b), even though there is clearly a dependence from one traversal to the next, as each insertion changes the tree. The reason for this is that if multiple points in a block travel down the same path of the tree, and the first point in the block inserts a node into the tree, subsequent points in the block see the *new* node that was inserted, as they would have in the original code. This means that so long as the points reach the empty node in the same order, point blocking preserves the dependence. *This pattern of behavior is quite common*, arising in many top-down tree building algorithms. Handling such cases requires a more sophisticated notion of what kinds of dependences preclude point blocking.

1.2 Thesis Statement

1.2.1 Objectives

During this research, the main objective was to develop a *tree dependence analysis* with which to represent and analyze the dependences within a pointer-based

```
tree = /* bst */
vals = /* ints > 0 */
foreach (i in vals)
  recurse(tree, i)

recurse(n, i)
  if (n.val = -1)
    n.val = i; return;
  if (n.val < i)
    if (n.l = null)
      n.l = new node;
      n.l.val = -1;
    recurse(n.l, i)
  else
    if (n.r = null)
      n.r = new node;
      n.r.val = -1;
    recurse(n.r, i)
```

(a) BST insertion code

```
foreach (i[] in vals)
  recurse(tree, i[])

recurse(n, i[])
  // l[], r[] = arrays
  if i.size = 0
    return;
  foreach (j in i)
    if (n.val = -1)
      n.val = j; continue;
    if (n.val < j)
      if (n.l = null)
        n.l = new node;
        n.l.val = -1;
      l[].append(j);
    else
      if (n.r = null)
        n.r = new node;
        n.r.val = -1;
      r[].append(j);

  recurse(n.l, l[]);
  recurse(n.r, r[]);
```

(b) Blocked BST code

Fig. 1.1.: BST insertion, unblocked and blocked

tree program. Analogously to array dependence analyses, which allow complex loop transformations to be performed even if there are loop-carried dependences, a tree dependence analysis must provide enough information to allow restructuring transformations like point blocking to be performed even in the presence of dependences between traversals. This work serves to extend many array dependence and optimization techniques to the realm of pointer-based data structures.

1.2.2 Procedures

In creating the analysis framework, a step-by-step development approach was followed. The following tasks were accomplished as part of the research:

- Creating a novel dependence test that can prove the legality of point blocking even in the face of complex dependences (Section 4), and a proof of the soundness of point blocking under this test.
- Creating an analysis that applies this dependence test to tree-traversal programs (Section 6), particularly one that reveals the *structure* of the dependences with respect to the control flow of the program.
- Refining the dependence analysis using *path conditions* to prove that certain dependences that appear to exist can never arise during an execution (Section 7).
- Performing experimental evaluation showing that this analysis enables significant performance improvements from three different transformations: point blocking, *traversal splicing* [17], and a transformation that automatically derives parallel tree construction implementations from their sequential specification.

These tests are then used to prove the legality of point blocking for numerous examples, including a complex oct-tree building algorithm extracted from Barnes-Hut (Section 8).

2. LITERATURE REVIEW

There exists a lot of work on both program logics for heap data structures as well as similar analyses and transformations for regular programs. Some works that focus on analyses and transformations for both regular programs and irregular programs will be discussed.

2.1 Analysis for Regular Programs

The past two decades have seen a lot of work done on analyzing and transforming regular programs. These programs operate over dense matrices and arrays using *affine* subscripts. By analyzing the patterns of dependences in loops that operate over matrices and arrays, it is possible to find effective transformations for these programs that give locality benefits and allow parallelization [1]. However, due to these analyses relying on affine subscripts to determine dependence information, they cannot be directly applied to irregular programs. One improvement on these restrictive properties is to use constraints containing *uninterpreted function symbols* to represent non-linear expressions [11]. While this helps improve the overly conservative model of analyzing only affine subscripts, it is still aimed at regular loop algorithms, and not recursive heap traversals.

Loop chaining is an abstraction of regular loops in order to group together loops that share data as a chain [18]. This leads to being able to find subsets of loops that can be executed in parallel to increase data reuse and locality while limiting the amount of communication that needs to happen. [19] extends this idea to work on loops where dependences are caused by indirect references, using a *full sparse tiling* algorithm with loop chaining. Applying tiling to sparse matrix approaches is another way of trying to apply regular loop transformations to programs beyond

dense matrices [20], but this requires either run-time information about the program, or in the loop chaining case, programmers need to include a data access specification. This specification is used to express data dependences in the program, removing the need for separate dependence analysis, but this forces programmers to express the required dependence information themselves.

The detection and transformation of regular loop computations to improve data reuse has recently been used to optimize stencil computations [21]. Stencil computations are a class of computation pattern where weighted sums of values at a set of neighboring points are computed over a grid. For higher-order stencil computations, there is a lot of arithmetic computation done over a small data set. With no computation reordering, stencils exhibit many of the same issues with data reuse as tree traversals do. However, high dimension stencils tend to slow down due to poor register usage, while tree traversals may operate inefficiently due to poor cache usage.

Polyhedral frameworks have been previously used as an abstraction to transform regular programs by analyzing their iteration space, allowing the transformations to be free from the original loop structure [22]. However, these types of analyses are very conservative when dealing with non-affine loop bounds or subscripts. There has been work done on taking these types of polyhedral frameworks and applying them to programs that use non-affine loop bounds or subscripts [12]. This allows iteration spaces to be defined for non-affine loop bounds, enabling non-affine transformations to be done on them. The analysis, however, requires run-time inspection and cannot be done during compile time.

2.2 Analysis for Irregular Programs

In order to verify that the proposed transformations work for irregular programs, analysis must be done on the shape of the programs. *Shape analysis* has been used to verify particular program properties in the past [16]. There has also been prior work on parallelizing programs based on *shape analysis* information to determine

what kind of irregular data structure is used (a tree, DAG, or arbitrary graph) [23]. Unfortunately, these analyses do not give information on where data accesses intersect within a traversal, which is needed to perform transformations on code.

To apply transformations like point blocking, any possibly conflicting access paths in a recursive method must be verified to never occur based on path invalidation. Recent work has laid out a method for recursively proving properties of inductive trees [24]. This uses an extension of first-order logic with recursive definitions called DRYAD. DRYAD allows finding simple recursive proofs of properties using formula abstraction and SMT solvers. In addition, ways to find conflicts in accesses of data structures have been laid out in [10, 15]. However, these two are aimed at being able to tell that there is a dependence in the data structure accesses, and don't go into more detail with regards to conflicts coming from different nodes in a tree.

Previous work has used *attribute grammars* [25] in order to create and tune parallel tree traversals [26]. This allows a program to be declaratively specified as an attribute grammar, then synthesized into a set of traversals. Various ways to evaluate parallel attributes are discussed in [27]. Attribute grammars make the dependences that we want to focus on explicit. Using this information, it is possible to perform transformations that fuse multiple traversals. Because the dependences in accesses are explicit, they are easier to work with. Attribute grammars are however a restrictive programming model, which means more work must be done in order to gain the benefits of these kinds of transformations.

3. BACKGROUND AND MOTIVATION

This chapter covers the necessary background material necessary for the thesis. It first discusses the theory of loop transformations for array programs, specifically loop interchange, which enables loop tiling. Then it summarizes recent work by Jo and Kulkarni that develops analogous tiling transformations for trees. This discussion lays the foundation for Section 4, which defines a dependence test for tree programs.

3.1 Loop transformations for array programs

For the past three decades, there has been substantial interest in determining the structure of dependences in programs that manipulate arrays by looping over them, so that locality-enhancing restructuring transformations can be applied. Such programs are common in scientific computing, where many linear algebra and stencil routines are most naturally formulated as array programs. Moreover, because the arrays that these programs manipulate often enjoy substantial reuse (consider matrix multiplication, which performs $O(n^3)$ computation over $O(n^2)$ data), there are fruitful opportunities for transformations of these programs to improve locality by bringing uses of the same piece of data closer together in time [7]. Perhaps the most popular locality-enhancing transformation for loops over arrays is *loop tiling*, which transforms a double-nested loop into a triple- (or quadruple-) nested loop [5] , as in the following abstract example:

```
for (i := 0; i < N; i++)
  for (j := 0; j < N; j++)
    A[f_1(i)][f_2(j)] = ...; ... = A[g_1(i)][g_2(j)]
}
```

Becomes:

```
for (ii := 0; ii < N; ii += B)
  for (j := 0; j < N; j++)
    for (i := ii; i < ii + B; i++)
      A[f_1(i)][f_2(j)] = ...; ... = A[g_1(i)][g_2(j)]
}
```

Research in loop transformations has largely concerned itself with whether loop tiling is legal and profitable [3,7,9]. For this transformation to work at all, it must be legal. The legality of tiling boils down to whether *loop interchange* is legal [8]; if the inner and outer loop of the above example can be swapped, then loop tiling is legal.

Determining whether loop interchange is legal requires understanding how interchange affects the behavior of the loop. Conceptually, loop interchange is a *rescheduling* of the loop iterations. The original loop consists of an *iteration space*—dynamic instances of the loop body, each with a different value of i and j—that is totally ordered: $(i_1, j_1) \prec (i_2, j_2) \Leftrightarrow (i_1 < i_2) \vee ((i_1 = i_2) \wedge (j_1 < j_2))$. Loop interchange moves the j loop to the outside, producing a different total ordering of the same iteration space: $(i_1, j_1) \prec (i_2, j_2) \Leftrightarrow (j_1 < j_2) \vee ((j_1 = j_2) \wedge (i_1 < i_2))$.

When is this rescheduling legal? Answering this question requires understanding the dependence structure of the loop [2]. If, in the original schedule, one iteration of the loop, (i_1, j_1), writes to a location that a later iteration, (i_2, j_2) reads from, the new schedule must not exchange the order of these two iterations, which would result in the second iteration reading the wrong value. Clearly, if there are no dependences between different loop iterations, interchange is legal. However, even if dependences exist, they might not be affected by the transformation. For example, suppose there were a dependence in the original schedule between the pairs of iterations (i, j) and $(i + 1, j + 1)$. Even in the interchanged loop, the (i, j) iteration will precede the

$(i+1, j+1)$ iteration, preserving the dependence. The following *dependence test* captures the conditions under which loop interchange is legal.[1]

$$\begin{aligned} \nexists\, i_1, i_2, j_1, j_2 \;.\; f_1(i_1) = g_1(i_2) \wedge f_2(j_1) = g_2(j_2) \wedge \\ (i_1 < i_2 \wedge j_1 > j_2) \end{aligned} \tag{3.1}$$

The first line of this test captures whether a pair of iterations access the same location, while the second line of the test captures whether those iterations will execute in a different order after interchange.

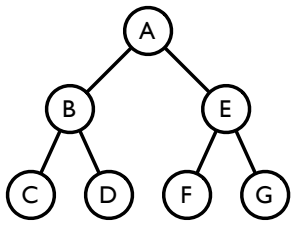
Sophisticated dependence analyses such as the Omega test [6] and compilers such as PLuTo [4] use integer linear programming–based techniques to prove that interchange is legal. These analyses rely on the fact that in most array programs, the indexing expressions f_1, f_2, g_1, and g_2 are affine, and hence amenable to ILP. As a result, a long standing open problem has been whether similar tiling techniques exist for non-affine, non-loop-based programs, and how to prove the legality of these techniques.

3.2 Loop transformations for trees

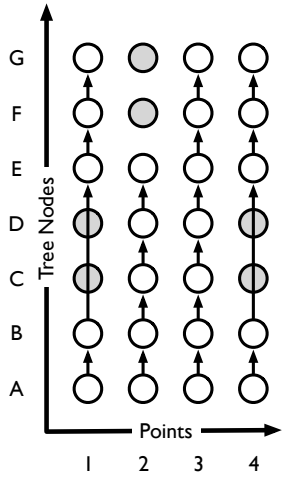
In recent work, Jo and Kulkarni [13] developed a locality-enhancing transformation called *point blocking* for programs that repeatedly traverse tree data structures. Figure 3.1(c) shows abstracted pseudocode capturing the general structure of these algorithms. As each point traverses the same tree, there is data reuse in the algorithm, and an opportunity to exploit locality if multiple points' operations on the same data can be brought closer together.

