TECHNIQUES FOR AUTOMATIC FUSION OF GENERAL TREE TRAVERSALS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Laith S. Sakka

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2020

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Milind Kulkarni, Chair

School of Electrical and Computer Engineering

Dr. Samuel Midkiff

School of Electrical and Computer Engineering

Dr. Xiaokang Qiu

School of Electrical and Computer Engineering

Dr. Tiark Rompf

School of Computer Science

Approved by:

Dr. Dimitrios Peroulis

Head of the School Graduate Program

ACKNOWLEDGMENTS

I would like to thank my advisor Milind Kulkarni for supporting me during my study and being a great mentor. A special thanks also to my lab-mate Kirshanthan Sundararajah for the long conversations we had on discussing several parts of this thesis.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Sakka, L. Ph.D., Purdue University, May 2020. Techniques for Automatic Fusion of General Tree Traversals. Major Professor: Milind Kulkarni.

Trees are common data structures that are used in many programs and applications. In its simplest form, a binary tree can be used to store numbers in sorted manners. Kd-trees, render trees and abstract syntax trees are more sophisticated examples of tree structures. Furthermore, in functional programming algebraic data types are essentially tree structures as well.

In several tree-based applications, a tree is constructed, and several traversals traverse the tree to perform different computations. Tree fusion is a transformation that targets combining and *fusing* different traversals that traverse the same tree and perform them together (ideally in one traversal). Traversal fusion has several performance benefits such as reducing the traversing overhead and the memory accesses, enhancing locality, and eliminating intermediate structures.

Previous work has been done on fusion and was mostly successful either in specific domains or limited scopes. This work introduces novel techniques for performing fusion in both imperative and functional programming settings with a focus on generality. The new techniques target general traversals; minimizing the burden on programmers and increasing the coverage of the transformation. Furthermore, it exploits fusion opportunities that previous approaches do not, achieving significant speedups for a wider range of programs.

1. INTRODUCTION

Many applications are built around traversals of tree structures: from compilers, where abstract syntax trees represent the syntactic structure of a program and traversals of those ASTs are used to analyze and optimize code; to web browsers and layout engines, where render trees express the structure of documents and traversals of those trees determine the location and appearance of elements on web pages and documents; to solving integral and differential equation of multi-dimensional spatial functions where kd-trees are used to represent piece-wise functions and operations on those functions are implemented as tree traversals. On top of that, in a functional programming setting, an algebraic data type is essentially a tree, and functions that consume it are tree traversals.

An important consideration in these applications is to reduce the number of times the tree is traversed by *fusing* operations from separate traversals that operate on the same tree. Such fusion has numerous potential benefits; reducing the number of memory accesses and call instructions, better locality and reduction in cache misses [1], and elimination of intermediate structures in functional settings, are examples of such benefits.

There is a fundamental tension between writing trees traversing applications in the most ergonomic manner—where, for example, a compiler is written as dozens of individual AST-rewriting passes [2, 3]—and writing these applications in the most performant manner—where many AST traversals must be *fused* into a single "or fewer" traversals. In an attempt to balance these competing concerns, there has been prior work on compiler and software-engineering techniques for writing simple, fine-grained tree traversal passes that are then automatically *fused* into coarse-grained passes for performance reasons [3–7]. In the world of functional programs, *deforestation* techniques rewrite data structure operations to avoid materializing intermediate

data structures, either through syntactic rewrites [8–10] or through the use of special combinators that promote fusion [11–15]. Several domain specific fusion frameworks were introduced.

1.1 Contributions Overview

This work introduces novel techniques for performing fusion in both imperative and functional programming settings with a focus on generality. The new techniques target general traversals; minimizing the burden on programmers and increasing the coverage of the transformation. Furthermore, it exploits fusion opportunities that previous approaches do not, achieving significant speedups for a wider range of programs.

Although fusion has the same goal in both imperative and functional settings (combine work from different traversals), each setting has its own challenges.

Imperative Settings. In imperative settings, statements can write and read the same memory locations creating a complicated dependence structure. A fusion transformation should preserve those dependencies and should be able to efficiently find a fused schedule that maximizes fusion under such constraints. Chapters 2 and 3 discuss the part of the thesis that is related to fusing imperative traversals. Chapter 2 introduces *Treefuser* [16] a new fusion approach that is the first to handle general traversals, and performs partial fusion where traversals are fused on some parts of the traversed trees but not others.

Chapter 3 introduces *Grafter* [17] an extension of Treefuser that handles heterogeneous trees with low fusion overhead, it introduces type-based partial fusion and supports more general and complicated patterns of traversals. To sum up, this work introduces a novel framework for fusing general imperative traversals that have shown to have a very small overhead and succeeded in fusing general and non-trivial traversals achieving significant speedups even for even small trees.

Functional Settings. Unlike imperative traversals where traversals traverse the same structure. A functional traversal consumes a tree and produces a new tree that is consumed by the following traversals. Most performance benefits from fusing functional traversal aka as *deforestation* comes from eliminating intermediate structures. The main challenge in the general fusion of functional programs is preserving the run time complexity and guarantee the transformation termination.

Chapter 4 discuss the challenges in general fusion in functional settings, and propose a deforestation algorithm that consists of fusion, tupling and intensive redundancy analysis that can handle fusing complicated and irregular traversals such as render tree traversals. The transformation shows that general fusion techniques for functional settings that have been abandoned for a long time are promising and that—with good engineering—can fuse complicated programs that cannot be fused using competing techniques.

2. TREEFUSER : ANALYZING AND FUSING GENERAL RECURSIVE TREE TRAVERSALS

This chapter describes the first piece of work in the thesis [16]; a general method for reasoning about and fusing general imperative recursive tree traversals. The proposed framework is the first to handle fusion of general traversals. Furthermore, it introduces *partial fusion*; which allows finer-grained fusion when total fusion of traversals is not possible.

2.1 Introduction

Many key applications rely heavily on performing a series of tree traversals. For example, the DOM (Document The previous chapter Object Model) of a website is represented as a tree of elements, where each element contains a number of attributes specifying color, visibility, position, size, etc. The values of these attributes determine the appearance of a rendered web page. These attributes and elements are dependent on one another (for example, the width of an element is dependent on the widths and x-positions of its children elements), so computing the appropriate values requires traversing the tree numerous times [18]. Closer to home, many compilers use Abstract Syntax Trees (ASTs) as an intermediate representation, and compilation phases are expressed as traversals over these ASTs. Again, these traversals read and modify various attributes of the AST, and are dependent on one another.

An important consideration in these applications is to reduce the number of times the tree is traversed by *fusing* operations from separate traversals that operate on the same node. Such fusion has significant performance benefits as discussed in the thesis introduction.

In many cases, such as web browsers, what are conceptually multiple traversals of the tree are *manually* fused into fewer traversals. In other cases, traversals can be expressed using high level abstractions like attribute grammars (for compilers [19] or DOM rendering [6]) or numerical operators (in MADNESS [20]) that can be fused using purpose-built frameworks that understand those abstractions (e.g., synthesis-based schedulers for attribute grammars [18] or *ad hoc* high level compilers for MADNESS operators [4, 5]).

These approaches suffer from various drawbacks: immense programmer effort and brittleness in the case of hand-written optimized traversals (adding a new DOM pass might require rewriting a large chunk of the renderer due to newly-introduced dependences), and the restrictions of high level abstractions in the latter (both in the inability of high level abstractions to capture more complicated traversal patterns and in the types of fusion decisions that can be made by the optimization framework).

We want the best of both worlds: the ability to write simple, fine-grained traversals to express the computations required by an application, as in approaches using high level abstractions, while retaining the flexibility to perform more complicated traversal patterns as in hand-written code (e.g., performing computations between recursive calls). When these traversals are optimized and fused, we should retain the same capabilities as in hand-written optimizations. In particular, there are two optimization opportunities that are critical to exploit in order to maximize fusion that, to our knowledge, existing approaches do not explore:

1. *Code motion*: By moving computations around within a traversal, fusion opportunities can be exposed that were previously disallowed. For example, moving a statement in a traversal from pre-order to post-order can allow it to be fused with other traversals. Code motion is especially important with fully-general traversals, where computations can be interleaved with recursive calls, as such in-order computations can introduce very complex dependence patterns.

2. *Partial fusion*: Even if the computations performed by two traversals have complex dependences that preclude fusion, it may be possible to execute the traversals together on *some* parts of the tree but not others yielding some of the benefits of fusion without requiring complete fusion. As in the case of code motion, allowing general, complex traversals necessitates partial fusion, as the more-complicated dependence structure often precludes total fusion of multiple traversals.

This work presents a general method for *traversal fusion* of recursive tree traversal programs written in a general-purpose language. This method provides several novelties: (a) a dependence graph representation of traversals that captures interactions between statements and calls; (b) a dependence test that uses the dependence graph to determine which fusion opportunities are legal and supports partial fusion and code motion; and (c) a synthesis procedure for fused traversals that naturally implements code motion.

We implement our method in a framework called Treefuser, a Clang source-to-source compiler pass, allowing programmers to express their set of tree traversals using (a subset of) C++. The pass automatically lifts the tree traversals to Treefuser's dependence representation, and then uses a simple greedy heuristic to synthesize a new, fused implementation of the set of traversals, including moving computations to remove problematic dependences and performing partial fusion when necessary.

To evaluate the suitability of Treefuser for enabling fine-grained expression of general tree traversals while supporting fusion, we evaluate several case studies: fusing together passes in Fast Multipole Method, fusing passes over a DOM rendering tree, and fusing complex AST passes in a compiler. In all three cases, we demonstrate that Treefuser is able to successfully fuse a substantial portion of the computation, requiring both code motion and partial fusion to do so. The fused traversals substantially improve locality and overall performance.

Outline

The remainder of this chapter is organized as follows. Section 2.2 gives an overview of our technique and presents a running example that highlights the types of transformations enabled by our framework. Section 2.3 presents a simple, but general, language for describing tree traversals, defines a dependence analysis for programs written in this language, and describes a dependence representation for summarizing dependences within and among traversals. Section 2.4 explains how the dependence representation can be used to determine the validity of fusion decisions. Section 2.5 describes how Treefuser synthesizes fused traversals from the dependence representation. Section 2.6 proves the soundness of the fusion transformation. Section 2.7 describes the Treefuser implementation. Section 2.8 evaluates Treefuser, and Section 2.9 concludes.

2.2 Overview

This section provides an overview of the Treefuser framework. We first discuss the opportunities for fusion in a running example (Section 2.2.1). We then describe how Treefuser constructs a dependence representation for the example (Section 2.2.2). We then explain how Treefuser uses the dependence representation to construct a fused traversal, including partial fusion, and show the final synthesized result (Section 2.2.3). Each of these steps are described in more detail in the remainder of the chapter.

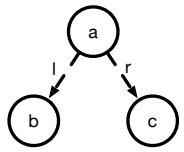
2.2.1 Running Example

Figure 2.1(a) shows two traversals, `f1` and `f2`, over a binary tree (the argument `n` to each traversal represents the node the traversal is currently visiting). While the traversals perform somewhat complex computations, we can consider each as a sequence of statements, where compound statements such as `if` statements are rep-

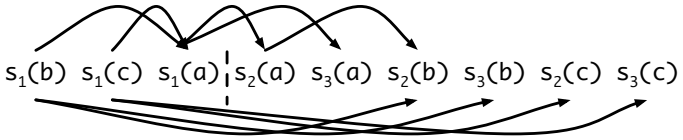
```
f1(n)
  if (n = null) return
  f1(n.l)
  f1(n.r)
  if (n.isLeaf)                S1
    n.v = 1
  else
    n.v = n.l.v + n.r.v
```

```
f2(n)
  if (n = null) return
  if (!n.isLeaf)               S2
    n.x = n.v
    n.l.x = n.x + 2
  n.y = n.v * 2                S3
  f2(n.l)
  f2(n.r)
```

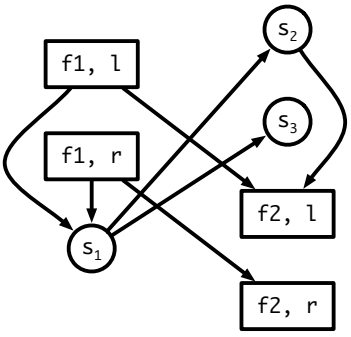
(a) Two traversals of a binary tree



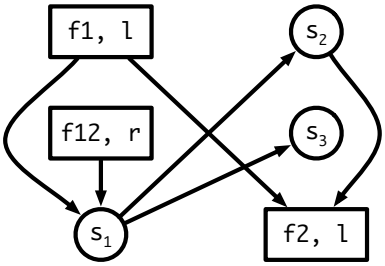
(b) Binary tree with three nodes



(c) Original schedule of execution, with dependences



(d) Dependence graph, pre-fusion



(e) Dependence graph, post-fusion

```
fused(n, do1, do2)
  if (n = null) return
  fused(n.l, do1, false)
  fused(n.r, do1, do2)
  if (do1)
    if (n.isLeaf)              S1
      n.v = 1
    else
      n.v = n.l.v + n.r.v
  if (do2)
    if (!n.isLeaf)             S2
      n.x = n.v
      n.l.x = n.x + 2
    n.y = n.v * 2              S3
  fused(n.l, false, do2)
```

(f) Fused traversals

$s_1(b)\ s_1(c)\ s_2(c)\ s_3(c)\ s_1(a)\ s_2(a)\ s_3(a)\ s_2(b)\ s_3(b)$

(g) Schedule of execution, after fusion

Fig. 2.1. TREEFUSER overview example: fusing two binary tree traversals.

resented as single statements, interspersed with recursive invocations of the traversal function that traverse the tree further. Hence, we can consider `f1` to perform one statement per node during its traversal, and `f2` to perform two, as shown by the gray outlined boxes. (For now, we ignore the `n = null` line of each traversal as that serves simply to bound the traversals; later we will explain how to soundly handle such conditions.)