Point blocking exploits locality by grouping multiple points into blocks and moving the blocks through the trees in lockstep [13]. Figure 3.1(e) shows this transformed code. Instead of the recursive method operating on a single point, it operates on *blocks* of points. After each point in the block interacts with a particular node, those

[1] In a full dependence test, there are additional constraints to ensure that both iterations fall within the bounds of the loop nest; these constraints are ignored for simplicity.



(a) Example tree for traversal

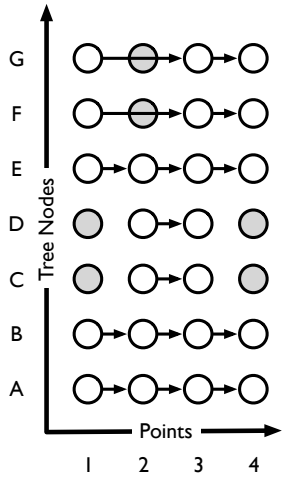


(b) Iteration space before point blocking

```
tree = /* tree root */
points = /* points */
foreach (p in points)
  recurse(tree, p)

recurse(n, p)
  if truncate?(n, p)
    return;
  if isleaf?(n)
    return;
  /* do work */
  recurse(n.left, p)
  recurse(n.right, p)
```

(c) Pseudocode for traversal



(d) Iteration space after point blocking

```
tree = /* tree root */
points = /* points */

//bp is block of points
foreach (bp in points)
  recurse(tree, bp)

recurse(n, bp)
  if isempty?(bp)
    return;
  foreach (p in bp)
    if truncate?(n, p)
      continue;
    if isleaf?(n)
      continue;
    /* do work */
    //add to next block
    nb.add(p)

  recurse(n.left, nb)
  recurse(n.right, nb)
```

(e) Blocked traversal

Fig. 3.1.: Point blocking

points that want to continue traversal are added to a "next" block, which continues down the tree; when the points finish visiting the subtree, all points resume their traversal. If a block is ever empty, that means no points want to visit a particular node (or subtree), so the traversal is truncated. In other words, a group of points are placed into a block, and the block traverses the tree, visiting all nodes in the tree that any point in the block would have visited; at each tree node, any points in the block that would have interacted with the tree node in the original code do so in the original order.

The key insight behind point blocking is that the tree-traversal algorithm can be abstracted as a loop nest, with the point loop as the outer loop and the recursive traversal as the inner "loop." Each "iteration" in this abstraction consists of the recursive method body being executed by a particular point at a particular node of the tree; the recursion and pointer-chasing merely serve to determine the order in which the nodes are visited.

Figure 3.1(b) shows an example iteration space and total order for a series of recursive traversals of the tree shown in Figure 3.1(a). The x-axis represents the points that traverse the tree, while the y-axis represents the nodes visited by the point. Note that some of the iterations are greyed out, and the traversal skips past them. A traversal may not visit the entire tree—it may be truncated and skip visiting a subtree. This is the source of "irregularity" in tree programs; array programs have more "regular," predictable iteration spaces.

Given this iteration space abstraction, Jo and Kulkarni describe a "loop interchange" transformation, with the total order shown in Figure 3.1(d). This has an analogous reordering effect as loop interchange in the regular iteration spaces produced by array programs; in the interchanged code, every point visits a particular node in the tree before moving on to the next node in the tree. Point blocking is a combination of strip mining the point loop (breaking the point loop into a series of smaller loops that operate over subsets of points) and then interchanging the in-

ner point loop with the traversal loop. This is directly analogous to strip mining + interchange, a common technique for tiling array programs [8].

3.2.1 "Multi callset" traversals

In the examples of Figure 3.1, every point traverses the tree in the same order—the only differences between traversals arise because points may skip entire subtrees during their traversal. In other words, there is a single linearization of the nodes of the tree, and each point's traversal is some subsequence of that linearization. Hence, when points are placed into a block, the order that the block traverses the tree is the same as the traversal orders of any of the individual points. These are known as "single callset" traversals. However, some algorithms, such as nearest neighbor, have point-dependent traversal orders, where different points traverse the tree in different orders; these are known as "multi callset" traversals.

In this work, only single call set traversal algorithms are addressed, as they are the only ones that admit a sophisticated dependence test. Multi callset algorithms can still be analyzed using a test for independence, but a thorough dependence test for these algorithms cannot be done. Section 4.3.1 elucidates the reasons why there can be no good dependence test for multicallset traversals.

4. POINT BLOCKING LEGALITY

This chapter lays out a dependence test for point blocking, analogous to the dependence test for array programs in Equation 3.1. For brevity, "iteration" is used to refer to the operation(s) performed by a single point at a single tree node. As previously discussed, all traversals are assumed to be single callset.

4.1 A conservative approach

As described in the introduction, Jo and Kulkarni set forth conservative criteria for point blocking legality [13]. In a repeated tree traversal, if there are no dependences between iterations at all, then any reordering of iterations (including the one imposed by point blocking) would be legal. However, in all of the applications Jo and Kulkarni examined, tree traversals are performed to compute *reductions* over the tree: as a point traverses the tree, it accumulates some value (a force, a correlation count, its nearest neighbor, etc.), often in non-commutative ways. Hence, there are clearly dependences between iterations that point "up" in the iteration space.

Jo and Kulkarni [13] noted that despite the rescheduling imposed by point blocking, each point still traversed the tree in the same order as before. Hence, any dependences carried over the "traversal loop" but not over the "point loop" would be preserved. Thus, they applied point blocking whenever the enclosing point loop was parallelizable, ensuring that any dependences were only carried across the traversal loop. However, this criterion is too conservative. Not all point loop–carried dependences are violated by point blocking, as in the BST-insertion example from Figure 1.1. Note that although it appears that different "points" traverse the BST differently, because each point only traverses from the root of the tree to a leaf, each traversal is still a subsequence of a single linearized traversal, meaning this is a single

callset algorithm. Point blocking can be correctly applied to the code, as shown in Figure 1.1(b), even though there is clearly a point loop–carried dependence. The reason for this is that if multiple points in a block travel down the same path of the tree, and the first point in the block inserts a node into the tree, subsequent points in the block see then *new* node that was inserted, as they would have in the original code. The loop-carried dependence is preserved! *This pattern of behavior is quite common*, arising in top-down tree building algorithms for building kd-trees and Barnes-Hut octtrees. Handling such cases requires a more sophisticated notion of what kinds of dependences preclude point blocking.

4.2 A dependence test for point blocking

To develop a more accurate dependence test for tree codes, consider the two clauses of the dependence test for array programs in Equation 3.1. The first clause picks out the existence of iterations that have a dependence. If only that clause were in the dependence test, then any loop-carried dependence would preclude loop interchange. It is the second clause of the test (on the second line) that provides the precision: a loop carried dependence is only a problem if the second iteration (such as the (i_2, j_2) iteration) encounters the dependence earlier in the j loop than the first iteration. This situation means that when the j loop is on the outside, what used to be the second iteration will actually execute earlier.

The iteration space diagrams of Figures 3.1(b) and 3.1(d) give us some insight into what an analogous dependence test for point blocking might look like. Each "iteration" in a traversal code is identified by a point/node pair: (p, n). Suppose there is a dependence between the traversal executed by point p_1 and a later point p_2: p_1 accesses a location in the tree when it is visiting node n_1, and p_2 accesses the same location in the tree when it is visiting node n_2, with at least one of these accesses being a write. This dependence is preserved by point blocking if n_2 is the same as n_1

(both points are at the same node when the dependence occurs) or n_2 is later in the traversal order than n_1.

To formalize this dependence test, we label each statement that reads or writes a location in the recursive method body as $s_1, s_2, \ldots$. Because the particular location read or written by a statement depends on where in the tree the recursive method is (consider that the code in Figure 1.1(a) reads from nodes in the tree, but the particular nodes being read depend on the argument `n`), the location being accessed by statement i during iteration (p, n) is specified as $s_i(p, n)$.

Making a recursive call requires accessing the arguments to the recursive call. Because point blocking defers making recursive calls until after all points in the block execute their method body, it makes sense to treat the read(s) performed as part of the method invocation as part of the *next* iteration performed by the point. This is easily handled by assuming there are dummy statements at the beginning of the method body that read the arguments to the method.

Two dynamic statements, $s_i(p_i, n_i)$ and $s_j(p_j, n_j)$ *interfere* (written $s_i(p_i, n_i) \bowtie s_j(p_j, n_j)$) when they access the same location and one of the statements is a write.

Interference between two dynamic statements, is defined as follows:

$$s_i(p_i, n_i) \bowtie s_j(p_j, n_j) \equiv \\ s_i(p_i, n_i) = s_j(p_j, n_j) \wedge \text{one of the statements is a write}$$

Note that just because a statement exists in a recursive method body does not mean that every point will execute that statement at every node of its traversal. Thus, an execution-based interference operator is defined, $\bowtie_e$, which adds the condition that statement s_i executes when point p_i is visiting node n_i.

From here, a dependence test determining whether or not point blocking is legal can be created; note the similarity to Equation 3.1:

$$\nexists\, p_i, p_j, n_i, n_j, s_i, s_j \;.\; s_i(p_i, n_i) \bowtie_e s_j(p_j, n_j) \wedge \\ (p_i \prec p_j \wedge n_i \succ n_j) \tag{4.1}$$

Theorem 4.2.1 *If Equation 4.1 is satisfied for a recursive traversal program, then applying point blocking to the program will not break any dependences.*

Proof To prove this, proceed by contrapositive: assume that applying point blocking to the program breaks dependences, and then show that the dependence test must be violated.

For a dependence to be broken, a dependence must first exist. Hence, let (p_i, n_i) and (p_j, n_j) be the two dependent iterations, with $(p_i, n_i) \prec (p_j, n_j)$. Therefore, $s_i(p_i, n_i) \bowtie_e s_j(p_j, n_j)$. In the original program, a point's traversal is completed before moving on to the next point. Hence, $p_i \prec p_j$. Note that if, after applying point blocking, p_i and p_j are placed in different blocks, this dependence will not be broken: the earlier block will complete its traversal before the later block starts, preserving the ordering of the iterations. This means p_i and p_j must be in the same block. Further, for the dependence to be violated, $(p_j, n_j) \prec (p_i, n_i)$ must be true after applying point blocking.

There are three possible cases for the ordering of n_i and n_j:

$n_i \prec n_j$: In this case, n_i appears before n_j in the original program's traversal order. Recall that the block traverses the tree in the same order as the original points would have. Hence, the block will visit n_i before it visits n_j in the transformed code, preserving the dependence.

$n_i = n_j$: In this case, the points access the same location when they are at the same node in the tree. In the point blocked code, each point in a block executes its entire method body before moving on to the next point, so p_i performs its access before p_j, preserving the dependence.

$n_i \succ n_j$: In this case, n_j precedes n_i in the traversal order, so the block will visit n_j before it visits n_i, and (p_j, n_j) will occur before (p_i, n_i), violating the dependence.

Since the dependence is assumed to be violated, the third case must obtain. Hence, there are two iterations, (p_i, n_i) and (p_j, n_j), and two statements s_i and s_j such that: $s_i(p_i, n_i) \bowtie_e s_j(p_j, n_j)$, $p_i \prec p_j$ and $n_i \succ n_j$, violating the dependence test. ■

4.2.1 DAG traversals

Point blocking can be applicable not only to traversals of trees, but to traversals of any recursive data structure, including DAGs and general graphs [13]. Note that the dependence test in Equation 4.1 is still valid for traversals of non-tree data structures. However, for DAGs and general graphs, the same node may be visited by a traversal more than once, so the $\succ$ relation between nodes in a traversal no longer obeys any sort of order. Because of the difficulty of determining the relation between two nodes in a DAG or graph traversal, if these analyses encounter a traversal of a data structure that cannot be proven to be a tree, then only Jo and Kulkarni's independence test for legality may be applied.

4.3 Simplified dependence tests

The dependence test of Equation 4.1 is difficult to apply. First, it may be hard to tell exactly when a statement might execute, due to complex, data-dependent control flow in the method body—not to mention that whether a particular iteration executes in the first place often depends on the structure of the tree, which is also input-dependent. Second, telling whether one node of the tree precedes another in the traversal order can also be tricky. Note, however, that it is possible to simplify the dependence test in various ways while preserving soundness, as long as the resulting dependence test is at least as strong. In particular, the following dependence test is stronger than that of Equation 4.1:

$$\begin{aligned} \forall\, p_i, p_j, n_i, n_j \;.\; & (p_i \prec p_j) \rightarrow \\ & (\exists s_i, s_j \;.\; s_i(p_i, n_i) \bowtie_* s_j(p_j, n_j)) \rightarrow \\ & (n_i \preceq_a n_j) \end{aligned} \tag{4.2}$$

where $\bowtie_*$ represents any interference test weaker than $\bowtie_e$, and $n_i \preceq_a n_j$ is the ancestry relationship, and is true iff $n_i = n_j$ or n_j is a descendant of n_i in the tree. Restated, the dependence test says that the transformation is safe when, for all iterations which are from two different points' traversals, if the two iterations interfere, the node where the earlier point's iteration occurs is an ancestor of the node where the later point's iteration occurs.

4.3.1 Aside: Multi callset point blocking

While we have focused on single callset applications, where all points' traversals are consistent with some canonical traversal order, Jo and Kulkarni developed versions of point blocking that apply to multi callset algorithms. In multi callset algorithms, there are multiple possible orders that a point could visit a node's children. For example, in a nearest neighbor search, points select an order to traverse the tree based on which subtree is more likely to contain the nearest neighbor. Thus, when two points reach a particular node in the tree, one point might visit the left child before the right, and the other might take the opposite order.

To handle such situations, Jo and Kulkarni's method creates a separate "next" block for each possible order of visiting a node's children. When processing a block of points, each point is added to the next block associated with the traversal order it chooses. Then, the two blocks are processed in sequence. The block of points visiting the right child before the left executes, followed by the block of points visiting the left child before the right.

The key issue here is that with multiple possible traversal orders, it is no longer possible to determine which points will visit which nodes first. In particular, consider two dependent iterations, (p_1, n_1) and (p_2, n_2), where point p_1 performs its traversal before point p_2. In the point blocked code, which iteration is executed before the other depends on the particular traversals taken by p_1 and p_2, which is often statically unknowable. In contrast, in the single callset case, the ordering only depends on

where n_1 and n_2 are in the tree. Hence, multi callset algorithms can only safely be transformed if there are *no* dependences between traversals. Because of this, only single callset algorithms are analyzed.

5. A SIMPLE LANGUAGE FOR TREE TRAVERSALS

To help formalize the discussion of our tree dependence analysis, a simple language is used to write recursive tree traversal algorithms. Because the analysis concerns itself with the behavior of the recursive method itself, rather than the code that invokes the method (much as array dependence analyses primarily concern themselves with the body of the loop in question, rather than the surrounding code), the language is used to describe the body of a recursive method that traverses a tree, with a signature shown in Figure 5.1. The recursive method is invoked from a frame program shown in Figure 5.2, which repeatedly traverses the tree for each of a (fixed, finite) set of points. Note that the method takes two arguments: **root** represents the current tree node being accessed by the method, while **point** represents the "point" performing the current traversal.

The points that traverse the tree and the nodes that constitute the tree are structures, each consisting of a number of fields. Tree node structures have one or more primitive fields, $f_p \in \mathbf{F}_p$ (holding values at each tree node), and one or more recursive fields, $f_r \in \mathbf{F}_r$ (references to their children in the tree), while point structures only have primitive fields. Figure 5.3 defines these structures for a program.

5.1 Syntax and assumptions

Figure 5.4 describes the syntax of recursive methods that traverse trees. Node references are local variables that can point to different nodes in the tree. There is a distinguished node reference, **root**, which names the reference passed in to the recursive method. Finally, there is a distinguished variable, **point**, that refers to the particular point structure passed in to the recursive method. For a given traversal of

```
void recurse (root, point)
```

Fig. 5.1.: Recursive method signature

```
Node tree = /* root of tree structure */
Set<Point> points = /* point set */
foreach (Point p : p)
  recurse(tree, p);
```

Fig. 5.2.: Frame program

$$f_p \in \mathbf{F}_p \text{ (Primitive fields)} \qquad f_r \in \mathbf{F}_r \text{ (Recursive fields)}$$

$$\text{Point structure: } \{f_p^*\} \qquad \text{Node structure: } \{f_p^* \;\; f_r^*\}$$

Fig. 5.3.: Node and point structures

the tree, this point reference is fixed—the same reference is passed to each recursive call.

Note that a few features simplify reasoning about the behavior of these algorithms. First, there are no loops in the method bodies. While some programs (such as Barnes-Hut) may loop over the children of a node, these loops can be statically unrolled to straight-line code. Second, once a path through the method body reaches the recursive calls (c), it performs one or more recursive calls then returns, ensuring that all tree traversals are pre-order.

The only means of manipulating the tree structure in a recursive method is by nullifying a subtree (by setting a recursive field to **null**), or by creating a new subtree (by setting a recursive field to point to a new tree node using **alloc**). Hence, if the traversal is called on a tree, after the traversal completes the resulting structure remains a tree. Proving that the initial structure is a tree can be done through shape analysis techniques Assume that programs never dereference **null** fields, and

$$f_p \in \mathbf{F}_p \text{ (Primitive fields)} \qquad f_r \in \mathbf{F}_r \text{ (Recursive fields)}$$

$$\text{Point structure: } \{f_p^*\} \qquad \text{Node structure: } \{f_p^* \;\; f_r^*\}$$

$$v \in \mathit{Values} ::= \mathbb{Z} \qquad l \in \mathit{Locations} ::= \mathbb{L} \cup \textsf{null}$$

$$n \in \mathit{NodeRefs} ::= \mathbf{root} \;\mid\; n_1 \;\mid\; n_2 \;\mid\; \ldots$$

$$\oplus ::= + \;\mid\; - \;\mid\; \times \;\mid\; \div$$

$$\odot ::= < \mid > \mid = \mid \neq \mid \geq \mid \leq$$

$$\begin{aligned}
s \in \mathit{Stmts} ::= {} & \textsf{skip} \;\mid\; \textsf{return} \;\mid\; s; s \;\mid\; c; \textsf{return} \\
& \mid\; \textsf{if}\ \mathit{bexp}\ \textsf{then}\ s\ \textsf{else}\ s \\
& \mid\; n := n \;\mid\; n := n.f_r \;\mid\; n.f_r := \textsf{null} \;\mid\; n.f_r := \textsf{alloc} \\
& \mid\; n.f_p := e \;\mid\; \mathbf{point}.f_p := e \\
c \in \mathit{Calls} ::= {} & \mathbf{recurse}\ (\mathbf{root}.f_r, \mathbf{point}) \;\mid\; c; c \\
e \in \mathit{Exprs} ::= {} & n.f_p \;\mid\; \mathbf{point}.f_p \;\mid\; e \oplus e \;\mid\; v \\
\mathit{bexp} \in \mathit{BExprs} ::= {} & n.f_r = \textsf{null} \;\mid\; n.f_r \neq \textsf{null} \;\mid\; e \odot e \\
p \in \mathit{Body} ::= {} & s; \textsf{return}
\end{aligned}$$

Fig. 5.4.: Language for defining recursive tree traversals

that programs initialize all fields of newly-allocated tree nodes before accessing them. In addition, any local variable or node reference is only defined once along any path through the program.

Finally, assume that the recursive method bodies are single callset (see Section 3.2), ensuring a single, canonical traversal order. More formally, each straight-line sequence of recursive calls that occurs in the recursive method body induces a partial order on the recursive fields of **root**. If all of those partial orders are consistent with each other, the program is single callset. [1]

Example programs Figure 5.5 shows how a quadtree traversal that occasionally updates a value at a node can be expressed in the simple language. Figure 5.6 shows how the BST insertion example from Figure 1.1 can be expressed.

5.2 Concrete semantics

The semantics for programs written in this language are defined in terms of the semantics of a particular tree traversal (the semantics of a single iteration of the frame program's loop). A traversal operates over a heap, h, that contains a set of cells representing tree nodes. Each tree node's primitive fields map to values, while its recursive fields map to other heap locations or **null**. A subset of the tree nodes are linked together through their recursive fields to form a tree rooted at **tree** in the frame program. The heap also contains a finite set of point structures.

During the execution of a traversal, a store σ maps references (including **root** and **point**) to heap locations. The program state contains a return value, ρ, that tracks whether the method is supposed to return. Hence, the evaluation relation for statements and calls is: $\langle s,\ \sigma,\ h,\ \rho\rangle \rightarrow \langle \sigma',\ h',\ \rho'\rangle$ and the evaluation relation for expressions is: $\langle s,\ \sigma,\ h\rangle \rightarrow v$.

[1] In the special case where the call sequences access *disjoint* sets of recursive fields, point blocking can be applied directly as presented in Jo and Kulkarni [13]. If the sequences are not disjoint, point blocking can still be applied, but Jo and Kulkarni's method will not preserve point ordering.

The formal semantics can be found in Appendix A. These semantics are straightforward, with variable uses and definitions looking up heap locations in the store and changing the mapping, respectively, and field dereferences of points and nodes accessing the heap as expected. The only non-standard aspect is the use of ρ: once an execution path encounters **return**, ρ is set to **T**, and subsequent statements along the path do not modify the store or heap.

The state at the beginning of a traversal is determined by the invocation of *recurse* by the frame program: $\langle p,\ \sigma[\mathbf{root} \mapsto \mathbf{tree}, \mathbf{point} \mapsto p],\ h,\ \mathsf{F}\rangle$, where p is a reference to the current point performing the traversal, and **root** starts out mapped to **tree**, the root of the tree structure (which resides in the heap). Assume that the tree structure has been correctly initialized prior to beginning traversal. All other local variables are initialized to 0 or **null** as appropriate.

```
1. root.v := root.v + 1;
2. if point.v = root.v
3.   return
4. else skip
5. if root.leaf = 1
6.   return
7. else skip
8. recurse (root.c1, point); recurse (root.c2, point);
9. recurse (root.c3, point); recurse (root.c4, point); return
```

Fig. 5.5.: Recursive method body for quadtree traversal

```
1. if root.v = −1
2.   root.v := point.v;  return
3. else
4.   if root.v < point.v
5.     if root.l = null
6.       root.l = alloc;  n1 := root.l;  n1.v := −1
7.     else skip
8.     recurse (root.l, point); return
9.   else
10.     if root.r = null
11.       root.r = alloc;  n1 := root.r;  n1.v := −1
12.     else skip
13.     recurse (root.r, point); return
```

Fig. 5.6.: Recursive method body for BST insertion

6. PATH-INSENSITIVE DEPENDENCE ANALYSIS

The first and most straightforward approach to dependence testing is a *path insensitive* analysis that assumes any statement in the method body might execute. This analysis proceeds in three steps:

1. Extracting the *rooted access paths* of every read and write performed in the tree. This involves associating every read and write to a field of a node in the method body with a field that can be reached through a series of accesses starting from **root**.

2. Identifying *conflicting access paths*. This involves determining whether, for two access expressions, at least one of which performs a write, there exist two distinct nodes in the tree where if the first access path were rooted at the first node, and the second access path were rooted at the second, the two paths would refer to the same node.

3. Determining whether any conflicting access paths imply a possible dependence that precludes point blocking.

If step 3 yields no problematic accesses, then point blocking is legal. Each of these steps are now described in more detail.