Suppose the two traversals visit the tree shown in Figure 2.1(b), with `f1` executing before `f2` leading to the sequence of statements shown in Figure 2.1(c). Each statement is parameterized by which node of the tree it operates on, and arrows between statements indicate dependences. For example, $s_1(b)$ writes to `b.v`, which is read by $s_1(a)$, $s_2(b)$ and $s_3(b)$. The vertical line separates the operations performed by the two traversals. As we see, the tree is traversed twice, and, in particular, each leaf node is visited twice.

Opportunity for fusion Careful examination of the dependence structure shows that it may be possible to perform operations from both traversals while visiting some of the nodes in the tree. In particular, all of the operations on node `c` can be performed when the *first* traversal visits node `c` (in other words, $s_2(c)$ and $s_3(c)$ can be hoisted up to execute immediately after $s_1(c)$). As a result, the second traversal need not visit the right child of the tree at all, saving one node visit. Note, however, that the two visits to node `b` must happen separately: s_1 must execute on `b` before executing on `a`, but s_2 must execute on `a` before executing on `b`. The only way to achieve this is by visiting `b` twice. We will now see how Treefuser identifies this opportunity for fusion.

2.2.2 Dependence Representation

The dependencies between the various operations of two traversals is a dynamic property: two statements depend on each other depending on where, in a particular tree, the two statements are being executed. To avoid reasoning about (unknow-

able) dynamic iterations, TREEFUSER uses a *dependence graph* representation that captures dependences between static instances of statements based on a dependence analysis. Section 2.3.3 details how this dependence graph is constructed. In short, each (compound) statement and recursive call in the original traversals is represented by a node in the dependence graph. Statement nodes are labeled by which traversal they are from and which (static) statement in that traversal they represent, while call nodes are labeled by which traversal they are from and by which child that call descends to.

Figure 2.1(d) shows the dependence graph for the statements of the traversals in Figure 2.1(a). The graph considers the execution of each traversal rooted at the same (arbitrary) node in the tree. Directed edges exist between statements if there may be dependences between those statements *when executed at the same node*. Hence, the dependence between $s_1(a)$ and $s_2(a)$ leads to an edge in the dependence graph between the node s_1 and the node s_2. Edges exist between statements and calls if there may be a dependence between the statement and statements that may execute during that call. Hence, the dependence between $s_1(b)$ and $s_1(a)$ leads to an edge between the call node (f_1, l) and the statement node s_1. Similarly, edges exist between calls if there may be a dependence between the statements that may execute during those calls. The direction of the edges corresponds to the execution of the original, unfused program: between two nodes from the same traversal, the edge is directed in program order, and between two nodes from different traversal, the edge is directed according to the order of the traversals. Essentially, the dependence graph in Figure 2.1(d) represents taking the execution graph of Figure 2.1(c) and collapsing the various vertices for nodes `b` and `c` into call nodes representing the corresponding recursive call.

2.2.3 Fusion and Code Generation

Two calls to the same child of a tree node are *fusable* if it is safe for two traversals to visit that child simultaneously (rather than one traversal completely processing that child subtree before the second traversal visits it). Two traversals are *fully fusable* if all of their child calls are fusable. If only some child calls are fusable, the two traversals are *partially fusable*. The dependence graph makes it easy to tell if two calls are fusable: if the two call nodes in the graph can be merged *without creating a cycle*, then the nodes are fusable. Section 2.6 argues that this validity test is sound.

In our running example, we can see that fusing the two calls to the left child results in a cycle involving s_1 and s_2. However, fusing the two calls to the right child does not create a cycle, yielding the dependence graph in Figure 2.1(e).

Given a fused dependence graph, TREEFUSER synthesizes the fused traversal by topologically sorting the dependence graph and then generating code as if the resulting graph were for a single traversal function (including guards to make sure that a statement only executes if its original traversal would have executed). Figure 2.1(f) shows the resulting code. Note that this synthesis procedure automatically incorporates code motion: in the fused, sorted graph, s_3 appears after the traversal visits the right child, whereas in the original traversal, s_3 executed prior to its traversal visiting the right child. Section 2.4 explains this process in more detail.

The final traversal produces the schedule of execution shown in Figure 2.1(g). This new schedule preserves all dependences—dependent operations appear in the same relative order as they did in the schedule of Figure 2.1(c)—but now node `c` is only visited once, providing the benefits of fusion (this fused traversal is valid for *any* tree, and this reduction in visits grows as the tree grows). The remainder of this chapter explains how TREEFUSER achieves this result.

2.3 A Dependence Analysis for Fusion

This section presents a dependence analysis for fusion. It is a variant of access-path-based dependence analyses, such as those used by [21] and [22]. Because we adopt an instance-wise notion of accesses [23] (that is, we consider accesses relative to a particular node that a traversal is visiting), our dependence test is informed by that of [24]. First, we present a simple (but general) language for expressing tree traversals that we will use to build our dependence analysis.

2.3.1 A Language for Tree Traversals

A tree that is being traversed consists of several *tree nodes*. Each tree node contains one or more *recursive fields* ($f_r \in \mathcal{F}_r$) that point to other tree nodes (we will often refer to these recursive fields as the "children" of the tree node), and zero or more *local fields* ($f_p \in \mathcal{F}_p$). These local fields can be simple primitive values or can be pointers to *node local* objects (objects that can only be reached through that local field). The nodes are arranged in a tree, connected through the recursive fields, and rooted at a single **root** node[1].

A tree is traversed by *recursive traversal functions* whose bodies can be expressed in the language given in Figure 2.2(a)[2]. Each function is k-ary: the first parameter of the function is a reference to the **node** the traversal operates on; the rest of the parameters are local variables.

The body of the traversal function is a series of (compound) statements and recursive calls. Each statement can be a single primitive statement (an assignment or a return) or an if statement that can then be arbitrarily nested. Expressions can de-reference fields of **node** and read local variables, and can invoke pure functions. Assignments can update local variables and primitive fields of **node** (note that the

[1]Note that our analysis does not concern itself with proving these properties of the tree; we leave this to shape analyses [25, 26]. The implementation in Section 2.7 relies on programmer- or shape analysis–provided annotations to convey these properties.

[2]TREEFUSER analyzes programs written in C++, but that respect the constraints of this language

$$v \in \mathit{Values} ::= \mathbb{Z} \qquad l \in \mathit{Locals} ::= l_1 \mid l_2 \mid \ldots$$
$$n \in \mathit{NodeRefs} ::= \textbf{node} \mid n_1 \mid n_2 \mid \ldots$$
$$\oplus ::= + \mid - \mid \times \mid \div$$
$$\odot ::= < \mid > \mid = \mid \neq \mid \geq \mid \leq$$
$$\mathit{bexp} \in \mathit{BExprs} ::= n.f_r = \textbf{null} \mid n.f_r \neq \textbf{null} \mid e \odot e$$
$$e \in \mathit{Exprs} ::= n.f_p \mid l \mid e \oplus e \mid v \mid f(e_1, e_2, \ldots)$$
$$s \in \mathit{Stmts} ::= \textbf{return} \mid \textbf{if } \mathit{bexp} \textbf{ then } s^* \textbf{ else } s^*$$
$$\mid n := n \mid n := n.f_r \mid l := e \mid n.f_p := e$$
$$c \in \mathit{Calls} ::= \textbf{traverse } (\textbf{node}.f_r, l_1, l_2, \ldots)$$
$$p \in \mathit{Body} ::= (s \mid c)^*$$

(a) Language for tree traversal

$\mathbf{f}_1\ (\textbf{root}, l_1, l_2)$
$\mathbf{f}_2\ (\textbf{root}, l_3, l_1)$
$\mathbf{f}_3\ (\textbf{root}, l_4, l_5)$

(b) Language for program

Fig. 2.2. TREEFUSER language

recursive fields of **node** cannot be modified). A recursive call takes one of the recursive fields of **node** as its first argument and local variables as its remaining arguments (we will often refer to these recursive calls as "child calls," as they are invoked on specific children of the current **node**). A few things to note about this language:

1. We do not allow loops in the body of the recursive function (unless those loops can be statically unrolled).

2. Recursive calls do not return values; return values can be de-sugared as writing to a primitive field of **node** in the callee, then reading from that primitive field (via the appropriate recursive field) in the caller.

3. Recursive calls cannot be made conditionally. This can be worked around by conditionally setting a local variable that is passed into the recursive call that causes the call to exit immediately.

4. Interestingly, the recursive invocations of the traversal functions: a) do not have to be invoked on every child, and b) can be invoked on a child more than once.

In other words, a recursive traversal function may not visit all children of a node, and may visit the same subtree multiple times.

A program analyzed by TREEFUSER is a sequence of calls to traversal functions defined using the language of Figure 2.2(a), all starting at **root** (Figure 2.2(b)). We do not consider the existence of intervening code between successive calls to traversal functions; handling any dependences that arise from such code is a straightforward extension of the analyses in this work.

Handling different child types The language of Figure 2.2(a) assumes that all tree nodes are the same type. This assumption is violated by many tree traversal codes (*e.g.*, the multitude of node types in a typical AST). We currently handle this case by merging all the node types into a single node type (a la a tagged union), and using conditionals to distinguish between the code that should run in each node type. If a recursive call is made to a child that does not exist for the current node type, that call sees that the child node is **null** and returns immediately. This transformation can be done automatically, though it reduces the precision of our analysis. Chapter 3, discusses this problem in more details and provides a solution that efficiently handles heterogeneous trees.

2.3.2 Access-path Analysis

The first step in TREEFUSER's dependence analysis is to perform an analysis to construct *access sets* for each compound statement and call in each recursive traversal function.

We abstract the "off-tree" locations that can be accessed by functions as a set of heap locations $h \in \mathcal{H}$—these correspond to whatever representation a (separately run) alias analysis uses to represent heap locations. We represent accesses to the nodes and fields of the tree using *access paths*. A *recursive access path* is a path to a node rooted at **node**. These access paths can be represented by elements of

the regular set $\mathcal{A}_r = \mathbf{node}(.\mathcal{F}_r)^*$. A *primitive access path* is a path rooted at **node** terminating in a primitive field, and can be represented by elements of the regular set $\mathcal{A}_p = \mathbf{node}(.\mathcal{F}_r)^*.\mathcal{F}_p$.

Statements Each compound statement s is associated with four sets: a set of heap locations read and written ($H_r(s)$ and $H_w(s)$ respectively) and a set of primitive access paths read and written ($\pi_r(s)$ and $\pi_w(s)$ respectively). Heap locations read and written by a statement can be computed via a straightforward dataflow analysis on the body of the function, treating function calls as no-ops. Unioning together all of the reads and writes of the component statements of a compound statement yields $H_r(s)$ and $H_w(s)$.