6.1 Collecting rooted access paths

First, reads and writes to tree nodes in the heap are transformed into reads or writes of *rooted access paths*. This transforms any read or write to tree nodes in the heap into reads or writes of *access paths*. Access paths are elements of the regular set $\mathcal{A} = \mathbf{root}(.f_r)^*$ and *primitive* access paths are members of the set

$\mathcal{A}_p = \mathbf{root}(.f_r)^*.(f_p \mid \iota)$. This allows reasoning about the locations being read and written to by the recursive method relative to the current iteration (*i.e.*, the current values of **root** and **point**). The special field ι allows us to tell when the node itself is being read from or written to. Because only accesses to locations within tree nodes can overlap between recursive calls, only these are used when looking for dependences. In the simple language, the point structures and locals accessed by each traversal are disjoint so they cannot induce any cross-traversal dependences.

To collect the access paths, an abstract interpretation [28] is used. Intuitively, the abstract interpretation executes every path through the recursive method body, determining what (sets of) nodes each node reference can refer to, and associating with each read and write of a tree node field an access path starting from **root**. The analysis isvloosely based on Wiedermann and Cook's [29] approach to identifying paths traversed in object-relational databases.

The abstract store, $\hat{\sigma}$, maps local variables, primitive fields of **point**, and primitive access paths to $\mathcal{P}(\mathbb{Z} \cup \{\mathsf{alloc}, \mathsf{null}\} \cup \bot)$, where $\bot$ represents unknown values; and maps **root** and node references to sets of access paths, $A \in \mathcal{P}(\mathcal{A})$. The program state consists of the abstract store, return flag (as in the concrete semantics), and two access path sets, $\pi_r, \pi_w \in \mathcal{P}(\mathcal{A}_p)$, which collect access paths being read from and written to, respectively.

The abstract semantics are given in Figure 6.1. The evaluation relation for statements and calls is $\langle s,\ \hat{\sigma},\ \pi_r,\ \pi_w,\ \rho\rangle \rightarrow \langle \hat{\sigma}',\ \pi_r',\ \pi_w',\ \rho'\rangle$, and the evaluation relation for expressions is $\langle e,\ \hat{\sigma}\rangle \rightarrow \langle \hat{v},\ \pi\rangle$. Note that expressions return a *set* of values, and can generate new access expressions; these expressions are always reads, so the evaluation relation generates only a single access path set. The initial abstract store maps all locals, primitive fields and primitive access paths to $\{\bot\}$, and maps **root** to $\{\mathbf{root}\}$ and everything else to $\varnothing$. The initial access path sets are $\pi_r = \{\mathbf{root}.\iota\}$ (recall that we assume that **root** is read in every iteration) and $\pi_w = \varnothing$.

Expressions (ALOAD-P, ALOAD-N) are handled as expected, with the only difference from the concrete semantics being that they return a *set* of values instead of

$$\frac{\hat{v} = \hat{\sigma}(\mathbf{point}.f_p)}{\langle \mathbf{point}.f_p,\ \hat{\sigma}\rangle \rightarrow \langle \hat{v},\ \varnothing\rangle}\ [\text{ALOAD-P}] \qquad \frac{A = \hat{\sigma}(n) \quad \hat{v} = \{\hat{\sigma}(a.f_p)\ \mid\ a \in A\}}{\langle n.f_p,\ \hat{\sigma}\rangle \rightarrow \langle \hat{v},\ \{a.f_p\ \mid\ a \in A\}\rangle}\ [\text{ALOAD-N}]$$

$$\frac{\langle e_1,\ \hat{\sigma}\rangle \rightarrow \langle \hat{v_1},\ \pi_1\rangle \quad \langle e_2,\ \hat{\sigma}\rangle \rightarrow \langle \hat{v_2},\ \pi_2\rangle \quad \hat{v} = \hat{v_1} \hat{\oplus} \hat{v_2}}{\langle e_1 \oplus e_2,\ \hat{\sigma}\rangle \rightarrow \langle \hat{v},\ \pi_1 \cup \pi_2\rangle}\ [\text{ABINOP}]$$

$$\frac{\langle e,\ \hat{\sigma}\rangle \rightarrow \langle \hat{v},\ \pi_e\rangle \qquad A_1 = \hat{\sigma}(n) \qquad A_2 = \{a.f_p\ \mid\ a \in A_1\}}{\langle n.f_p := e,\ \sigma,\ \pi_r,\ \pi_w,\ \mathsf{F}\rangle \rightarrow \langle \hat{\sigma}[\mathsf{mapall}(A, f_p, \hat{v})],\ \pi_r \cup \pi_e,\ \pi_w \cup A_2,\ \mathsf{F}\rangle}\ [\text{ASTORE-N}]$$

$$\frac{A_1 = \sigma(n_2) \qquad A_2 = \{a.f_r\ \mid\ a \in A_1\}}{\langle n_1 := n_2.f_r,\ \hat{\sigma},\ \pi_r,\ \pi_w,\ \mathsf{F}\rangle \rightarrow \langle \hat{\sigma}[n_1 \mapsto A_2],\ \pi_r \cup \{a.\iota\ \mid\ a \in A_2\},\ \pi_w,\ \mathsf{F}\rangle}\ [\text{ADEF-N}]$$

$$\frac{A_1 = \sigma(n) \qquad A_2 = \{a.f_r\ \mid\ a \in A_1\}}{\langle n.f_r := \mathsf{alloc},\ \hat{\sigma},\ \pi_r,\ \pi_w,\ \mathsf{F}\rangle \rightarrow \langle \hat{\sigma},\ \pi_r,\ \pi_w \cup \{a.\iota\ \mid\ a \in A_2\},\ \mathsf{F}\rangle}\ [\text{AALLOC}]$$

$$\frac{\begin{array}{c}\langle bexp,\ \hat{\sigma}\rangle \rightarrow \langle \hat{v},\ \pi_e\rangle \\ \langle s_1,\ \hat{\sigma},\ \varnothing,\ \varnothing,\ \mathsf{F}\rangle \rightarrow \langle \hat{\sigma}',\ \pi_r',\ \pi_w',\ \rho'\rangle \qquad \langle s_2,\ \hat{\sigma},\ \varnothing,\ \varnothing,\ \mathsf{F}\rangle \rightarrow \langle \hat{\sigma}'',\ \pi_r'',\ \pi_w'',\ \rho''\rangle\end{array}}{\begin{array}{c}\langle \mathsf{if}\ bexp\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2,\ \hat{\sigma},\ \pi_r,\ \pi_w,\ \mathsf{F}\rangle \rightarrow \\ \langle \hat{\sigma}' \sqcup \hat{\sigma}'',\ \pi_r \cup \pi_r' \cup \pi_r'' \cup \pi_e,\ \pi_w \cup \pi_w' \cup \pi_w'',\ \rho' \wedge \rho''\rangle\end{array}}\ [\text{AIF}]$$

Fig. 6.1.: Abstract semantics to collect access expressions

just one, and that expressions that reference the tree (see ALOAD-N) can add accesses to the access set. Binary operations yield the result of applying the operation to all pairs of values from the two operands' value sets (with the operation yielding $\perp$ if one of the values is $\perp$).

The rules for **skip** and **return** are not presented, as they are analogous to the concrete semantics, simply passing through the abstract store, heap and access path sets. The rules for sequencing of statements thread through the access path sets, setting the return flag and skipping over the execution of subsequent statements if necessary. Interestingly, function calls (**recurse**) are handled much like **skip**. Even though a call reads an access path to make the recursive call, that read is instead associated with the beginning of the next iteration (see Section 4.2), and is captured by the initial access path set of $\mathbf{root}.\iota$.

ADEF-L evaluates the expression, collecting any new access paths that arise, and returning a set of values, which are then mapped to the local variable being defined. ASTORE-N, which provides the semantics for $n.f_p := e$, shows an example of adding new access paths. After looking up the set of access paths that n is mapped to, for each such access path a, we add $a.f_p$ to the set of written access paths. The helper function `mapall` takes care of mapping each of the primitive access paths accessed by $n.f_p$ to the result of evaluating e. ADEF-N adds $a.f_r.\iota$ to the set of read access paths for all a that n_2 is mapped to.

AALLOC is interesting. It creates a new access path, indicating that $n.f_r.\iota$ has been written to. It only changes the store by setting the special primitive field $n.f_r.\iota$ to **alloc**. No other access paths are changed. In essence, the abstract semantics assume the tree structure itself already exists. Allocating a new node does not add a new node to the tree. Instead, it just writes to an existing node, as recorded by the access. The assumption that programs initialize fields before accessing them means

that there is no worry about updating the values of any other fields.[1] A similar rule is used for **null**.

AIF, unsurprisingly, runs both branches of the if statement, collecting the access paths from the boolean expression as well as both branches of the if statement. $\hat{\sigma}' \sqcup \hat{\sigma}''$ creates a new abstract store, where variable or access path maps to the union of its mappings in $\hat{\sigma}'$ and $\hat{\sigma}''$. Note, too, that if both branches of the if statement call **return**, evaluating the if statement sets the return flag to true.

6.2 Identifying conflicting access expressions

After collecting the accesses for the recursive method, the next step is to determine which accesses could result in dependences—two accesses that touch the same location in the tree, with at least one of them a write.

Definition 6.2.1 *For a pair of accesses,* $\mathbf{root}.\alpha$ *and* $\mathbf{root}.\beta$*, the two access paths* collide*—written* $\mathbf{root}.\alpha \sim \mathbf{root}.\beta$*—if there exists a two nodes in a tree (of unbounded size),* n_1 *and* n_2 *such that* $n_1.\alpha$ *refers to the same location as* $n_2.\beta$*.*

This definition lends itself to a straightforward approach to finding access paths that collide. Consider the access path pair $\mathbf{root}.\alpha \in \pi_w$ and $\mathbf{root}.\beta \in (\pi_w \cup \pi_r)$. Without loss of generality, let α be the longer access path than β (*i.e.*, it contains at least as many field dereferences). Then $\mathbf{root}.\alpha \sim \mathbf{root}.\beta$ iff β is a suffix of α.

If β is not a suffix of α, then, because the access paths traverse a tree, there is no way for the two to refer to the same field. Conversely, if β *is* a suffix of α, then let γ be a sequence of field accesses such that $\gamma.\beta = \alpha$. Note that γ's last field access must be a recursive field (if $\beta \neq \alpha$, otherwise $\gamma = \epsilon$). Then let n_1 be an arbitrary node in the tree (for example, the global root of the tree), and let n_2 be the node at $n_1.\gamma$. It is clear that $n_1.\alpha = n_2.\beta$.

[1] AALLOC introduces some inexactness to the set of accesses: if a new node is allocated for an access path, old node references that have the same access path will appear to access the new node as well. This does not affect soundness, as it can only introduce additional dependences.

If two access paths collide and one of them is a write, then there is a potential dependence between them. The set of such pairs can then be computed, $S \subseteq \pi_w \times (\pi_w \cup \pi_r)$:

$$S = \{(a, b) \mid a \in \pi_w \wedge b \in (\pi_w \cup \pi_r) \wedge a \sim b\}$$

6.3 Applying the dependence test

After collecting the access paths, and identifying potential dependences, the final step is to determine whether the conflicting access paths preclude point blocking.

Note that the access paths in S are relative to **root**, which is the index identifier for the traversal "loop" in the application. When iteration (p, n) executes a statement that reads from access path $\mathbf{root}.\alpha$, the field in the tree being read is $n.\alpha$. For each pair of conflicting access paths in S, $(\mathbf{root}.\alpha, \mathbf{root}.\beta)$[2], compute γ as described previously. Let p_1 and p_2 be points such that $p_1 \prec p_2$. For all nodes n, during iteration $(p_1, n.\gamma)$, location $n.\gamma.\beta$ may be accessed by some statement s_1, and during iteration (p_2, n), location $n.\alpha$ may be accessed by some statement s_2. By the definition of conflicting accesses, $s_1(p_1, n.\gamma) \bowtie s_2(p_2, n)$.