Generating $\pi_r(s)$ and $\pi_w(s)$ is similar to other analyses [21, 24, 27], so we treat it briefly here (essentially, we perform an alias analysis on node references, using an access-path-based representation). Access paths read and written are computed by an alias analysis that tracks the (set of) recursive access paths referred to by node references. Assignments to node references generate new recursive access paths for that node reference in the obvious way (maintaining multiple possible access paths for the left hand side of the assignment if the right hand side maps to multiple paths). Assignments to primitive fields of a node reference create *write* access paths by appending the primitive field to the access path(s) of the node reference, and uses of primitive fields in expressions create *read* access paths by appending the primitive field to the access path of the node reference. Conditionals in **if** statements can generate read accesses, but are otherwise not evaluated; the analysis processes both branches of the conditional. Access paths associated with node references are joined in **if** statements using set union. For a compound statement s, $\pi_r(s)$ is the union of all the read access paths in the components of the compound statement, and $\pi_w(s)$ is the union of all the write access paths.

Calls Each call c contains summaries of all the reads and writes that could be performed during the invocation of the call (including further recursive invocations).

Summarizing the heap accesses is straightforward. A call's heap read and write sets ($H_r(c)$ and $H_w(c)$) are the unhttps://www.overleaf.com/project/5cbd2fe803073944b64c030eion of the method body's statement's heap read and write sets. Note that because heap locations cannot be accessed via fields of the tree, the set of possible heap locations accessed by the recursive invocation is exactly the same as the caller's.

Summarizing a call's accesses to the tree is subtle. Because a recursive call can make additional recursive calls, each access to the tree can repeat, each time being rooted at a different node. To capture this information, we define an *extended access path* (this follows the approach of [21]). Rather than being a member of the regular set $\mathcal{A}_p$, an extended access path is, itself, a regular set.

To construct the extended access paths for a particular call, **traverse** (**node**.$r_1, \dots$), we first construct the *base* read (respectively, write) set of access paths by unioning all the read (respectively, write) access path sets of the statements in the method body. Consider a specific primitive access path in the read set: **node**.$r_a.r_b.p_a$. The extended access path for this path for this call is: **node**.$r_1(.\mathcal{F}_r)^*.r_a.r_b.p_a$. In other words, we take the suffix of the access path and insert a regular set consisting of the child that the recursive call descends to (which represents the root of the access paths in the immediate callee) followed by zero or more additional children (which represents any subsequent recursive calls made by the method)[3]. By constructing extended access paths for each base access path for a call, we can obtain a set of regular sets whose union (over) approximates all of the fields of the tree that recursive call might access.

[3]This is made more precise by using the subset of recursive fields that are actually visited by the traversal function in question, rather than $\mathcal{F}_r$

Running example In our running example (Figure 2.1(a)), there are no heap accesses, so we need to only consider tree access paths. We can obtain the following access paths for the statements (note that in our example, `n` represents **node**):

$$\pi_r(s_1) = \{n.l.v, n.r.v\} \qquad \pi_w(s_1) = \{n.v\}$$
$$\pi_r(s_2) = \{n.v, n.x\} \qquad \pi_w(s_2) = \{n.x, n.l.x\}$$
$$\pi_r(s_3) = \{n.v\} \qquad \pi_w(s_3) = \{n.y\}$$

The extended access paths for the calls are:

$$\pi_r(f_1(n.l)) = \{n.l(.(l|r))^*.l.v, n.l(.(l|r))^*.r.v\}$$
$$\pi_w(f_1(n.l)) = \{n.l(.(l|r))^*.v\}$$
$$\pi_r(f_1(n.r)) = \{n.r(.(l|r))^*.l.v, n.r(.(l|r))^*.r.v\}$$
$$\pi_w(f_1(n.r)) = \{n.r(.(l|r))^*.v\}$$
$$\pi_r(f_2(n.l)) = \{n.l(.(l|r))^*.v, n.l(.(l|r))^*.x\}$$
$$\pi_w(f_2(n.l)) = \{n.l(.(l|r))^*.x, n.l(.(l|r))^*.l.x, n.l(.(l|r))^*.y\}$$
$$\pi_r(f_2(n.r)) = \{n.r(.(l|r))^*.v, n.r(.(l|r))^*.x\}$$
$$\pi_w(f_2(n.r)) = \{n.r(.(l|r))^*.x, n.r(.(l|r))^*.l.x, n.r(.(l|r))^*.y\}$$

2.3.3 Building a Dependence Graph

We are now prepared to build the dependence graph to capture dependences between statements and calls, which can help determine when fusion is legal. The key to the dependence graph is that it captures the execution of every statement and call across the set of traversals, T, considered for fusion *assuming they are executing at the same node in the tree*. This is because the goal of fusion is to merge computations at the same node of the tree together so that the two traversals can occur simultaneously.

The dependence graph is a directed graph $G = (V, E)$. The graph contains a vertex for each (compound) statement and call in the set of traversal functions T. Every

vertex is labeled using a labeling function $l : V \to T \times (\{s\} \cup \mathcal{F}_r)$. In other words, a vertex is labeled with a tuple whose first element is the traversal function the vertex is from, and whose second element denotes whether the vertex is a statement (labeled s) or a call (labeled with the recursive field of **node** that the call descends to).

The edges in the graph are directed, and capture data and control dependences between statements and calls, both within traversals and across traversals.

For every pair of vertices v_1 and v_2, where v_1 appears before v_2 in program order (either they are in the same traversal and v_1 is earlier in the traversal, or v_1 appears in an earlier traversal than v_2), $(v_1, v_2) \in E$ iff one or more of the following is true:

1. v_1 and v_2 are part of the same traversal and there is a dependence through local variables between v_1 and v_2. This can be established through a straightforward intra-procedural reaching-definitions analysis.

2. v_1 and v_2 are part of the same traversal and v_1 is a compound statement that contains a **return** statement. In this case, v_2 is *control dependent* on v_1, as if v_1 returns, v_2 will not execute. On the other hand, if v_2 contains a **return** statement, there will be a control dependence from v_1 to v_2 to ensure that v_1 *does* execute.

3. $H_w(v_1) \cap (H_r(v_2) \cup H_w(v_2)) \neq \varnothing$ or $H_w(v_2) \cap (H_r(v_1) \cup H_w(v_1)) \neq \varnothing$. In other words, the two statements or calls access the same heap location, and at least one of them is a write.

4. v_1 and v_2 are both statements, and $\pi_w(v_1) \cap (\pi_r(v_2) \cup \pi_w(v_2)) \neq \varnothing$ or $\pi_w(v_2) \cap (\pi_r(v_1) \cup \pi_w(v_1)) \neq \varnothing$. That is, if v_1 and v_2 are both statements, there is an edge between them if they share an access path, with at least one of those accesses being a write.

5. v_1 (resp. v_2) is a call and its extended access path(s) *collide* with the (extended) access paths of v_2 (resp. v_1). The notion of collision is subtle. Assume, without loss of generality, that v_1 is a call. For each extended access path $a_1 \in \pi_w(v_1)$,

we intersect a_1 with each (extended) access path in $\pi_r(v_2) \cup \pi_w(v_2)$[4], and test for the emptiness of the resulting regular set. If any such set is non-empty, then v_1 collides with v_2. We can perform a similar procedure to check for the scenario where v_2 performs the write and v_1 performs any colliding access.

The resulting dependence graph captures the ordering between statements and calls that must be respected by any attempted traversal fusion: if two statements are to execute at the same node in the tree at the same time, and there is a dependence edge between them, they must execute in an order that respects that dependence. Note that the dependence graph is guaranteed to be acyclic (since the edges are directed in program order).

Running example Figure 2.1(d) shows the dependence graph for our running example, leaving out the null test at the beginning of each traversal. Each null test results in control dependences between that test and all of the other statements/calls in its traversal.

2.4 Traversal Fusion

This section explains how Treefuser uses the dependence graph to determine whether fusion is legal.

2.4.1 Traversal Fusion as Child-call Fusion

Fusing together two traversals means executing operations from multiple traversals while visiting a node of a tree, rather than visiting the node separately for each traversal. We can phrase the problem of fusing traversals as a problem of fusing child calls. Suppose two traversals start at the same node (*e.g.*, the root node). If we can combine recursive calls to the same child from the two traversals, then a single visit to that child will accomplish the work of both traversals (note that this does not mean

[4]If v_1 is a statement, we can treat its access paths as singleton regular sets.

the total number of visits is cut in half; after descending to the child, the traversals may diverge upon further recursive calls).

Viewing fusion as fusing child calls is a generalization of "total" fusion. Suppose we have two traversals, f_1 and f_2 that operate over a binary tree. Hence, each traversal has two recursive calls, one visiting **node**.l and another visiting **node**.r. If the two traversals' calls to **node**.l are fused and their calls to **node**.r are fused, then we have achieved total fusion: the fused function will traverse the tree just once, and at each child call will do the work of both f_1 and f_2. If only some of the children are fused, then we have achieved "partial" fusion. In Figure 2.1(f), the calls to the right child are fused together, but the calls to the left child are not. If both traversals are at the same node, they will both descend to the right child together, but will process the left child separately. In general, then, the more recursive calls we can fuse together, the fewer times we will visit nodes in the tree. Indeed, TREEFUSER's goal is not to "fuse traversals." Instead, its goal is to fuse together multiple recursive calls.

2.4.2 Dependence Test and Fusion

TREEFUSER constructs a dependence graph as outlined in Section 2.3.3. This dependence graph can then be used to test whether two calls can be fused together.

First, we slightly modify the labeling function for the graph. Rather than mapping vertices to traversals and types ($l : V \to T \times (\{s\} \cup \mathcal{F}_r)$), the labeling function maps vertices to *sets* of traversals and types ($l : V \to \mathcal{P}(T) \times (\{s\} \cup \mathcal{F}_r)$). The initial dependence graph's vertices are all labeled with single-element traversal sets.

Then, we define an operation called *fuse*, which takes an acyclic graph, $G = (V, E)$, and two call vertices, $v_1, v_2 \in V$, where $l(v_1) = (T_1, r_1)$ and $l(v_2) = (T_2, r_1)$ (i.e., both vertices are calls that descend to the same child). $fuse(G, v_1, v_2)$ produces a new graph $G' = (V', E')$ with:

V' is V with v_1 and v_2 removed, and a new vertex v_{12} added, where $l(v_{12}) = (T_1 \cup T_2, r_1)$

E' retains all edges in E that are not incident on either v_1 or v_2, *removes* any edge between v_1 and v_2, and for edges incident on v_1 (or v_2), replaces v_1 (resp. v_2) with v_{12}.

G' represents fusing the calls v_1 and v_2. We can define *fusability* as follows:

Definition 2.4.1 *Two calls, v_1, where $l(v_1) = (T_1, r_1)$, and v_2, where $l(v_2) = (T_2, r_1)$, are* fusable *iff $fuse(G, v_1, v_2)$ is acyclic.*

The soundness proof for this definition is in Section 2.6, but the intuition is as follows. Dependence edges between vertices in G require that the vertices execute in that order to preserve either data or control dependence. Merging together two call vertices (i.e., merging together two calls) ensures that those calls fwill happen simultaneously, so the two visits to the child will occur at the same time. Any other vertices that are dependent on either of those two calls will still need to respect their original dependences, and if there is a cycle in the merged graph, there is no program order that can respect all dependences.

Note that fusion does not require that the two child calls fused together come from different traversals. In some settings, it may be more natural to conceive of a traversal operation as visiting the same subtree multiple times, even though it is possible to implement the operation with a single visit to a subtree. In such cases, a programmer can write the "multiple subtree" version of the traversal and rely on fusion to merge the subtree visits together.

Fusing together two child calls produces G', and the process of fusing together further child calls can be repeated recursively. The operation *fuse* is the fusion mechanism; choosing which child calls to apply it to is a policy decision (see Section 2.7) and different fusion policies might result in different fused graphs.

Running example Figure 2.1(d) is the dependence graph for our running example. There are two possible options for child fusion here: the two calls to the left child (labeled (f_1, l) and (f_2, l)), and the two calls to the right child (labeled (f_1, r) and

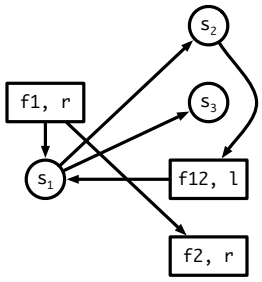


Fig. 2.3. Invalid dependence graph, resulting from attempted fusion of left children in Figure 2.1(d).

(f_2, r)). Figure 2.1(e) shows the result of fusing the two right children, producing a new merged node (f_{12}, r). Here we see that there are no cycles, so this is a valid fusion option. Had we tried to fuse the two left children, we would have obtained the dependence graph in Figure 2.3, producing a cycle, which is an invalid fusion. Intuitively, this is invalid because the first traversal needs to visit its left child before computing `n.v = n.l.v + n.r.v`, while the second traversal needs to compute `n.l.x = n.x + 2` before visiting its left child. If both traversals descend to the left child at the same time, one traversal will be unable to perform its computations at the correct time.