By the dependence test in Equation 4.2, it is clear that for these potential dependences not to preclude point blocking, $n.\gamma \preceq_a n$ must be true. This can only be the case if $\gamma = \epsilon$. By verifying this condition for all pairs of conflicting access paths, it is possible to determine whether point blocking is legal.

Soundness The key proof obligation to prove the soundness of this dependence analysis is to show that the set of accesses collected by the abstract interpretation is able to find every s_i and s_j where there exist p_i, p_j, n_i, n_j such that $s_i(p_i, n_i) \bowtie_e s_j(p_j, n_j)$. If the set of statements tested for interference is a superset of these s_i and s_j, then the remainder of the dependence analysis (which ensures the proper ordering of p_i, p_j, n_i and n_j) soundly applies the dependence test from Equation 4.2. To prove this, it must be shown that if there are two statements that could interfere

[2]Assume, without loss of generality, that β is a suffix of α.

with each other in two specific iterations, there must be a pair of conflicting accesses that conflict in the same two iterations.

Theorem 6.3.1 *If there exist s_i and s_j such that there exist p_i, p_j, n_i, n_j and $s_i(p_i, n_i) \bowtie_e s_j(p_j, n_j)$, there exists an access path pair $(\mathbf{root}.\alpha,\ \mathbf{root}.\beta) \in S$ such that $n_i.\alpha = n_j.\beta$.*

Proof sketch: Note that the only way for two statements to interfere in the simple language is if they access the same fields of a tree node. Note further that any tree node accessed by a recursive method body *must* be accessible from **root**, and it can be accessed by only one path. Assume, without loss of generality, that s_i is a write and s_j is a read, and that the interference is through a primitive field access. Then s_i must be of the form $n_x.f_p := ...$ and s_j must contain an expression of the form $n_y.f_p$ By the antecedent, there must be some node m such that when **root** is mapped to n_i, $n_x = m$—and there must be exactly one access path $\mathbf{root}.\alpha = m$—and likewise, when **root** is mapped to n_j, there must be some access path $\mathbf{root}.\beta = m$. Hence, $n_i.\alpha = n_j.\beta$. By structural induction on the abstract semantics, upon encountering statement s_i, the abstract store will contain a mapping from n_x to $\mathbf{root}.\alpha$, adding the access $\mathbf{root}.\alpha.f_p$ to π_w; likewise, upon encountering s_j, $\mathbf{root}.\beta.f_p$ will be added to π_r. Because the abstract interpretation explores all paths, both accesses will be in the access path sets at the end of execution. Moreover, because both accesses refer to the same node in the tree, and each node in the tree can be accessed by only one path from the global root of the tree, either $\alpha.f_p$ will be a suffix of $\beta.f_p$ or vice versa, and the pair will be added to the set of conflicting accesses.

6.4 Examples

Quadtree traversal Running the abstract interpretation over the example from Figure 5.5 generates the following access paths:
$\pi_w = \{\mathbf{root}.v\}$, $\pi_r = \{\mathbf{root}.\iota, \mathbf{root}.v, \mathbf{root}.leaf\}$

There is one pair of conflicting access paths: ($\mathbf{root}.v$, $\mathbf{root}.v$). For two points, p_1 and p_2, with $p_1 \prec p_2$, iteration (p_1, n) writes to the same location that (p_2, n) does. For this pair, $\gamma = \epsilon$, so the dependence does not preclude point blocking. In particular, if p_1 and p_2 are in the same block, p_1 will perform its write before p_2 does, just as in the original, non-blocked code. Hence, despite the dependence between traversals, point blocking is legal for this code. Note that the previously described simple dependence test of Jo and Kulkarni would have claimed that point blocking is illegal here, as the traversals are not independent of each other.

BST insertion Running the analysis over the BST insertion example from Figure 5.6 generates the following access paths:

$\pi_w = \{\mathbf{root}.v, \mathbf{root}.l.\iota, \mathbf{root}.l.v, \mathbf{root}.r.\iota, \mathbf{root}.r.v\}$,

$\pi_r = \{\mathbf{root}.\iota, \mathbf{root}.v, \mathbf{root}.l.\iota, \mathbf{root}.r.\iota\}$

Each access path in π_w conflicts with itself. But by the same analysis as in the quadtree example, these conflicts do not preclude point blocking: they all arise when different points are at the same node of the tree. However, the access paths $root.v \in \pi_r$ and $root.l.v \in \pi_w$ conflict with each other. Here, iteration $(p_1, n.l)$ reads from the same location that iteration (p_2, n) writes to. γ is l in this case, so the potential dependence precludes point blocking. However, point blocking is legal for this code—the path-insensitive dependence analysis is too conservative. To develop a dependence analysis that correctly handles this code, the *conditions* under which certain accesses happen must also be considered.

7. CONDITIONAL DEPENDENCE ANALYSIS

The dependence test covered in the previous section finds point blocking to be legal unless there is a dependence between an earlier point's access in a later node in the traversal to a later point's access in an earlier node in the traversal. Even using the dependence test, the code in Figure 5.6 still exhibits a problematic dependence. This is because dependence test of the previous section assumes that all accesses in an iteration will happen. However, there are many situations in which it is possible to rule out some subset of accesses such that the problematic dependences will not occur. Consider two points p_1 and p_2 with $p_1 \prec p_2$, and the tree in Figure 3.1(a). When point p_1 is at node c, it reads from $c.v$ in line 1. That is the same field that point p_2 could write to at node b in line 6, when it writes to $\mathbf{root}.l.v$.

However, as previously stated, point blocking is still legal for this code. This is because reads and writes performed during traversals are not always unconditional in each iteration. It is often the case that if a traversal performs a particular access, other traversals *cannot* perform certain accesses: if iteration (p_1, c) reads from $c.v$, it is clear that iteration (p_1, b) must have established that $b.l \neq$ **null** (as that is the only way for **recurse** $(b.l, \mathbf{point})$ to be executed in line 8). Hence, when iteration (p_2, b) executes, it *will not* execute line 6, and the access that causes the problematic dependence will not happen.

This chapter describes how the dependence analysis of the previous section can be augmented to engage in this type of reasoning on conditions. The key insight is that the symbolic *path conditions* under which various accesses might occur can be determined, relative to arbitrary nodes in the tree. Given these conditions, it is possible to prove that if the first of two potentially conflicting accesses occurs, the second cannot. In other words, $\bowtie_*$, the test for interference between two statements, can

$$v \in \mathbb{Z} \qquad b \in \{\mathsf{T}, \mathsf{F}\} \qquad a \in \mathcal{A} \qquad a_p \in \mathcal{A}_p$$

$$E = a_p \mid v \mid E \oplus E$$

$$P = E \odot E \mid a.\iota = \textbf{null} \mid a.\iota \neq \textbf{null}$$

$$F = b \mid P \mid F \wedge F \mid F \vee F$$

Fig. 7.1.: Logical fragment for path conditions

be refined to be more precise about whether the two interfering statements actually both execute.

7.1 Attaching conditions to access paths

The first step is to attach symbolic path conditions to each access path that can occur in a program. A path condition is some logical formula, $\phi \in (F \cup E)$, over access paths and values (including **null**), produced from the logical fragment given in Figure 7.1.

To track path conditions, the abstract semantics of the previous section are extended. First, the access paths are extended to be a 3-tuple of an access path, a formula in the logic, and a flag that indicates whether the access path was a *strong* access. If an access path was generated by a variable dereference that only pointed to a single access path, the access path is strong, and is amenable to strong updates.

Expressions now yield formulae ($\Phi \in \mathcal{P}(F \cup E)$) in addition to sets of values (an expression can produce more than one conditional formula because variables accessed in an expression may map to more than one access path). Statements and expressions carry with them a *condition*, k, a predicate defining when statement might execute. The conditions capture a precondition that holds before a basic block executes. Hence, these conditions are updated when executing if statements. Figure 7.2 shows the relevant portion of the extended semantics. The evaluation relation for expressions is now $\langle e,\ \hat{\sigma},\ k\rangle \rightarrow \langle \hat{v},\ \pi_e,\ \Phi\rangle$ and the evaluation relation for statements is now

$$\frac{\hat{v} = \hat{\sigma}(\mathbf{point}.f_p)}{\langle \mathbf{point}.f_p,\ \hat{\sigma},\ k\rangle \rightarrow \langle \hat{v},\ \varnothing,\ \{\mathbf{point}.f_p\}\rangle}\ [\text{FLOAD-P}]$$

$$\frac{A = \hat{\sigma}(n) \quad \hat{v} = \{\hat{\sigma}(a.f_p)\ \mid\ a \in A\}}{\langle n.f_p,\ \hat{\sigma},\ k\rangle \rightarrow \langle \hat{v},\ \{a.f_p[k][|A| = 1]\ \mid\ a \in A\},\ \{a.f_p\ \mid\ a \in A\}\rangle}\ [\text{FLOAD-N}]$$

$$\frac{\langle e_1,\ \hat{\sigma},\ k\rangle \rightarrow \langle \hat{v_1},\ \pi_1,\ \Phi_1\rangle \quad \langle e_2,\ \hat{\sigma},\ k\rangle \rightarrow \langle \hat{v_2},\ \pi_2,\ \Phi_2\rangle}{\langle e_1 \odot e_2,\ \hat{\sigma},\ k\rangle \rightarrow \langle \{\bot\},\ \pi_1 \sqcup \pi_2,\ \Phi_1 \hat{\odot} \Phi_2\rangle}\ [\text{FCMPOP}]$$

$$\frac{\begin{array}{c}\langle bexp,\ \hat{\sigma},\ k\rangle \rightarrow \langle \hat{v},\ \pi_e,\ \Phi\rangle \qquad \forall a[k_*][b_*] \in \pi_w\ .\ \nexists a[k][*] \in \pi_e \\ \langle s_1,\ \hat{\sigma},\ \varnothing,\ \varnothing,\ k \wedge (\bigvee\{\phi\ \mid\ \phi \in \Phi\}),\ \mathsf{F}\rangle \rightarrow \langle \hat{\sigma}',\ \pi'_r,\ \pi'_w,\ k',\ \rho'\rangle \\ \langle s_2,\ \hat{\sigma},\ \varnothing,\ \varnothing,\ k \wedge (\bigvee\{\neg\phi\ \mid\ \phi \in \Phi\}),\ \mathsf{F}\rangle \rightarrow \langle \hat{\sigma}'',\ \pi''_r,\ \pi''_w,\ k'',\ \rho''\rangle \\ k_T = \mathsf{munge}(\hat{\sigma}', k', \pi'_w) \wedge \neg\rho' \qquad k_F = \mathsf{munge}(\hat{\sigma}'', k'', \pi''_w) \wedge \neg\rho''\end{array}}{\begin{array}{c}\langle \mathsf{if}\ bexp\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2,\ \hat{\sigma},\ \pi_r,\ \pi_w,\ k,\ \mathsf{F}\rangle \rightarrow \\ \langle \hat{\sigma}' \sqcup \hat{\sigma}'',\ \pi_r \sqcup \pi'_r \sqcup \pi''_r \sqcup \pi_e,\ \pi_w \sqcup \pi'_w \sqcup \pi''_w,\ k_T \vee k_F,\ \rho' \wedge \rho''\rangle\end{array}}\ [\text{FIF1}]$$

$$\frac{\begin{array}{c}\langle bexp,\ \hat{\sigma},\ k\rangle \rightarrow \langle \hat{v},\ \pi_e,\ \Phi\rangle \qquad \text{FIF1 does not apply} \\ \langle s_1,\ \hat{\sigma},\ \varnothing,\ \varnothing,\ k,\ \mathsf{F}\rangle \rightarrow \langle \hat{\sigma}',\ \pi'_r,\ \pi'_w,\ k',\ \rho'\rangle \\ \langle s_2,\ \hat{\sigma},\ \varnothing,\ \varnothing,\ k,\ \mathsf{F}\rangle \rightarrow \langle \hat{\sigma}'',\ \pi''_r,\ \pi''_w,\ k'',\ \rho''\rangle\end{array}}{\begin{array}{c}\langle \mathsf{if}\ bexp\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2,\ \hat{\sigma},\ \pi_r,\ \pi_w,\ k,\ \mathsf{F}\rangle \rightarrow \\ \langle \hat{\sigma}' \sqcup \hat{\sigma}'',\ \pi_r \sqcup \pi'_r \sqcup \pi''_r \sqcup \pi_e,\ \pi_w \sqcup \pi'_w \sqcup \pi''_w,\ k,\ \rho' \wedge \rho''\rangle\end{array}}\ [\text{FIF2}]$$

Fig. 7.2.: Abstract semantics to collect conditional access expressions

$\langle s,\ \hat{\sigma},\ \pi_r,\ \pi_w,\ k,\ \rho\rangle \rightarrow \langle \hat{\sigma}',\ \pi'_r,\ \pi'_w,\ k',\ \rho'\rangle$. The starting path condition for a program is T.