2.5 Synthesizing a Fused Traversal

This section describes how TREEFUSER synthesizes a tree traversal from a dependence graph, taking into account any fused child calls. The key is generating a *single* tree traversal function, regardless of how many original traversals are represented in the dependence graph, and regardless of how many child calls are fused. We begin by presenting a basic traversal template that generates a valid single traversal for the *unfused* dependence graph (Section 2.5.1). We then show how this same template can be modified to generate traversals for merged children (Section 2.5.2). Finally, we explain how this synthesis procedure naturally and implicitly incorporates any required code motion (Section 2.5.3).

Note that while fusion is a valid operation on *any* recursive calls to the same child, whether from different traversals or from the same traversal, the synthesis procedure presented here only applies to fusion of calls from different traversals.

2.5.1 Traversal Template

TREEFUSER generates a single traversal function from an (unfused) dependence graph through rewrite rules. Consider a dependence graph generated from a set of traversals $T = \{t_1, t_2, \dots\}$. The signature of the new traversal function is:

$$\textbf{traverse}_f\ (\textbf{node}, o_1, o_2, \dots, b_1, b_2, \dots)$$

Where o_1, o_2, etc., are pointers to (local) structures that contain fields corresponding to the local variable parameters of traversals t_1, t_2, etc. (assume, without loss of generality, that the names of the local variables in all of the traversals are unique).[5] The variables b_1, b_2, etc., are boolean variables corresponding to each traversal. Intuitively, b_1, b_2, etc., determine whether a particular traversal is "active" during a given visit to a node (i.e., during a given execution of the traversal function). The overall traversal is called by initializing the o objects to the appropriate local variables, calling **traverse** on the root node, and passing in **true** for all of the boolean variables (i.e., at the root, all traversals are active).

The next step is filling in the body of the traversal function. The first part of the traversal function extracts all the local variables from (non-**null**) objects passed into $\textbf{traverse}_f$ and stores them in local variables with the same names as in the original traversals.

TREEFUSER then generates a topological sort of the dependence graph to determine the order statements/calls should appear in the body. It then inserts the statements and calls into the body one by one.

[5] We use this level of indirection, rather than directly passing local variables, because not all local variables might exist at a given call site, as we will see later. In such cases, we pass **null** for the relevant parameter.

Statements For each statement s_i with vertex label $(\{t_j\}, s)$, TREEFUSER generates the following code:

$$\textbf{if } (b_j = \textbf{true}) \textbf{ then } s_i'$$
$$\textbf{label } i:$$

where s_i' is s_i with all **return** statements replaced with

$$b_j = \textbf{false}; \textbf{goto } i$$

Recall that b_j is the variable that determines whether traversal t_j is meant to execute during this visit to a node. Hence, s_i, which comes from traversal t_j should only execute if b_j is **true**. Moreover, if s_i calls **return**, the traversal should truncate at this tree node and not descend to any further children. This is achieved by having the traversal turn itself off by setting b_j to **false**, and then skip over the remainder of s_i (with the **goto** statement).

Calls For each recursive call vertex label $(\{t_j\}, f_r)$, (i.e., a recursive call that looks like:
traverse $(\textbf{node}.f_r, l_{j_1}, l_{j_2}, \dots))$, TREEFUSER generates the following code:

$$\begin{aligned}
&\textbf{if}(b_j = \textbf{true}) \textbf{ then} \\
&\quad o_j.l_{j_1} = l_{j_1}; o_j.l_{j_2} = l_{j_2}; \dots \\
&\quad \textbf{traverse}_f\ (\textbf{node}.f_r, \textbf{null}, \dots, o_j, \dots, \textbf{false}, \dots, b_j, \dots)
\end{aligned}$$

In other words, if traversal t_j is still active by the time it gets to the call, it initializes o_j with the necessary parameters to execute traversal t_j, and passes in **null** for all the other objects. It then passes b_j into the recursive call (signifying that if t_j is st ill active, it should continue to child f_r), and passes **false** for all the other boolean variables (signifying that, for this call, no other traversal is active).

Running example Figure 2.4 shows the result of using this transformation to produce a single traversal covering `f1` and `f2` from our running example. Note that

```
f(n, b1, b2)
  if (b1) if (n = null)
    b1 = false; goto l1
l1:
  if (b1) f1(n.l, b1, false)
  if (b1) f1(n.r, b1, false)
  if (b1)                          S1
    if (n.isLeaf)
      n.v = 1
    else
      n.v = n.l.v + n.r.v
  if (b2) if (n = null)
    b2 = false; goto l2;
l2:
  if (b2)                          S2
    if (!n.isLeaf)
      n.x = n.v
      n.l.x = n.x + 2
  if (b2)                          S3
    n.y = n.v * 2
  if (b2) f2(n.l, false, b2)
  if (b2) f2(n.r, false, b2)
```

Fig. 2.4. Single traversal function generated for running example by TREEFUSER.

this fused traversal clearly performs exactly the same work as the original traversals—indeed, it generates exactly the same schedule of computation as in Figure 2.1(c).

2.5.2 Generating Code for Fused Calls

So what happens if the dependence graph we are synthesizing code from contains fused calls? Recall that when a vertex in the dependence graph represents a fused call, its label captures the *set* of traversals that the call belongs to. Consider generating code for a fused call with label $(\{t_i, t_j\}, f_r)$ (a call to child f_r from traversals t_i and t_j). The generated code for this call is:

$$\textbf{if}(b_i = \textbf{true} \vee b_j = \textbf{true})\ \textbf{then}$$
$$\quad \textbf{if}\ (b_i = true)\ \textbf{then}\ o_i.l_{i_1} = l_{i_1}; \ldots\ \textbf{else}\ o_i = \textbf{null}$$
$$\quad \textbf{if}\ (b_j = true)\ \textbf{then}\ o_j.l_{j_1} = l_{j_1}; \ldots\ \textbf{else}\ o_j = \textbf{null}$$
$$\quad \textbf{traverse}_f\ (\textbf{node}.f_r, \textbf{null}, \ldots, o_i, o_j, \ldots, \textbf{false}, \ldots, b_i, b_j, \ldots)$$

The fused call occurs when *either* of the two traversals are active. The call takes in both sets of local variables (if both calls are active) and passes in the active status of both traversals. Hence, if both t_i and t_j are active when this call is made, the resulting invocation will execute the statements from both t_i and t_j. Note that *no other part of the generated code needs to change.* All compound statements can keep their same, single-traversal guards.

Applying this rewrite to the fused graph in our running example (Figure 2.1(e)) yields the code of Figure 2.1(f). Note that this code has redundant checks and unnecessary checks elided for clarity (for example, both null checks occur at the beginning of the method body and return if `n = null`, so can be combined into a single, unguarded null check).

2.5.3 Code Motion

An interesting side effect of the fusion and synthesis procedure is that code motion is naturally and implicitly incorporated. The dependence graph tracks dependences between statements and calls without regard to their original position in their various traversals (indeed, the only time program order matters in the dependence graph is in determining the targets of control dependence edges and determining the direction of dependence edges). As a result, if a statement must move and can move to enable fusion, the fusion procedure will discover this and the synthesis process will implement that code motion through its topological sort. If the necessary code motion cannot

happen due to the existence of other dependences, cycles in the graph will make that clear.

In our running example, statement s_3 is preorder work in traversal `f2`. However, because of its dependence on s_1, it must appear *after* s_1 in any fused traversal. s_1, in turn, *must* appear as post-order work due to its dependences on the calls to the left and right children. The only way to achieve both objectives is to move s_3 to be post-order work as well, which the topological sort achieves.

2.6 Soundness

There are two correctness criteria that must be met for fusion to be sound. First, we must *preserve work*: all of the operations performed by a given traversal in the unfused code must be performed in the fused code, and no more. In other words, each traversal must visit exactly the same portions of the tree it did before, and no more. Second, we must *preserve dependences*: if two operations have a dependence, then the operations must occur in the same order both in the original, unfused code and in the fused code.

We address these two criteria in turn.

2.6.1 Preservation of Work

The first step in proving soundness is confirming that the transformed code performs exactly the same amount of work as the original code—in other words, exactly the same set of operations (statements) are performed. We will show this assuming that all the dependences are satisfied (a fact we will prove in the next section); as a result, as long as the fused traversal executes the same set of operations as the original traversals, we can be sure that the operations each behave the same way.

Theorem 2.6.1 *As long as all dependences are satisfied, a fused traversal method synthesized from the dependence graph will execute exactly the same set of operations as in the original, non-fused traversals.*

Proof We show two things: i) if a statement or call executes in the original code, it will execute in the transformed code; ii) if a traversal *truncates* in the original code—meaning that traversal stops for a particular subtree—the same will happen in the transformed code.

1. If a statement executes in the original code, then execution of its traversal must have proceeded to this statement from the root of the tree *without* encountering a **return**. In the transformed code, the boolean flag controlling execution of this traversal starts as **true**. It can only be set to **false** upon encountering a **return**. As long as the flag is **true**, because we are assuming all dependences are satisfied, the traversal will make all of the same calls as the original traversal until it arrives at the statement in question, which will execute.

2. A traversal truncates in the original code by either executing **return**, ensuring that it will not visit its children, or by encountering a base case and no longer making child calls. If truncation happens due to a **return**, the transformed code skips over the remainder of the compound statement and sets the boolean flag to **false**. This ensures that no more work from the traversal will happen at the current node and, because that boolean flag cannot be re-set to **true** without returning from the current node, ensure that no more work will be performed even if the traversal descends into children from the current node.

Hence, statements in the fused traversal will execute iff they executed in the original code. ■

2.6.2 Preservation of Dependences

The other aspect of soundness is the preservation of dependences: if two (dynamic) statements occur in a particular order in the original code, and there is a dependence between them, they must execute in the same order in the fused code. To do otherwise would lead to different computational results. Note that we do not care about the relative ordering of statements that *do not* have dependences between them (indeed, the entire point of fusion and other transformations is to alter the execution order of these statements). We also assume here that the trees being traversed are finite: each traversal will eventually terminate.

Theorem 2.6.2 *Given an original, acyclic dependence graph, G, obtained from a set of traversals T, a dependence graph G', obtained by fusing two calls in G, and the traversal function t', synthesized from G': if G' is acyclic, then all statements that are ordered by data or control dependences in T execute in the same order in t'.*

Proof We consider all possible cases of dependences between statements and calls, both within a single traversal in T and across multiple traversals in T, and argue that if a dependence exists in T, it will be preserved in t'. We denote the jth statement in traversal i as $s_{i,j}$, and the jth call to the kth child in traversal i as $c^k_{i,j}$. All statements and calls are relative to executing at a particular **node** in the tree, and hence dependences between statements only occur at that particular **node**. If any dependence occurs between operations at different nodes, that dependence must be between a statements and a call, or between two calls. Note that here we rely on the sound overapproximation of dependences produced by the static analysis for building the dependence graph (Section 2.3).

$s_{i,j} \to s_{i',j'}$, $s_{i,j} \to c^k_{i',j'}$, $c^k_{i,j} \to s_{i',j'}$: If $i = i'$, there is a dependence between two statements or a statement and a call from the same traversal in the original code and this dependence is captured by G. In any fused dependence graph, this edge still exists. A topological sort of the dependence graph will require that the first statement appear before the second in t', so the dependence is preserved.

If $i \neq i'$, this dependence is between two statements/calls in different traversals. Note that for the dependence to be in this order, traversal t_i must have occurred before traversal $t_{i'}$. This dependence will be reflected in G and preserved in G', and, in t', the operation from the first traversal will appear before the operation from the second. Because the dependence occurs when the two traversals are at the same node, the two operations'8 appearing in the same order in t' suffices to preserve the dependence.

$c^k_{i,j} \rightarrow c^{k'}_{i',j'}$**:** These are two calls, which are captured as separate vertices in G. If $k \neq k'$ or if $k = k$ but the two calls are unfused in G', then this dependence is preserved by the same argument as the previous case.

When the two calls *are* fused is the interesting case. In the original code, all of the first call will complete before all of the second call. The dependence exists because the first call executes one or more statements that one or more statements in the second call depend on. We must show that, despite the fusion of the two calls, the necessary statements from the first call still execute before the relevant statements from the second. We show this by induction on the maximum depth of the call stacks that originate from the two calls.

We denote the maximum depth of the call chain originating from $c^k_{i,j}$ as k_i, and from $c^{k'}_{i',j'}$ as $k_{i'}$. Note that k_i and $k_{i'}$ are bounded by the depth of the tree, as each recursive call moves down to a child node of the tree. Without loss of generality, assume that $k_i < k_{i'}$.

Base case: $k_i = 1$: If both t_i and $t_{i'}$ start at node n and invoke $c^k_{i,j}$ and $c^{k'}_{i',j'}$, respectively, ending at node n', there is a dependence between the two traversals at node n'. Because dependences in the dependence graph are captured relative to an arbitrary **node**, this dependence at n' is captured in the dependence graph G. Moreover, because $k_i = 1$, the dependence must be from a *statement* from t_i to a statement or call in $t_{i'}$. Hence, this dependence must also exist in G'. The topological sort ensures that the (static) statement that leads to the operation

in t_i will occur before the (static) statement that leads to the operation in $t_{i'}$, and hence in the fused case, when both traversals descend to the child at the same time, the dependence will be preserved, by the same argument as in the first case above.

Inductive hypothesis: $k_i = l$: Assume that if the call chains rooted at $c^k_{i,j}$ are at most l calls deep, that the dependence is guaranteed to be preserved.

Inductive case: $k_i = l+1$: Consider descending one call deep into the call chain (i.e., both t_i and $t_{i'}$ make their first recursive calls). There must be at least one dependence between some operation in t_i and some operation in $t_{i'}$ (including between successive recursive calls). Again, these dependences must be captured by G. If the dependences involve any operations except $c^k_{i,j}$ and $c^{k'}_{i',j'}$, then the dependences are also captured by G', and by the arguments from above, these dependences are preserved by the fused traversal. If the dependences are between $c^k_{i,j}$ and $c^{k'}_{i',j'}$, then, because we have moved one level down in the tree, the maximum depth of the call chain rooted at $c^k_{i,j}$ is now l. Appealing to the inductive hypothesis, this dependence is preserved.

Hence, all dependences that exist between traversals in T are overapproximated and captured in G, and, if two calls in G are fused to produce G', all such dependences are preserved by the traversal synthesized from G'. ■

Note that Theorem 2.6.2 can be applied recursively: the fused graph G' is, itself, a valid dependence graph for a recursive traversal. Further fusion that does not create cycles is valid by the same argument.

2.7 Implementation

Treefuser is implemented as a Clang[6] compiler tool. It implements the process of generating a dependence graph, identifying candidates for fusion, and synthesizing

[6]http://clang.llvm.org

a new traversal function, as detailed in the previous sections. The implementation is publicly available for download[7].

2.7.1 Implementation Limitations

Because our prototype implementation of traversal fusion focuses on the challenges specific to this problem, TREEFUSER relies on several annotations and restrictions to simplify the implementation:

1. Rather than building a shape analysis to verify the "treeness" of the structure being traversed [25, 26], TREEFUSER relies on annotations to identify tree data structures and recursive fields with in those structures .

2. TREEFUSER does not allow pointers in the traversal function (except for the operating tree node). However TREEFUSER allows the existence of pointers and loops in external "leaf" [8] functions, which use annotations to abstract the access effects of the functions rather than analyzing them. These annotated functions are the only functions allowed to be called in the body of the traversals (other than the recursive traversing function calls).

3. Section 2.3.1 mentions some other restrictions of the language that exists in TREEFUSER as well (i.e: traversals should have void return types), a more robust implementation can relax many of them as mentioned earlier.

These restrictions are not fundamental limitations of the fusion approach, but instead are limitations of TREEFUSER's prototype implementation. Many of these restrictions can be relaxed by using more sophisticated analyses in the access-path construction process (*e.g.*, alias analysis to analyze heap accesses in the program, interprocedural analysis to handle function calls, and widening operators to account

[7] https://bitbucket.org/plcl/treefuserrelease.git

[8] We call these functions leaf functions because they do not call back to any of the traversal functions considered for fusion.

for potentially unbounded control flow in traversal bodies). Adding such analyses to the implementation would both reduce the annotation burden and lift many of the restrictions on the language. Nevertheless, the annotation burden is quite low even in our prototype implementation. The next section discusses these annotations in more detail.

2.7.2 Annotation Usage

TREEFUSER's prototype implementation uses annotations on data structures and effect annotations on "leaf" functions to avoid the need to implement shape and alias analyses and to allow users to implement more complex control constructs than its input language allows. This section briefly describes these annotations and shows an example of using these annotations in a program. TREEFUSER has four types of annotations:

1. `TF_TreeStructure` : A class annotation used to identify tree structures.

2. `TF_TreeRecursiveField` : A class member annotation used to identify recursive fields within the tree structure. This annotation, and the previous one, are used in lieu of a shape analysis to identify tree data structures.

3. `TF_Traversal` : A function annotation that is used to direct TREEFUSER to the functions that it should consider for fusion. TREEFUSER analyzes and validates the compliance of the those functions with the restrictions imposed by the language, and only valid traversals are considered for fusion.

4. `TF_AbstractAccess`$((id, access\_type, access\_location)|...)$: An effect annotation on (leaf) functions that summarizes the read and write effects of these functions called from traversal functions. These leaf functions have no restrictions on the operations they perform, provided they are annotated with a `TF_AbstractAccess` annotation. The effect summaries are given as a list of tuples, each of which consists of:

(a) id: An abstract id that refers to an accessed location or data structure. We use ids to allow treating operations on complex data types (*e.g.*, inserting a key into a set) as a single read or write on an abstract location.

(b) access_type: Whether the access is a read ('r') or write ('w').

(c) access_location: Indicates whether the accessed location is a *local* to the current node of the tree or *global* and accessible from all nodes.

TREEFUSER looks for sequences of "potentially fusable" methods in the remainder of the code that operate on the same tree and attempts to perform fusion as outlined in Section 2.4.

Figure 2.5 shows an example of a simple program written in TREEFUSER that uses the different annotations described earlier: a definition of binary tree with a set of integers stored at each node, `keys`, along with a `mark` flag. `markNodes` is a traversal that operates on the tree, marking nodes that have the given input `key` stored in its `keys` set. An annotated function, `checkLocalSet` is used to do the local search in the `keys` set, the set `keys` is abstracted by the id `1` and any functions that access `keys` should use the same id.

2.7.3 Fusion Heuristic

TREEFUSER's dependence graph abstraction prescribes a test for when call fusion is legal, and its synthesis procedure generates valid fused traversal methods for any legal dependence graph. However, the particular *order* in which calls are fused is open to exploration. Unfortunately, the space of possible fused traversals is vast. Choosing to fuse together children in different orders can introduce different sets of dependences (because of the vertex merging step in the fusion process), and hence different choices for fusion can lead to different performance characteristics.

TREEFUSER implements a simple greedy heuristic. It chooses a call vertex in the graph as the *target* call. It then iterates through each remaining *unfused* call and attempts to fuse it with the target call. If the call cannot be fused with the target it

```
1 class TF_TreeStructure CompleteBinaryTree{
2 public:
3         set< int > keys;
4         int mark;
5         TF_TreeRecursiveField CompleteBinaryTree * left, * right;
6         TF_AbstractedAccess((1,'r','local')) int checkLocalSet(int key){
7             if(keys.find(key)!=data.end())
8               return 1;
9             else
10              return 0;
11        }
12 };
13
14 TF_Traversal void  markNodes(CompleteBinaryTree * n, int key ){
15
16        markNodes(n->left);
17        markNodes(n->right);
18        int found =  n->checkLocalSet(searchKey);
19        if( found )
20          n->mark = 1;
21 }
```

Fig. 2.5. TREEFUSER code example.

is set aside. Once all unfused calls have been attempted to be fused with the target, TREEFUSER chooses another unfused call as the new target call. This process repeats until no more fusion is possible.

Ultimately, what we desire is a maximization of *dynamic* fusion: having as many traversals overlap as much as possible during the execution of the traversal method(s) on a particular tree. Sadly, heuristics such as maximizing the number of static calls that are fused (already a challenging objective, due to the combinatorial search space) may not achieve this goal: fusing together several calls may leave the two calls that account for the most work *in a particular tree* unfused. Dynamic fusion is a run-time property and hence is not particularly amenable to static optimization, though profiling information may help. We leave a full exploration of effective fusion heuristics to future work. However, our evaluation does look at the potential effect of the fusion heuristic on the performance of the fused schedule (Section 2.8.4).

2.7.4 Optimizations

Traversal simplification The basic code generation strategy described in Section 2.5 produces a *correct* fused traversal, including any opportunities for partial fusion, but also an inefficient one. While the fused traversal does not traverse more nodes of the tree than it has to, it performs more work at each node of the tree than is often necessary. This is because the fused traversal invokes *the same recursive function* at each node. In other words, even if a particular (original) traversal is not active at a particular node of the tree, the recursive function will still include all of the operations of that traversal. While those operations are masked off by the boolean flags generated by the fused synthesis, these additional conditional checks can still result in a substantial number of extra instructions. Moreover, calling traversal functions with a large number of arguments, even if many of those arguments are **null** or **false**, can add overhead. To mitigate these overheads, Treefuser performs a *traversal simplification* optimization.

Recall that fusion in Treefuser boils down to fusing together multiple child calls in a traversal function. If two child calls are fused among a set of five recursive traversals, that fused child call will only have two traversals active. However, the fused function that gets called includes the code of all five traversals, even though the other three traversals are "deactivated" by passing **false** for their active flags.

Traversal simplification notes that if only two calls are active for a given child call, Treefuser can instead synthesize a fused traversal *as if those were the only two traversals in the system.* In other words, the called function will only contain code from the two fused traversals. In the limit, if there is a child call that is *unfused*—only one traversal is active—Treefuser will merely call the appropriate original traversal function on that child. Deciding how to synthesize these smaller, simpler traversal functions is simple, since the dependence graph representation captures which traversals are active at each call. In our running example (Figure 2.1(f)),

traversal simplification will replace the first call to `fused` with a call to `f1`, and the third call to `fused` with a call to `f2`

Note that traversal simplification does not necessarily mean that *only* the code for active traversals will exist in a traversal function. Traversal functions may be deactivated due to dynamic effects during the traversal. Traversal simplification operates on a sound, static overestimate of which traversals are active at any point in an execution.

Improved code motion When performing a topological sort of the fused dependence graph to generate the synthesized code, there are no restrictions on the relative positioning of statements that do not have any dependences among them. Treefuser attempts to place statements from the same source traversal next to each other in the synthesized traversal. By doing so, Treefuser can merge together the guard conditionals that ensure execution only happens when the traversal is active, and can merge together the jump targets that any **return**s are directed to. In addition, Treefuser tries to place dependent statements close together to improve reuse.

2.8 Experiments and Evaluation

As discussed in the introduction, there are many applications of traversal fusion: improving locality or parallelism, eliminating intermediate structures, increasing optimization opportunities, etc. The common factor uniting these applications is *reducing the number of node accesses* by the set of traversals. We evaluate three case studies, demonstrating that we are able to substantially reduce the number of times tree nodes are visited and enhance the performance and the locality of the traversals:

1. We fuse two passes from the popular Fast Multipole Method (FMM) [28].

2. We fuse multiple DOM traversals in a document rendering engine, written in a general language.

3. We fuse 6 interlocking abstract syntax tree passes in a small compiler, each written as a fine-grained traversal[9] in a general language.

All three case studies require TREEFUSER's code motion to perform effective fusion. While the FMM case study's passes can be totally fused, the other two case studies also require TREEFUSER's partial fusion to obtain any benefits. We also evlaute the sensitivity of the fusion transformation to the fusion heuristic by comparing the performance of different fusing schedules of the AST passes.

Experimental platform As mentioned in Section 2.7, TREEFUSER is written as a Clang tool. Traversals are written in subset of C++, and annotated as described in Section 2.7 for processing with TREEFUSER. After processing, the synthesized code is compiled with LLVM version 3.8.0. The execution platform for the various performance runs is a dual 12-core, Intel Xeon 2.7 GHz Core with 32 KB of L1 cache, 256 KB of L2 cache, and 20 MB of L3 cache.

2.8.1 Case Study 1: Fast Multipole Method

The fast multipole method (FMM) is a numerical technique used to evaluate all pairwise interactions of a large number of points distributed in a space (e.g. long-ranged forces in the n-body problem, gravitational potential, and computational electromagnetic problems) [29, 30].