Expressions accessing fields generate atomic formulae as expected. When an expression generates an access path, the condition for the expression is attached to the access path. The cardinality of the access path set in the store is checked to determine whether the generated access path is a strong access. Comparison operations produce a new formula set from combining all pairs of formulae from its operands' formula sets

(*e.g.*, if one operand has the formulae $\{\mathbf{point}.x\}$ and the other has the formulae $\{1, 2\}$, then combining them with $\hat{=}$ produces the formula set $\{\mathbf{point}.x = 1, \mathbf{point}.y = 2\}$). The rules for most statements are not shown; the only difference between these semantics and the semantics in Figure 6.1 is that when an access occurs, the statement's condition is associated with the access path. The strong tag is set, and a strong update performed on the abstract store, if the access path refers to exactly one node.

The other key rules in the semantics are for **if** statements. The formulae generated by the test condition are attached to the true and false branches of the **if** statement. If the test expression generates multiple formulae, the true branch is taken if *any* of the formulae are true, while the false branch is taken if *any* of the formulae are false; the conditions for the two branches are assembled appropriately. Joining together access paths ($\sqcup$) logically ors the conditions under which the access paths occur, and logically ands the strong tag.

The path condition after the **if** statement executes is subtle. It seems as though it should simply revert to the original condition, k, after control has re-converged. However, along one of the branches of the if statement, a write may have happened that invalidated part of the path condition. Consider **if** $\mathbf{root}.v = 0$ **then** $\mathbf{root}.v := 1$ **else skip**. After the statement executes, $\mathbf{root}.v \neq 0 \vee \mathbf{root}.v = 1$. The path condition must be updated to account for any writes made along the branch of an if statement. In other words, writes that occur along a branch of an if statement might invalidate portions of the condition under which the branch occurred.

The helper function **munge**$(\hat{\sigma}, k, \pi_w)$ creates two formulae: k_1, which captures all possible values of access paths that were *definitely* written along the branch (determined by checking the strong tags); and k_2, which removes from k conditions that are invalidated by writes that *may* happen along the branch. The function returns $k_1 \wedge k_2$, which amounts to a postcondition for that branch of the **if** statement. The disjunction of the munged conditions from both branches of the **if** statement yields the *precondition* for the following statement. Note that if there are no writes along the branches, then the resulting path condition will again be k.

$$
\begin{aligned}
\pi_w = \{&\mathbf{root}.v\ [\mathbf{root}.v = -1],\\
&\mathbf{root}.l.\iota\ [\mathbf{root}.v \neq -1 \wedge \mathbf{root}.v < \mathbf{point}.v \wedge \mathbf{root}.l.\iota = \textsf{null}],\\
&\mathbf{root}.l.v\ [\mathbf{root}.v \neq -1 \wedge \mathbf{root}.v < \mathbf{point}.v \wedge \mathbf{root}.l.\iota = \textsf{null}],\\
&\mathbf{root}.r.\iota\ [\mathbf{root}.v \neq -1 \wedge \mathbf{root}.v \geq \mathbf{point}.v \wedge \mathbf{root}.r.\iota = \textsf{null}],\\
&\mathbf{root}.r.v\ [\mathbf{root}.v \neq -1 \wedge \mathbf{root}.v \geq \mathbf{point}.v \wedge \mathbf{root}.r.\iota = \textsf{null}]\}\\
\pi_r = \{&\mathbf{root}\ [\textsf{T}], \mathbf{root}.v\ [\textsf{T}],\\
&\mathbf{root}.l.\iota\ [\mathbf{root}.v \neq -1 \wedge \mathbf{root}.v < \mathbf{point}.v],\\
&\mathbf{root}.r.\iota\ [\mathbf{root}.v \neq -1 \wedge \mathbf{root}.v \geq \mathbf{point}.v]\}
\end{aligned}
$$

Fig. 7.3.: Conditional access paths in BST insertion

This treatment of **if** statements only occurs if the condition of the **if** statement accesses portions of the tree that have not yet been written (see the second premise of FIF1); otherwise, no conditional information is passed along branches of the **if** statement (FIF2). Figure 7.3 shows the results of running this analysis on the BST-insertion example (the tag for strong accesses is elided for brevity).

A similar analysis can be used to determine under which conditions recursive calls are made. The only difference is that the path condition prior to making the recursive call is also **munge**d to produce a precondition for the call. In essence, the condition attached to the recursive call is a statement about the state of the tree when the call is made. For example, the condition for the recursive call in line 8 of Figure 5.6 is:

$$
\begin{aligned}
&\mathbf{root}.v \neq -1 \wedge (\mathbf{root}.v < \mathbf{point}.v) \wedge\\
&((\mathbf{root}.l.\iota = \textsf{alloc} \wedge \mathbf{root}.l.v = -1) \vee \mathbf{root}.l.\iota \neq \textsf{null})
\end{aligned}
$$

7.2 Using conditions to disprove dependences

Suppose there is a potential dependence between two accesses $(\mathbf{root}.\alpha[\phi_\alpha], \mathbf{root}.\beta[\phi_\beta])$ where $\alpha = \gamma.\beta$. The dependence that appears to preclude point blocking arises when $(p_1, n.\gamma)$ executes access path $\mathbf{root}.\beta$, and (p_2, n) executes access path $\mathbf{root}.\alpha$. The formulae ϕ_α and ϕ_β indicate the conditions under which

the two accesses occur. If it can be shown that whenever ϕ_β is true during iteration $(p_1, n.\gamma)$, ϕ_α *will not* be true during iteration (p_2, n), then the dependence cannot arise. The procedure for doing this proceeds as follows:

1. First, construct a more precise condition for access $\mathbf{root}.\beta$. In particular, ϕ_β is a formula in terms of access paths rooted at **root**, which must be bound to the dynamic iteration instance. This is easily accomplished by substituting $n.\gamma$ for **root** to create ϕ'_β. Then substitute n for **root** to create ϕ'_α and query an SMT solver to determine ϕ'_β is incompatible with ϕ'_α. If so, continue to step 3.

2. ϕ'_β being compatible with ϕ'_α does not mean that both accesses will happen. ϕ'_β was computed with a starting path condition of T. To make the condition more precise, *propagate* the conditions of the previous iteration down to $(p_1, n.\gamma)$. Define δ such that $\delta.f_r = \gamma$. Substituting $n.\delta$ for the path conditions associated with all recursive calls $\mathbf{recurse}\ (\mathbf{root}.f_r, \mathbf{point})$, information about the state of the tree during iteration $(p_1, n.\delta)$, immediately before making a recursive call to start iteration $(p_1, n.\gamma)$, can be obtained. The disjunction of all such recursive conditions (call this ϕ_δ) is a sound approximation of the state of the tree before $(p_1, n.\gamma)$ executes. Essentially, one instance of the recursive method is inlined. Then the abstract interpretation with an initial condition of ϕ_δ is re-run, generating a stronger condition under which access $\mathbf{root}.\beta$ occurs.

   This "inlining" process is repeated, backing up one iteration at a time, until iteration (p_1, n) is reached. This cannot be inlined beyond this point—n could be the global root of the tree, and hence there could be no earlier iteration in the traversal. Note that this process is decidable, as there are a finite number of paths through the recursive method body. In practice, potentially-dependent iterations are nearby in the tree, so inlining only needs to be done one or two times.

   After performing this inlining, the result is a much stronger path condition, ϕ'_β, for the problematic access. The SMT solver is then queried once again to

determine whether the path conditions are incompatible. If they are not, then this dependence is declared as a true conflict, and as such it fails the overall dependence test.

3. If ϕ'_α is incompatible with ϕ'_β, then it is determined that whatever computation p_1 performs during its traversal prevents p_2 from performing the access $\mathbf{root}.\alpha$. It is possible, however, for a traversal in between p_1 and p_2 to "reactivate" p_2's bad access. Thus, it must be ensured that no other accesses can affect the path condition ϕ'_α that prevents p_2 from performing the bad access. Any access paths in π_w that collide with any access paths in ϕ'_α are searched; these writes affect the path condition, and hence if some iteration performs the write, it may cause the bad access to occur. The same conditional dependence test is used to ensure that *those* accesses cannot happen. Note that any access path that appears in ϕ'_α must also appear in π_r. Hence, there are a bounded number of access paths to consider and the number of tests is finite.

7.3 Example

Consider the conflicting access paths $(\mathbf{root}.v[\mathsf{T}], \mathbf{root}.l.v[\mathbf{root}.v \neq -1 \wedge \mathbf{root}.v < \mathbf{point}.v \wedge \mathbf{root}.l.\iota = \mathsf{null}])$. These access paths preclude point blocking if iteration $(p_1, n.l)$ performs the first access and iteration (p_2, n) performs the second access. Substitute $n.l$ and n for the conditions to generate: $\phi'_\beta = \mathsf{T}$ and $\phi'_\alpha = n.v \neq -1 \wedge n.v < n.v \wedge n.l.\iota = \mathsf{null}$. These conditions are not incompatible with each other, so the recursive method is "unrolled" by one iteration, passing the recursion condition from iteration (p_1, n) to $(p_1, n.l)$. The new ϕ'_β is:

$$n.v \neq -1 \wedge (n.v < n.v) \ \wedge ((n.l.\iota = \mathsf{alloc} \wedge n.l.v = -1) \vee n.l.\iota \neq \mathsf{null})$$

The refined condition under which iteration $(p_1, n.l)$ reads $n.l.v$ is clearly incompatible with the condition under which iteration (p_2, n) writes $n.l.v$—the latter requires that $n.l.\iota = \mathsf{null}$, while the former only happens when $n.l.\iota \neq \mathsf{null}$.

Finally, it must be made sure that there is no intervening traversal that writes to $n.l.\iota$, possibly "re-activating" the write in iteration (p_2, n). Note that the only access path that writes to $n.l.\iota$ does so under the same condition as the write to $n.l.v$, and is therefore invalidated by the same argument. Repeating the process for all conflicting access paths, it can be determined that all pairs that might introduce a problematic dependence are incompatible with each other.

8. IMPLEMENTATION AND EVALUATION

8.1 Analysis implementation

The analysis was implemented in JastAdd [30], a compilation framework for Java. The analysis analyzes recursive Java methods that are constrained to only use operations analogous to the operations in the simple specification language (Section 5); if a method does not obey those restrictions, it cannot be analyzed. It is assumed that either a shape analysis or a programmer annotation has established that the recursive data structure being traversed is a tree. The path-insensitive analysis assumes that local variables cannot overlap between different traversals. Further analysis to determine possible aliasing is needed if the algorithm allows assigning tree nodes to local variables. The conditional analysis (Section 7) passes path conditions to the Z3 SMT solver [31], which checks whether they are compatible or not. The conditional analysis currently assumes that all writes used to compute post-conditions are strong (*i.e.*, in a single basic block, each write *definitely* happens), which is valid for the benchmarks we have studied.

Table 8.1.: Analysis results, runtimes in seconds (with 95% confidence intervals).