In this case we are considering the application of computing the gravitational potential for a large set of points distributed in a two dimensional space, the points are arranged in quad-tree where each leaf node contains a subset of the points that reside within a specific subspace, and interior nodes have summary information (e.g., the center of mass of all descendant points).

FMM is typically implemented as multiple top-down and bottom-up passes over the quad-tree. We consider two of these passes: the first traversal, which updates the

[9]These traversals are too complex to easily express using attribute grammars.

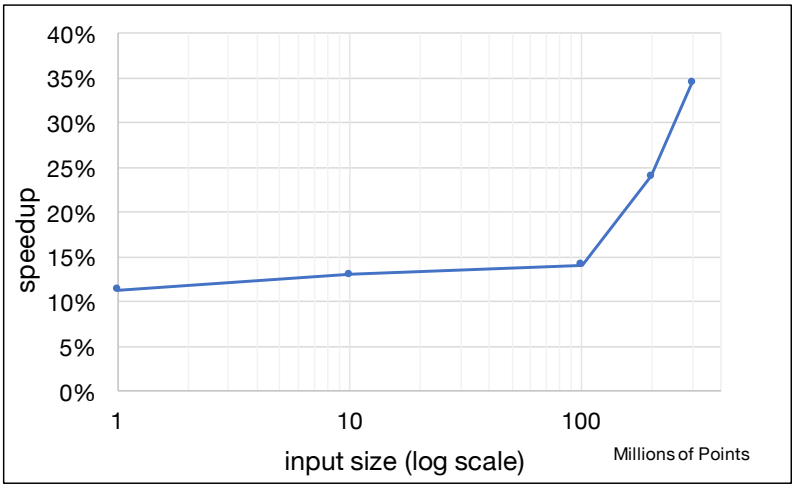


Fig. 2.6. Speedup of FMM traversals fused by Treefuser

potential of each node based on its siblings, and the second traversal, which propagates these values down the tree. We made slight modifications to the FMM implementation from the Treelogy benchmark suite [31] to fit the code into our restricted language (these modifications mostly consisted of moving complex mathematical computations into functions annotated with—obvious—read and write effects).

Treefuser was able to fully fuse the two passes and exhibit a performance improvement up to 35% over the unfused version, Figure 2.6 shows the speedups for different input sizes.

2.8.2 Case Study 2: Fusing Render-tree Passes

Tree traversals are common in any document rendering engine, including in web browsers. Considering a simple scenario: once the web page is acquired as DOM (Document Object Model) and CSSOM (CSS Object Model), the browser rendering engine constructs what is known as a render-tree. Then the rendering engine performs multiple traversals upon this tree to display the content on screen: determining the position of objects on the screen, the style of elements, etc. While these traver-

sals individually appear to be simple, in the presence of multiple traversals it is not trivial to fuse them manually. Automatically fusing these traversals can enhance the performance of the rendering engine without sacrificing programmability.

We have implemented four different traversals sequentially performed on a binary render-tree.[10] These traversals closely follow the ones shown in [18]. These traversals require partial fusion: all four traversals can be fused while visiting the left child, visits to the right child are decomposed into sets of two traversals. When rendering a tree with $2^{19} - 1$ elements, the unfused traversals visit 1835004 nodes in total, while the fused traversals visit 1048555 nodes, a reduction in node visits of 42.9%. Table 2.1 illustrates the performance benefits of fusing render-tree traversals: Treefuser delivers a 45% performance improvement, driven by substantial reductions in L2 and L3 cache accesses and misses.

Table 2.1.
Performance of render-tree traversals fused by Treefuser.

Performance Parameter	**Value**
Runtime Speedup	1.4515
Normalized L2 Data Cache Accesses	0.5688
Normalized L2 Data Cache Misses	0.5473
Normalized L3 Cache Accesses	0.5474
Normalized L3 Cache Misses	0.4843

2.8.3 Case Study 3: Fusing AST Passes

We implemented a compiler for a simple language (essentially, typed IMP) that builds an abstract syntax tree (AST) and then performs several AST passes over

[10]Render-trees are typically binary trees: larger out degrees are captured with the equivalent of "head" and "rest" pointers.

that tree to perform various compilation steps. The AST, which is naturally written using several different node types, and the traversals are rewritten according to the procedure described in Section 2.3.1 to yield a single tree node type. The resulting tree nodes have 12 different possible children. The six AST passes vary in complexity from performing constant folding, to a pair of passes that together perform available expressions analysis (avoiding the iterative nature of dataflow analysis by taking advantage of the structured nature of the AST [32]). Table 2.2 lists the six passes, their size in lines of code, and the number of recursive calls they make. These passes are called in succession. Notably, the passes have complex dependences on one another. For example, constant folding changes the expressions in the AST, which affects which expressions are available.

To get a sense of how complex the traversals are, Figure 2.7 shows the dependence graph for just the available expressions pass—manual fusion of these passes would be impractical. Further, these passes are non-trivial to express using formalisms such as attribute grammars due to their complexity (*e.g.*, constant propagation visits one child type *twice*, which would have to be decomposed into multiple attributes for expression in an attribute grammar).

Table 2.2.
Details of the AST fused passes.

Pass	**LoC**	**# of calls**	**Cumul. analysis time (ms)**
Constant folding	50	12	123
Constant prop.	96	13	137
Find uses	40	12	160
Find defs	47	7	176
Computed exprs.	63	10	187
Available exprs.	100	12	225

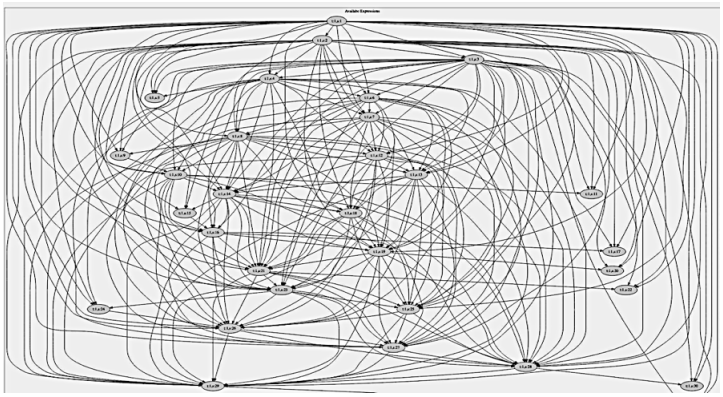


Fig. 2.7. Dependence graph of available expressions pass.

TREEFUSER **performance** The final column in Table 2.2 shows the amount of time TREEFUSER takes to analyze the dependences in a set of passes and synthesize a fused traversal. Because analysis time depends on the number and complexity of the passes analyzed, we give times for various configurations of passes. An entry in the column for a given analysis pass gives the analysis time for a configuration containing all of the passes in the table from the first row to the row in question (e.g., the entry for the constant prop. pass gives the time to fuse the constant prop. and the constant folding passes, while the entry for the last row gives the time to fuse a program that uses all six passes).

As we can see, TREEFUSER takes very little time to analyze and generate fused traversals. Note, moreover, that most of the analysis time goes in fixed overhead (e.g., analyzing the code to find fusable traversals). The analysis complexity itself is $O(n^2)$ where n is the number of statements. (Generating the access paths is $O(n)$, as our target language does not contain loops, and building the dependence graph is $O(n^2)$ as all pairs of statements/calls must be compared. The complexity of performing fusion could, in theory, be higher depending on the complexity of the fusion heuristic used.) In particular, going from analyzing 50 lines of code (row 1) to 400 (row 6) adds only about .1 seconds to the total analysis time.

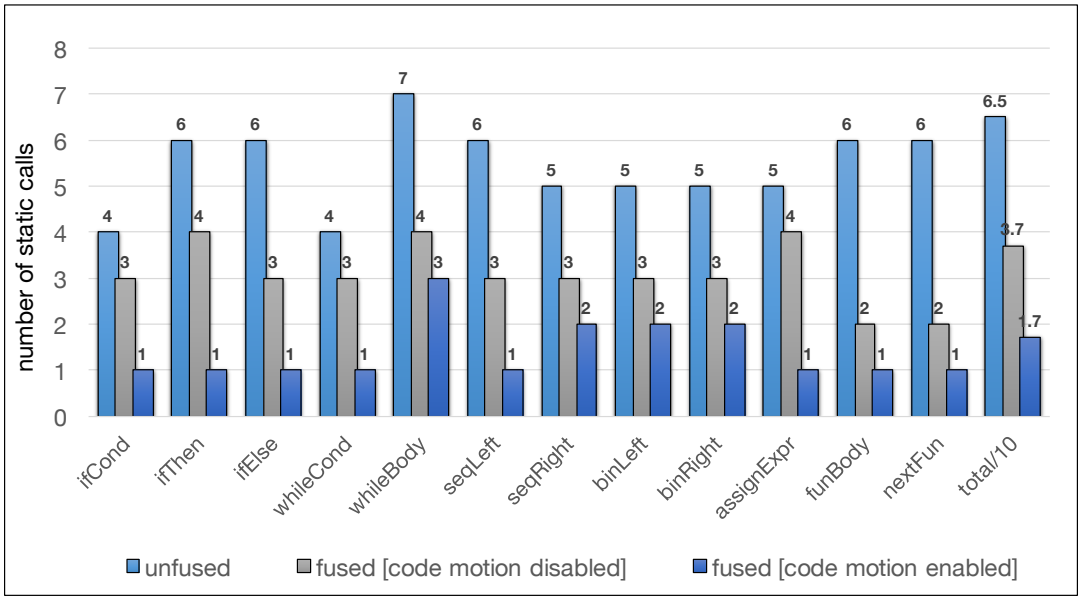


Fig. 2.8. Reduction of the number of static recursive calls for AST traversals fused by TREEFUSER

.

Static fusion effectiveness The first measure of effectiveness we study is *static* fusion effectiveness: given the six passes, each of which invokes some number of child calls, how many of those calls is TREEFUSER able to fuse together?

Figure 2.8 [11]. shows the number of calls made to each recursive field type in the tree node[12].

We find three things: first, some traversals cannot be fully fused with other traversals, however fusing some calls in those traversals is possible, which indicates that partial fusion is *necessary* to achieve better reduction in traversal calls. Second, fusion with code motion turned off produces a fused traversal with 37 calls (versus 65 calls in the baseline traversals). Finally, enabling code motion fuses the traversal further to just 17 calls. These results suggest that the two key novelties of TREEFUSER's

[11]Data is categorized by the visited child, "total" bar is scaled by a factor of 10 to fit on the chart

[12]Note that `whileBody` has seven calls, even though there are only six passes; this is because the constant propagation AST pass visits the body of the while loop twice.

transformations, generalized code motion and partial fusion, are necessary to perform effective fusion.

Dynamic fusion effectiveness Looking at static call fusion alone does not tell the entire story. Two calls may be fused, but be for portions of the AST that do not account for much of the traversal. Instead, we also want to investigate *dynamic* fusion effectiveness: for a test suite of programs, how many traversal calls are eliminated by fusion. We examine three different input programs of different sizes; Figure 2.9 [13] shows the number of times the AST nodes are visited by the different configurations of traversals: unfused, fused without code motion, and fused with code motion. We can see that without code motion, we can reduce the number of node visits by 19% in aggregate. When combined with code motion, we are able to reduce AST node visits by 56% in aggregate, demonstrating the ability of Treefuser to substantially coarsen the computation at each node in the tree.

Performance Fusion has the potential to enhance locality by bringing accesses to same set of data closer. However, the specific improvements in locality depend on the amount of data reuse between traversals, and the overall footprint of the computation. Figure 2.10(a) shows the reduction in L2 and L3 cache misses for three different input programs. The amount of reduction in the cache misses differs from one program to another, but in all cases there is a substantial decrease in the number of misses in both levels of cache.

The reduction in the cache misses directly affects the runtime of the traversals, Figure 2.10(a) shows the runtime speedups for the the three programs, with different sizes (larger sizes are generated by replicating the functions in the base program with different names, to simulate larger whole-program compilation of programs with similar AST structures). For small ASTs, the runtime is very short, and subject to

[13]Data is shown across three programs for unfused traversals, fused without code motion, and fused with code motion, the numbers of visits is normalized to the number of visits in the unfused traversals.