Benchmark	**Conflicts**	**Z3 calls**	**No Z3 Runtime**	**Total Runtime**
ll	1	1	$0.7914 \pm .0945$	0.8174 ± 0.0959
bst	8	16	0.8800 ± 0.136	1.220 ± 0.154
skew	16	32	0.9527 ± 0.0430	1.687 ± 0.0498
kdtree	510	3060	23.94 ± 0.462	109.0 ± 0.481
bh	3448	27584	280.3 ± 6.78	1432 ± 16.5

8.2 Benchmarks

The dependence test of Equation 4.2 was applied to five benchmarks, ranging from simple microbenchmarks to complex data-structure construction algorithms:

***ll*:** Repeatedly appending values to a linked list, with traversal starting from the head of the list.

***bst*:** Building a binary search tree, as in Figure 1.1.

***skew*:** Building a *skew-heap* [32].[1]

***bh*:** Building a Barnes-Hut quadtree.

***kdtree*:** Building a kd-tree using top-down insertion.

The analysis is able to prove that the each of these benchmarks passes the dependence test, and hence can be soundly transformed using point blocking, as well as other optimizations; the following section describes the performance benefits of these transformations. Note that not only do all of these benchmarks modify the contents of the tree structure being traversed, they also *morph* the structure of the tree by adding additional nodes and edges. In all five cases, the full conditional dependence analysis of Section 7 is required to verify the dependence test.

To see that proving the legality of these transformations is non-trivial, consider the tree-building code in Figure 8.1. Barnes-Hut is an algorithm for performing n-body simulation where the tree is built by inserting points one by one from the root. When an insertion reaches an interior node where the appropriate child node has not been created yet, it allocates the node and places the appropriate node data there. If a traversal reaches an interior node where the child node is a leaf (so another point is already there), it moves the existing point one step farther down the tree and marks the old child as no longer a leaf, and then continues recursion. Otherwise the traversal just continues traversing the tree.

[1]The algorithm was slightly modified to fit our language restrictions.

```
1.  if (root.x ≥ point.x)
2.    if (root.y ≥ point.y)
3.      if root.child_0 = null
4.        root.child_0 := alloc; root.child_0.isLeaf := 1;
5.        root.child_0.x := point.x; root.child_0.y := point.y;
6.      else
7.        if root.child_0.isLeaf = 1
8.          //Put point at root.child_0
9.          //at root.child_0.child_n
10.          /* compute child_n */
11.          root.child_0.child_n := alloc ...
12.          root.child_0.isLeaf := 0
13.          recurse (root.child_0, point);
14.        else
15.          recurse (root.child_0, point); return
16.    else
17.      //repeat code for root.child_1
18. else
19.    if (root.y ≥ point.y)
20.      //repeat code for root.child_2
21.    else
22.      //repeat code for root.child_3
```

Fig. 8.1.: 2D Barnes-Hut Tree Building

There are two potentially problematic dependences here: creating a new child in lines 3–5, and creating a new grandchild in lines 8–12. In both cases, conditional dependence analysis rules out the dependence: in the first case, the code only executes if the child is **null**, and after executing the code, the child is no longer **null**. In the second case, the code only executes if the child is a leaf, and after executing the code, the child is no longer a leaf. By tracking these conditions, the analysis proves that if a point performs the first access in a dependence pair, a later point cannot perform the second access in the dependence.

Analysis performance

Table 8.1 summarizes the results of running our analysis on each benchmark. Conflicts is the number of pairs that require the conditional dependence analysis of Section 7 to rule out as problematic. The number of Z3 calls made for each benchmark is counted, as benchmarks with more recursive calls require that more paths be checked to rule out conflicts. The upshot of these results is that for all five benchmarks, the simple independence test is not sufficient (some access paths interfere); moreover, the conditional analysis is *required* to verify the dependence test.

Both the overall analysis time, and the analysis time not including calls to Z3 were measured. Most of the benchmarks are analyzed very quickly. Note that ***bh*** takes quite a bit longer than the other benchmarks, due both to the larger number of access paths and to the 8 recursive calls in the method body, which leads to a commensurate increase in the number of Z3 calls.

Transformation evaluation

After proving that the benchmarks pass the dependence test, three different transformations were applied them, using the legality established by the dependence test:

1. *Point blocking*, described in detail in Section 3.2

2. *Traversal splicing* [17]. In contrast to point blocking, traversal splicing tiles the "tree loop" instead of the point loop. The original version of traversal splicing reorders the point loop during execution, and hence is not amenable to the dependence test that we develop in this work. However, for benchmarks where a point only visits one child of any node, traversal splicing performs no reordering, and hence is legal whenever the dependence test of Equation 4.2 holds.

3. *Parallelization*. It is well-known that top-down tree building algorithms can be parallelized by recursively building left and right subtrees in parallel. An implementation of parallelization from the sequential version of the traversal code is used where point blocking is applied to the code, then each of the left and right calls (e.g., the two recursive calls in Figure 1.1(b)) are run in parallel. The resulting parallel implementation requires no locks and also guaranteed to produce the same tree as the original sequential code.

Experimental configurations

All experiments were run on a 48-core AMD Opteron system running at 2.3 GHz, with 64 KB of L1 cache per core, 512K of L2 cache per core, and 6MB of L3 cache shared among groups of 6 cores. The baseline code for point-blocking is written in Java (and is the same code analyzed by the analysis framework described above). Point blocking fully blocks the code, using block sizes equal to the input size. For infrastructural reasons, the baselines for the traversal splicing experiments and the parallelization experiments are written in C++: the Java version of the benchmarks were analyzed to prove the transformations' legality, then were ported to C++. For the parallelization transformation, Cilk+ [33] was used for parallelism. The parallelized code was run using 4 threads, and compared to a baseline of the Cilk+ code running on a single thread.

Table 8.2.: Speedups of transformed benchmarks (with 95% confidence intervals).

Bench.	**Blocking**	**Splicing**	**Parallelization**
ll	1.42 (1.39, 1.45)	N/A	N/A
bst	2.59 (2.52, 2.65)	2.00 (1.87, 2.13)	1.54 (1.53, 1.56)
skew	1.59 (1.53, 1.66)	0.86 (0.85, 0.86)	0.76 (0.73, 0.79)
kdtree	1.80 (1.75, 1.85)	2.65 (2.64, 2.66)	2.07 (2.01, 2.12)
bh	1.17 (1.15, 1.18)	1.28 (1.27, 1.29)	2.67 (2.58, 2.75)

For ***ll***, 60,000 values were inserted; to avoid stack overflow, tail-call optimization was performed on the *transformed* code. For each of the other benchmarks, trees were built using 10 million points/values. The splicing and parallelization transformations are only applied to the four tree-based benchmarks. Table 8.2 presents the results.

Results discussion

Each of these transformations is able to achieve substantial speedups on most of the benchmarks. The exceptions are *bh*, which has no speedup for point blocking, but good speedup for splicing, and *skew*, where the opposite is true. This is likely because of the structure of those benchmarks and transformations: point blocking tiles the point loop, while traversal splicing tiles the tree loop, and the two benchmarks each benefit from a different transformation. Parallelization has no speedup for ***skew*** due to low available parallelism (there is relatively more work to be done at the root node, which must be done sequentially) and Cilk overheads (a 1.13× speedup is seen when using two cores, which dissipates when using four cores).

As each of these transformations is enabled by this dependence test, and, moreover, *would not* have been proven legal by prior dependence tests, including Jo and Kulkarni's [13], this clearly demonstrates the utility of the precise dependence test and the analyses that check it. he goal of these experiments is not to evaluate these

transformations against each other; indeed, these transformations are not a contribution of this work. Instead, the aim is to show that extending these transformations to a wider class of kernels through this dependence test and dependence analysis is beneficial. Note, for example, that by applying the dependence analysis to ***bh***, almost the entirety of the application—the two major kernels, tree building and tree traversal, comprise 99% of its runtime—is now amenable to point blocking and traversal splicing.

8.2.1 Benchmarks discussion

It is of note that these tree building algorithms are generally used to set up a tree for some other algorithm. For the cases of BST, Linked List, and KDTree, this other algorithm may be run some unbounded number of times, as the data structure is searched any time data in the tree is needed for something. For Barnes-Hut, the tree is used to calculate forces acting within the n-body simulation, which may take much longer than simply building the tree (the tree building took up approximately 3% of the runtime of the baseline benchmark for 10 million nodes). Likewise, skew heaps are often used to create a balanced heap for a heapsort, which may also take significantly more time than building the heap. It would appear that if the amount of time these tree building algorithms take is only a fraction of the amount of time any other algorithm using the tree takes, then optimizing this tree building part is not worthwhile. However, in many of these cases, things that use these tree data structures are highly parallel and easy to heavily optimize; when the parts of the algorithms that are not building up a tree are optimized enough, the difference in runtimes between tree building and the rest of the algorithms may be much closer than initially expected. These sections of programs that are not parallel prove to be a much more difficult problem to analyze than the simpler parallelizable sections, even if they take up a smaller portion of the overall runtime. In addition, it is very important to note that tree building algorithms are not the only recursive tree

algorithms that need this more sophisticated dependence test to prove the legality of point blocking. Indeed, any application that performs multiple repeated traversals on that tree while still writing to various nodes might be amenable to point blocking, and would previously be ruled as unable to be transformed.

9. FUTURE WORK AND CONCLUSIONS

9.1 Future work

Interesting avenues of future work abound. While the current analysis is mainly focused on traversals of trees, note that any acyclic data structure traversed with a recursive method using a single canonical traversal order should be transformable, though they likely require a more sophisticated set of aliasing tests to determine when access paths might collide.

An intriguing problem that bears some similarity to the problem of applying point blocking is scheduling *attribute grammars*, which compute attribute values by performing traversals over a parse tree. A useful optimization for attribute grammars is to compute multiple attributes in a single traversal [26,34]. This requires reasoning about the dependences captured by the attribute grammar to ensure that during a single traversal of the tree, each attribute is computed in the right order. It is likely that this problem can be thought of as an instance of *traversal fusion*: each attribute represents a traversal, and the goal is to determine whether two traversals can be combined into a single traversal, as in loop fusion. Extending the tree dependence analysis to handle this scenario is a promising target for future work.

9.2 Conclusions

This thesis presents techniques for analyzing dependences in programs that recursively traverse trees. It develops an accurate dependence test that identifies only those dependences that preclude point blocking. Through a conditional tree dependence analysis, it is able to prove the legality of point blocking and other transformations

for a wide range of programs, including ones that mutate trees during execution, such as tree building codes.

Through multiple decades of compiler research sophisticated dependence analysis frameworks like the unimodular and polyhedral frameworks were developed to apply transformations like loop tiling to array programs in the face of complex dependences. Despite these decades of research, similar analyses for pointer-based programs have been an elusive target. This thesis presents the first dependence analysis toolkit that can prove the legality of analogous "loop" transformations over pointer-based data structures. It presents ways to determine potentionally problematic dependences in programs that execute a recursive call over many points, showing that dependences that take place only within a single node are never problematic. Because of the way point blocking transforms traversals, every node is still visited by the same set of points in the same order, thus preserving any dependences from accesses within a single node. In addition it shows that even some dependences that happen between multiple nodes are not problematic by showing the specific set of conflicts that need to arise to cause the program's correctness to be violated. By analyzing the paths through a recursive method, some of these conflicts can be shown to never actually arise, allowing point blocking to work correctly on these programs.

All of the tree building benchmarks studied show that the analysis proves they are amenable to some locality- or parallelism-enhancing transformation. While improving the runtimes of these tree-building algorithms may not be a significant improvement over the runtime of an overall application by itself, there may be many cases where algorithms that were previously ruled unable to be transformed could be the majority of an application, and therefore give very significant overall speedups to programs.

REFERENCES

REFERENCES

[1] R. Allen and K. Kennedy, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann, 2001.