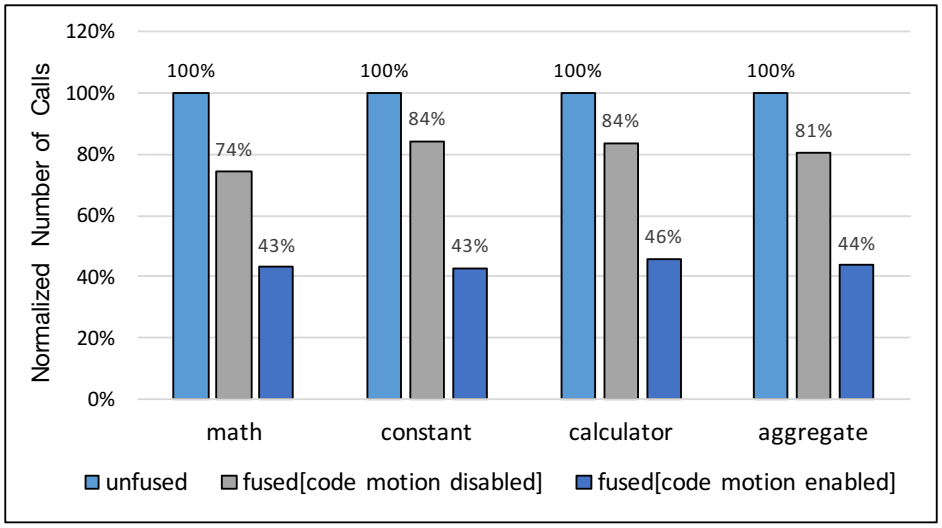


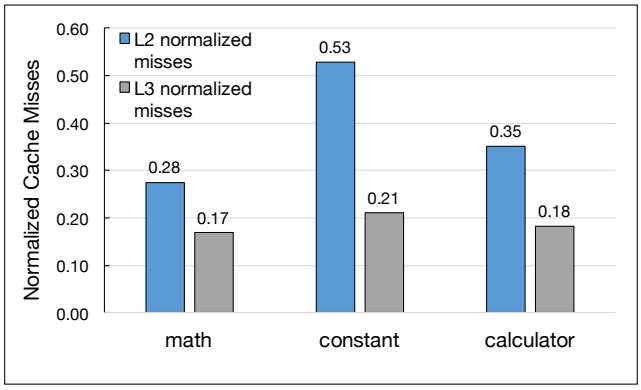
Fig. 2.9. Reduction in dynamic number of node visits for AST traversals fused by Treefuser.

measurement noise, hence there is no consistent performance story.[14] Once the ASTs grow large enough to fill the cache, performance improvements can be seen, with speedups of up to 70%.

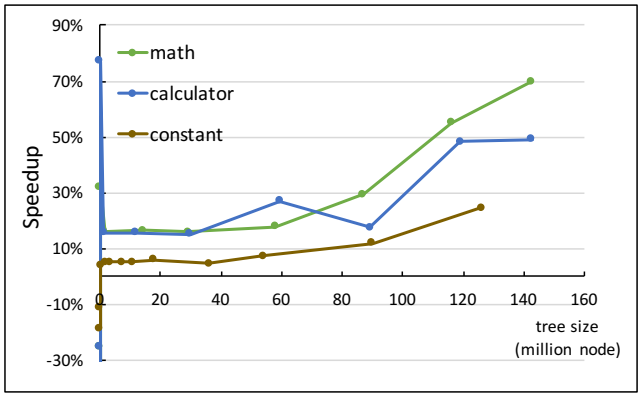
Note that ASTs are not data-intensive structures and the interaction between the accesses in most traversals is limited to some fields and the node pointer. We would expect a larger performance improvement in data intensive applications with larger chunks of data reuse. AST traversals were chosen here to show the ability of Treefuser to fuse complicated traversals.

Overhead There are two sources of overhead for the fusion transformation. First is an increase in instruction overhead, due to the additional guards inserted by (partial) fusion. The measured instruction overhead for the AST passes was relatively low: 10% or less for all three input programs. A second source of overhead is an increase in instruction misses due to the larger code footprint of fused code. While this

[14]Note that performing fusion for small trees is akin to performing loop tiling for small matrices—there is no opportunity for performance gains since the whole structure fits in cache.



(a) Cache misses of the fused AST traversals version normalized to the unfused version.



(b) Speedup of fused AST traversals over unfused traversals.

Fig. 2.10. Cache and Runtime performance of AST traversals fused by Treefuser.

overhead is noticeable for small inputs, for larger inputs the large decrease in data cache accesses and misses more than compensates.

2.8.4 Sensitivity Analysis

Different fused schedules might be generated using the general approach discussed in this work: the fusion heuristic used in constructing the fused graph (Section 2.7.3), along with the chosen topological order (Section 2.5.1), determine the final fused schedule. Treefuser uses a greedy approach for generating the fused graph and an arbitrary topological order in the synthesis process. This section studies the sensitivity of performance to these choices.

Five different fused schedules of the AST passes are studied. The fused graphs of these schedules were generated by applying the greedy approach that is used in Treefuser, but with the nodes being merged in a different random order each time (exploring more sophisticated fusing heuristics is beyond the scope of this work).

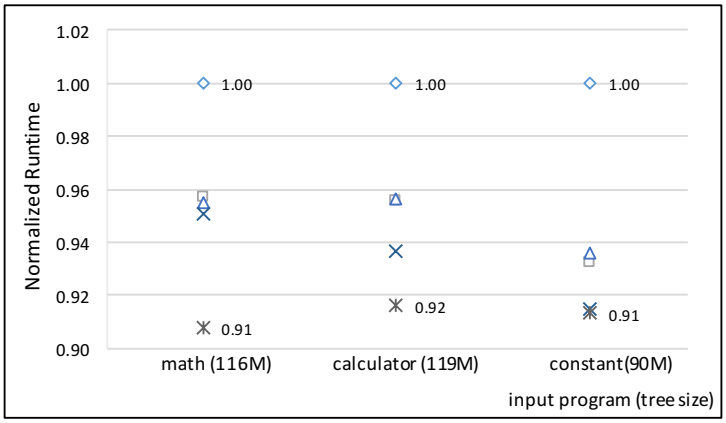


Fig. 2.11. Sensitivity analysis of fusing and ordering criteria in TREEFUSER.

Figure 2.11[15] shows the runtime of these five fused schedules, normalized to the best schedule among all of them. It is worth mentioning first that even the *worst*-performing schedule shows a significant speedup of (34%, 26% and 6%) for the three input programs respectively from left to right. While the performance differences are statistically significant, there is only about a 10% difference between the best-performing and worst-performing fusion choices. The results shows that it will be useful to further explore the space of fusion policies, yet, even with an arbitrary choice, fusion gives a decent speedup.

2.9 Conclusions

This chapter presented TREEFUSER, a framework for performing *fusion* of general tree traversals written in an imperative language. Unlike previous work on fusion of tree traversals, TREEFUSER allows programmers to write a series of small, fine-grained tree traversals in a general-purpose language, without the use of *ad hoc*, high-level abstractions, and then automatically merges the traversals together to perform

[15]The X axis shows the input name and the number of nodes for each of the inputs; the Y axis shows the runtime of the different schedules normalized to the best schedule.

coarser-grained operations at each node in the tree. Treefuser integrates two key novelties: general code motion that allows more operations to be fused together, and partial traversal, that allows for the coarsening of computation even if two traversals are not fully fusable.

The enabling techniques for Treefuser's transformations are a dependence representation that can be extracted from general traversal code, a dependence test that allows the framework to determine the validity of fusion while accounting for code motion, and a synthesis procedure that allows complex traversals to be synthesized from this high level abstraction. We show through three case studies that i) when there is reuse between traversals, Treefuser is able to provide good performance benefits; and ii) Treefuser is able to perform intricate partial fusion and code motion on a series of styling passes over a render-tree and AST traversals in a compiler, dramatically reducing the number of node accesses in both cases for large trees.

3. GRAFTER: SOUND, FIND-GRAINED TRAVERSAL FUSION FOR HETEROGENEOUS TREES

The previous chapter introduces Treefuser, a novel approach for fusing tree traversals using a dependency graph abstraction. However, the previously presented is not very practical. More specifically, it does not support heterogeneous trees, mutual traversals, or tree mutations.

This chapter introduces the second part of this thesis; Grafter [33] an extension for the work presented in chapter 2 that supports a more rich language where heterogeneous trees can be expressed without loss of efficiency. Furthermore, grafter supports traversals that are written as a set of mutually recursive functions, and allow structural leaf mutations for the traversed tree.

Overall, grafter is capable of performing more fusion (it introduces type based partial fusion) with lower overhead and achieves better speedups even for small trees (which Treefuser fail to do).

3.1 Introduction

Many applications are built around traversals of tree structures: from compilers, where abstract syntax trees represent the syntactic structure of a program and traversals of those ASTs are used to analyze and rewrite code; to web browsers and layout engines, where render trees express the structure of documents and traversals of those trees determine the location and appearance of elements on web pages and documents; to solving integral and differential equation of multi-dimensional spatial functions where kd-trees are used to represent piecewise functions and operations on those functions are implemented as tree traversals. There is a fundamental tension between writing these applications in the most ergonomic manner—where, for exam-

ple, a compiler is written as dozens of individual AST-rewriting passes [2, 3]—and writing these applications in the most performant manner—where many AST traversals must be *fused* into a single traversal to reduce the overhead of traversing and manipulating potentially-large programs [3].

In an attempt to balance these competing concerns, there has been prior work on compiler and software-engineering techniques for writing simple, fine-grained tree traversal passes that are then automatically *fused* into coarse-grained passes for performance reasons [3–7, 16]. In the world of functional programs, *deforestation* techniques rewrite data structure operations to avoid materializing intermediate data structures, either through syntactic rewrites [8, 9] or through the use of special combinators that promote fusion [13]. For web browsers, render-tree passes can be expressed in high-level, attribute grammar–like languages [6] and then passed to a compiler that generates fused passes [7]. For solving differential and integral equations, computations on spatial functions can be expressed using high-level numerical operators [4] that are then fused together into combined kd-tree passes by domain-specific compilers [4, 5]. In compilers, AST-rewriting passes can be restructured using special *miniphase* operations that are then combined (as directed by the programmer) into larger AST phases that perform multiple rewrites at once [3].

Previous approaches rely on programmers using special-purpose languages or programming styles to express tree traversals, limiting generality, Treefuser, by Sakka et al., offers an alternative [16]. Programmers write *generic* tree traversals in an imperative language—a subset of C—with no restrictions on how trees are traversed (unlike Rajbhandari et al. who limit computations to binary trees traversed in pre- or post-order [4, 5], or Petrashko et al. who require very specific traversal structures in miniphases [3]). Treefuser analyzes the dependence structure of the general tree traversals to perform *call-specific partial fusion*, which allows *parts* of traversals to be fused (unlike other prior work), and integrate code motion which implicitly restructures the order of traversals (e.g., transforming a post-order traversal into a pre-order

traversal) to maximize fusion opportunities. TREEFUSER hence represents the most general extant fusion framework for imperative tree traversals.

Unfortunately, TREEFUSER suffers from several key limitations that prevent it from fulfilling the goal of letting programmers write idiomatic, simple tree traversals while relying on a compiler to automatically generate coarse-grained efficient traversals. First, TREEFUSER's dependence representation requires that trees be *homogeneous*: each node in the tree must be the same data type. This means that to support trees, such as abstract syntax trees, that are naturally *heterogeneous*, TREEFUSER requires programmers to unify all the subtypes of a class hierarchy into a single type—e.g., a tagged union—distinguishing between them with conditionals. Second, TREEFUSER does not support mutual recursion—traversals written as a set of functions, rather than a single one—requiring the use of many conditionals to handle different behaviors. As a corollary of not supporting heterogeneous trees or mutual recursion, TREEFUSER does not support virtual functions, a key feature that, among other things, allows complex traversals to be decomposed into operations on individual node types. These limitations require expressing traversals with unnatural code and produce spurious dependences, that can inhibit fusion. Finally, TREEFUSER does not support tree *topology mutation*. While fields within nodes in a tree can be updated in TREEFUSER traversals, the topology of the tree must be read-only. This makes it unnatural to express some AST rewrites (by, e.g., changing a field in a node to mark it as deleted, instead of simply removing the node) and impossible to express others.

3.1.1 Contributions

This chapter presents a new fusion framework, GRAFTER, that addresses these limitations to support a more idiomatic style of writing tree traversals, getting closer to the goal of fusing truly general tree traversals. The specific contributions this work makes are:

1. Grafter provides support for heterogeneous types. Rather than requiring that each node in the tree share the same type (as in prior work on fusing general traversals), Grafter allows recursive fields of a node in a tree to have any type, enabling the expression of complex heterogeneous tree structures such as ASTs and render trees.