[2] J. R. Allen and K. Kennedy, "Automatic loop interchange," in *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, 1984, pp. 233–246. [Online]. Available: http://doi.acm.org/10.1145/502874.502897

[3] U. Banerjee, "Unimodular transformations of double loops," in *Languages and Compilers for Parallel Computing*, 1991.

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 101–113. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375595

[5] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/106972.106981

[6] W. Pugh, "The omega test: a fast and practical integer programming algorithm for dependence analysis," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 4–13. [Online]. Available: http://doi.acm.org/10.1145/125826.125848

[7] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, ser. PLDI '91. New York, NY, USA: ACM, 1991, pp. 30–44. [Online]. Available: http://doi.acm.org/10.1145/113445.113449

[8] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, 1989, pp. 655–664. [Online]. Available: http://doi.acm.org/10.1145/76263.76337

[9] P. Feautrier, "Some efficient solutions to the affine scheduling problem: I. one-dimensional time," *Int. J. Parallel Program.*, vol. 21, pp. 313–348, October 1992. [Online]. Available: http://portal.acm.org/citation.cfm?id=171447.171448

[10] R. A. van Engelen, J. Birch, Y. Shou, B. Walsh, and K. A. Gallivan, "A unified framework for nonlinear dependence testing and symbolic analysis," ser. ICS '04, 2004.

[11] W. Pugh and D. Wonnacott, "Non-linear array dependence analysis," *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pp. 1–14, 1996.

[12] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout, "Non-affine extensions to polyhedral code generation," ser. CGO '14, 2014.

[13] Y. Jo and M. Kulkarni, "Enhancing locality for recursive traversals of recursive structures," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2011, pp. 463–482. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048104

[14] R. Ghiya, L. Hendren, and Y. Zhu, "Detecting parallelism in c programs with recursive data structures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 35–47, 1998.

[15] J. R. Larus and P. N. Hilfinger, "Detecting conflicts between structure accesses," ser. PLDI '88, 1988.

[16] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric Shape Analysis via 3-valued Logic," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 3, pp. 217–298, May 2002. [Online]. Available: http://doi.acm.org/10.1145/514188.514190

[17] Y. Jo and M. Kulkarni, "Automatically enhancing locality for tree traversals with traversal splicing," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2012, pp. 355–374. [Online]. Available: http://doi.acm.org/10.1145/2384616.2384643

[18] C. D. Krieger, M. M. Strout, C. Olschanowsky, A. Stone, S. Guzik, X. Gao, C. Bertolli, P. H. J. Kelly, G. R. Mudalige, B. van Straalen, and S. Williams, "Loop chaining: A programming abstraction for balancing locality and parallelism." in *IPDPS Workshops*. IEEE, 2013, pp. 375–384. [Online]. Available: http://dblp.uni-trier.de/db/conf/ipps/ipdps2013w.html#KriegerSOSGGBKMSW13

[19] M. M. Strout, F. Luporini, C. D. Krieger, C. Bertolli, G.-T. Bercea, C. Olschanowsky, J. Ramanujam, and P. H. J. Kelly, "Generalizing run-time tiling with the loop chain abstraction," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1136–1145. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2014.118

[20] M. M. Strout, L. Carter, and J. Ferrante, "Compile-time Composition of Run-time Data and Iteration Reorderings," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 91–102. [Online]. Available: http://doi.acm.org/10.1145/781131.781142

[21] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, "A framework for enhancing data reuse via associative reordering," ser. PLDI '14, 2014.

[22] P. Feautrier, "Automatic parallelization in the polytope model," *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pp. 79 – 103, 1996.

[23] R. Ghiya and L. J. Hendren, "Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1996, pp. 1–15.

[24] P. Madhusudan, X. Qiu, and A. Stefanescu, "Recursive Proofs for Inductive Tree Data-structures," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012, pp. 123–136. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103673

[25] D. Knuth, "Semantics of context-free languages," *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, 1968. [Online]. Available: http://dx.doi.org/10.1007/BF01692511

[26] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodik, "Parallel schedule synthesis for attribute grammars," ser. PPoPP '13, 2013.

[27] M. Jourdan, "A survey of parallel attribute evaluation methods," in *Attribute Grammars, Applications and Systems*, ser. Lecture Notes in Computer Science, H. Alblas and B. Melichar, Eds. Springer Berlin Heidelberg, 1991, vol. 545, pp. 234–255. [Online]. Available: http://dx.doi.org/10.1007/3-540-54572-7_9

[28] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977, pp. 238–252. [Online]. Available: http://doi.acm.org/10.1145/512950.512973

[29] B. Wiedermann and W. R. Cook, "Extracting queries by static analysis of transparent persistence," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '07. New York, NY, USA: ACM, 2007, pp. 199–210. [Online]. Available: http://doi.acm.org/10.1145/1190216.1190248

[30] T. Ekman and G. Hedin, "The JastAdd Extensible Java Compiler," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 1–18. [Online]. Available: http://doi.acm.org/10.1145/1297027.1297029

[31] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[32] D. D. Sleator and R. E. Tarjan, "Self Adjusting Heaps," *SIAM J. Comput.*, vol. 15, no. 1, pp. 52–69, Feb. 1986. [Online]. Available: http://dx.doi.org/10.1137/0215004

[33] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223. [Online]. Available: http://doi.acm.org/10.1145/277650.277725

[34] U. Kastens, "Ordered attribute grammars," *Acta Informatica*, vol. 13, pp. 229–256, 1980.

APPENDIX

$$\frac{l = \sigma(\mathbf{point}) \quad v = h(l.f_p)}{\langle \mathbf{point}.f_p,\ \sigma,\ h\rangle \to v}\ [\text{LOAD-P}] \qquad \frac{l = \sigma(n) \quad v = h(l.f_p)}{\langle n.f_p,\ \sigma,\ h\rangle \to v}\ [\text{LOAD-N}]$$

$$\frac{\langle e_1,\ \sigma,\ h\rangle \to v_1 \quad \langle e_2,\ \sigma,\ h\rangle \to v_2 \quad v = v_1 \oplus v_2}{\langle e_1 \oplus e_2,\ \sigma,\ h\rangle \to v}\ [\text{BINOP}]$$

$$\langle \textsf{skip},\ \sigma,\ h,\ \textsf{F}\rangle \to \langle \sigma,\ h,\ \textsf{F}\rangle\ [\text{SKIP}] \qquad \langle \textsf{return},\ \sigma,\ h,\ \textsf{F}\rangle \to \langle \sigma,\ h,\ \textsf{T}\rangle\ [\text{RETURN}]$$

$$\frac{\langle s_1,\ \sigma,\ h,\ \textsf{F}\rangle \to \langle \sigma',\ h',\ \textsf{T}\rangle}{\langle s_1; s_2,\ \sigma,\ h,\ \textsf{F}\rangle \to \langle \sigma',\ h',\ \textsf{T}\rangle}\ [\text{SEQ-RET}]$$

$$\frac{\langle s_1,\ \sigma,\ h,\ \textsf{F}\rangle \to \langle \sigma',\ h',\ \textsf{F}\rangle \qquad \langle s_2,\ \sigma',\ h',\ \textsf{F}\rangle \to \langle \sigma'',\ h'',\ \rho\rangle}{\langle s_1; s_2,\ \sigma,\ h,\ \textsf{F}\rangle \to \langle \sigma'',\ h'',\ \rho\rangle}\ [\text{SEQ-CONT}]$$

$$\frac{\langle e,\ \sigma,\ h\rangle \to v \qquad l = \sigma(\mathbf{point})}{\langle \mathbf{point}.f_p := e,\ \sigma,\ h,\ \textsf{F}\rangle \to \langle \sigma,\ h[l.f_p \mapsto v],\ \textsf{F}\rangle}\ [\text{STORE-P}]$$

$$\frac{\langle e,\ \sigma,\ h\rangle \to v \qquad l = \sigma(n)}{\langle n.f_p := e,\ \sigma,\ h,\ \textsf{F}\rangle \to \langle \sigma,\ h[l.f_p \mapsto v],\ \textsf{F}\rangle}\ [\text{STORE-N}]$$

$$\frac{l_1 = \sigma(n_2) \qquad l_2 = h(l_1.f_r)}{\langle n_1 := n_2.f_r,\ \sigma,\ h,\ \textsf{F}\rangle \to \langle \sigma[n_1 \mapsto l_2],\ h,\ \textsf{F}\rangle}\ [\text{DEF-N}]$$

$$\frac{l = \sigma(n)}{\langle n.f_r := \textsf{alloc},\ \sigma,\ h,\ \textsf{F}\rangle \to \langle \sigma,\ h[l.f_r \mapsto \mathit{fresh}],\ \textsf{F}\rangle}\ [\text{ALLOC}]$$

$$\frac{\langle \mathit{bexp},\ \sigma,\ h\rangle \to \textsf{T}\langle s_1,\ \sigma,\ h,\ \textsf{F}\rangle \to \langle \sigma',\ h',\ \rho'\rangle}{\langle \textsf{if}\ \mathit{bexp}\ \textsf{then}\ s_1\ \textsf{else}\ s_2,\ \sigma,\ h,\ \textsf{F}\rangle \to \langle \sigma',\ h',\ \rho'\rangle}\ [\text{IF-T}]$$

$$\frac{l = h(\sigma(\mathbf{root}).f_r)\langle p,\ \sigma[\mathbf{root} \mapsto l],\ h,\ \textsf{F}\rangle \to \langle \sigma',\ h',\ \rho\rangle}{\langle \mathbf{recurse}\ (\mathbf{root}.f_r, \mathbf{point}),\ \sigma,\ h,\ \textsf{F}\rangle \to \langle \sigma,\ h',\ \textsf{F}\rangle}\ [\text{CALL}]$$

Fig. 1.: Concrete semantics for traversal

A. CONCRETE SEMANTICS FOR SPECIFICATION LANGUAGE

Figure 1 gives a subset of the concrete semantics for performing a traversal; the rules not shown follow the same pattern. The state at the beginning of a traversal is determined by the invocation of *recurse* by the frame program: $\langle p,\ \sigma[\mathbf{root} \mapsto \mathbf{tree}, \mathbf{point} \mapsto p],\ h,\ \mathsf{F}\rangle$, where p is a reference to the current point performing the traversal, and **root** starts out mapped to **tree**, the root of the tree structure (which resides in the heap). Assume that the tree structure has been initialized prior to beginning traversal. All other local variables are initialized to 0 or **null** as appropriate.

SKIP has standard semantics, leaving the store and heap untouched. RETURN changes the return flag to **T**. This flag is checked during statement sequencing (SEQ-RET and SEQ-CONT); if the first statement returns **T**, the second statement does not execute. IF-T has standard semantics, executing the true branch of the if statement; the semantics for the false branch are analogous. STORE-P stores the result into the appropriate point structure in the heap (looking up the heap location using σ).

Accessing tree nodes follows a similar pattern. DEF-N extracts the heap location pointed to by $n_2.f_r$, and maps n_1 to it. STORE-N dereferences n to update the primitive field of the appropriate tree node. ALLOC is similar to STORE-N, except that it updates the appropriate *recursive* field in the heap to point to a freshly-allocated tree node (with recursive fields initialized to **null** and primitive fields initialized to 0). The semantics for assigning null to a tree node's recursive field are similar.

Expressions have standard semantics. The rules for loading from **point** and references are shown. Loading from **point** requires looking up which point structure is referenced in the store, then loading the appropriate field from the heap. Loading

from a reference loops up the appropriate location in the store. Binary operations combine the results of their operands as expected.

The semantics of calls are relatively straightforward. The method body is re-executed with a new store, where **root** is remapped to the canonical access path the recursive call is invoked on and **point** retains the same mapping as the original store. Note that local variables are not remapped; however, because programs assumed to be well-formed, these variables will be re-initialized before being used. After the call returns, execution continues with the old store (thus returning to the old mapping for **root**), but the updated heap. Note, also, that the return flag of the call is always reset to **F**; if calls are sequenced, all calls execute, following the semantics of SEQ-CONT.