2. To further support heterogeneous tree types, Grafter supports mutual recursion and virtual functions, allowing children of nodes to be given static types that are resolved to specific subtypes at runtime, more closely matching the natural way that traversals of heterogeneous trees are written. This support requires developing a new dependence representation that enables precise dependence tests in the presence of mutual recursion and dynamic dispatch. Grafter also adds support in its dependence representation for accommodating insertion and deletion of nodes in the tree.

3. Grafter generalizes prior work's call-specific partial fusion to incorporate *type-specific partial fusion*. This allows traversals to be fused for some node types but not others, yielding more opportunities for fusion. Moreover, by leveraging type-specific partial fusion and dynamic dispatch, Grafter's code generator produces simpler, more efficient fused traversal code than prior approaches, resulting in not just fewer passes over the traversed tree, but fewer instructions total.

Table 3.1[1] summarizes Grafter's capabilities in relation to prior work. At a high level, prior work either supports heterogeneous trees but not fine-grained fusion or vice versa, while Grafter supports both, as well as the ability to write general traversals, analyze dependences and perform sound fusion automatically.

[1]We exclude syntactic rewrites of functional programs (e.g., [8]), as their rewrites are not directly analogous to fusion.

Table 3.1.
Grafter in comparison to prior work. Note that Grafter provides finer-grained fusion than TreeFuser.

Approach	**Heterogeneous trees**	**Fine-grained fusion**	**General expressivity**	**Dependence analysis**
Stream fusion [13]	✓	✗	✗	**NA**
Attribute grammars [7]	✓	✗	✗	✓
Miniphases [3]	✓	✗	✗	✗
Rajbhandari et al. [5]	✗	✗	✗	✗
TreeFuser [16]	✗	✓	✓	✓
Grafter	✓	✓	✓	✓

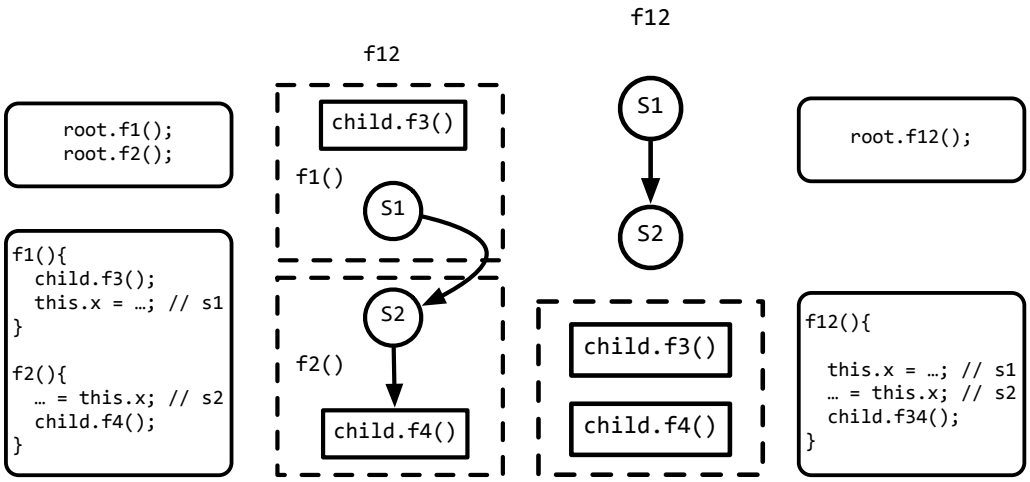
We show across several case studies that Grafter is able to deliver substantial performance benefits by fusing together simple, ergonomic implementations of tree traversals into complex, optimized, coarse-grained traversals.

3.1.2 Outline

The remainder of the chapter is organized as follows. Section 3.2 provides an overview of our fusion framework Grafter. Section 3.3 lays out the design of Grafter: the types of traversals it supports, how it represents dependences given the more complex traversals, how it performs type-directed fusion, and how it synthesizes the final fused traversal(s). Section 3.4 details the prototype implementation of Grafter. Section 3.5 evaluates Grafter across several case studies. discusses related work and Section 3.6 concludes.

3.2 Grafter Overview

Grafter adopts a strategy for fusing traversals where a programmer writes individual tree traversals in a standard, C++-like language (Section 3.3.1), as functions



(a) Unfused code (b) Outline and inline step (c) Reorder step (d) Fused code

Fig. 3.1. Illustration of steps for fusing mutual traversals in GRAFTER.

that traverse a tree structure. The fields of each tree node can be heterogeneous, and part of a complex class hierarchy, and the (mutually) recursive functions that perform a tree traversal can leverage dynamic dispatch when visiting children of a node. A full example of this style of program is shown in Figure 3.2, but for illustration purposes, we use a simple example shown in Figure 3.1(a), with two calls to the functions f_1 and f_2. Each of those functions consists of a single statement (s_1 and s_2, respectively), and a *traversing* call on the `root`'s child, `child` (f_3 and f_4, respectively). These two function calls are not independent of each other: s_1 in f_1 updates `this.x` while s_2 in f_2 reads `this.x`. GRAFTER performs fusion on such traversals to generate a new set of mutually-recursive functions that perform fewer traversals of the tree.

The fusion process starts with a sequence of traversals that are invoked on the same node of a tree, in this case `root`. Note that it is clearly safe to outline the two calls in Figure 3.1(a) into one call to function f_{12} that executes the two functions back-to-back in their original order as shown in Figure 3.1(b) (the two calls are outlined and then inlined).

GRAFTER then creates a *dependence graph* representation of the new function f_{12}. This dependence graph represents each call and statement of the traversals as a vertex, and (directed) edges are placed between vertices if two statements *when executing at a particular node in the tree* can access the same memory location (GRAFTER borrows this representation from TREEFUSER [16].) In Figure 3.1(b), there is a dependence between s_1 and s_2. We also assume, for illustration purposes, that there is a dependence between s_2 and the call $child.f_4()$. GRAFTER finds dependences between calls and statements by considering the transitive closure of what calls may access. Section 3.3.2 describes how GRAFTER analyzes accesses to find dependences and construct the dependence graph.

f_{12} is now a new, single function that performs multiple pieces of work on `root`, and invokes multiple traversal functions on `root.child`. To optimize the body of f_{12} its desirable to have $s1$, $s2$ executed closer to each other for locality benefits—if they access shared fields of `root`, then that data is likely to remain in cache. Furthermore, if the calls f_3 and f_4 on `child` are back-to-back then we can further fuse them into one call and both save a function invocation and "visit" `root.child` only once, instead of twice.

The key challenge to performing this fusion is that the reordering necessary to bring statements closer together and, more importantly, to bring traversal calls closer together, is not always safe. GRAFTER performs reordering for the statements using the dependence graph, trying to bring traversals of the same child closer to each other without reordering any dependence edges (and hence violating dependences). This reordering is done by grouping the traversal calls that visit the same node together. Figure 3.1(c) shows the results of such re-ordering.

Note that in this example the reordering step changes the traversal f_1 from a post-order traversal to a pre-order traversal ($s1$ is executed at parents before their children in f_{12}, while it is executed at the children before their parents in the original function $f1$). In other words this reordering involves performing implicit code motion

that can safely changes the schedule of the traversals for the purpose of achieving more fusion.

As calls are grouped together, GRAFTER is presented with *new* sequences of functions that this same fusion process can be applied to. In our example, the two calls grouped in the dashed box in Figure 3.1(c) are invoked on the same node of the tree (`root.child`), so GRAFTER can *repeat* this merging process, creating a new merged function f_{34}, building a new dependence graph for the merged function, rearranging its statements, and so on. Each time this re-ordering is performed, more and more operations from multiple logical traversals on the same node(s) of the tree are brought closer together, improving locality, and more and more function invocations from multiple logical traversals are collapsed, reducing invocation overhead and the total number of times the collection of traversals visit nodes of the tree.

If GRAFTER encounters a sequence of functions that has been fused before, of course, it can simply call the already-fused implementation. GRAFTER bounds the amount of functions that can be fused together to ensure this process terminates. Section 3.3.3 describes in detail how GRAFTER performs its fusion, including how it handles virtual functions.

Note that encountering cases where an already created fused function is being called again is the key for having significant performance improvement, since that means that the locality and traversing overhead enhancements will be achieved recursively. In the limit, instead of 2 traversals visiting each node of the tree once each, we will have a single traversal that visits each node only once—*total fusion*. But any amount of collapsing still promotes locality and reduces node visits.

The end result of GRAFTER's fusion process is a set of mutually recursive functions that together form a partially-fused traversal. Crucially, these fused functions are analyzed on a per-type basis, meaning that fusion can occur *partially*—not all sequences of calls in a function need to be fused—and *type-specifically*—fusion can occur for some concrete instantiations of virtual functions, but not others. This

process leads to more fine-grained, precise fusion decisions than prior work such as TREEFUSER.

The following section describes the details of GRAFTER fusion process in details.

3.3 Design

This section describes the design of GRAFTER's components in detail. In particular, we first describe how GRAFTER analyzes traversal functions to identify dependences, allowing it to build the dependence graph representation used to drive fusion (Section 3.3.2). Then we explain how GRAFTER uses the dependence representation to synthesize new, fused functions (Section 3.3.3). But first, we explain the language that GRAFTER uses to express its traversals.

3.3.1 Language

The language programmers use to write traversals in GRAFTER is a subset of C++, allowing programmers to integrate fusible tree traversals in larger projects.

A tree in GRAFTER is defined as an annotated C++ class, where instances of the class represent tree nodes. We call any such annotated class a *tree type*. Figure 3.3(a) shows the grammar for defining tree types. Children of a tree node are pointers to other tree types (not necessarily the same type as, or a subtype of, the node itself). Tree nodes can also store other (non-child) objects and primitive fields—we call these *data fields*.

GRAFTER traversals are written as member methods of tree types, implicitly "visiting" the tree node they are invoked on, and calling other traversal functions to continue the traversal. We call these *traversal methods* [2]. In order to support tree children with abstract types, a tree in GRAFTER can inherit fields and virtual

[2]For completeness, GRAFTER allows other methods to be defined for tree types, but if they are not explicitly annotated as traversal methods, GRAFTER will not consider them for fusion.

```
1  int CHAR_WIDTH;
2
3  class Element {
4    Element *Next;
5    int Height = 0, Width = 0;
6    int MaxHeight = 0, TotalWidth = 0;
7    virtual void computeWidth(){};
8    virtual void computeHeight(){};
9  };
10
11 class TextBox: public Element {
12   String Text;
13   void computeWidth(){
14     Next->computeWidth();
15     Width = Text.Length;
16     TotalWidth = Next->Width + Width;
17   };
18   void computeHeight(){
19     Next->computeHeight();
20     Height = Text.Length * (Width/CHAR_WIDTH) + 1;
21     MaxHeight = Height;
22     if(Next->Height > Height)
23       MaxHeight = Next->Height;
24   };
25 };
26
27 class Group: public Element {
28   Element *Content;
29   BorderInfo Border;
30   void computeWidth(){
31     Content->computeWidth();
32     Next->computeWidth();
33     Width = Content->Width + Border.Size*2;
34     TotalWidth = Width + Next->Width;
35   };
36   void computeHeight(){
37     Content->computeHeight();
38     Next->computeHeight();
39     Height = Content->MaxHeigh + Border.Size*2
40     MaxHeight = Height;
41     if(Next->Height > Height )
42       MaxHeight = Next->Heigh;
43   };
44 };
45
46 class End: public Element {
47 };
48
49 int main(){
50   Element *ElementsList = ...;
51   ElementsList->computeWidth();
52   ElementsList->computeHeight();
53 }
```

Fig. 3.2. An example of a program written in Grafter.

traversal methods from other tree types and can specialize inherited virtual traversals by overriding them.

Figure 3.3(b) shows the grammar for defining a traversal method in GRAFTER. Parameters of traversals are objects or primitives and are passed by value; furthermore, without loss of generality, we assume traversals do not have a return value[3]. The body of the traversal is a sequence of, non-traversing statements (*simple* statements) interleaved with traversing statements (*traverse* statements), which are function calls to traversal methods invoked on the traversed node or one of it's children.

Assignment in GRAFTER only allows writing to data fields, and hence tree nodes can not be modified in an assignment statement. Local variables in the body of the traversal can either be data definitions (primitive or objects), or aliases to tree nodes (rules 13, 14). Note that an alias variable is a constant pointer to a tree node that can only be assigned once to a descendant tree node and cannot be changed. These local variables make it easier to write traversals while precluding the need for a complex alias analysis[4].

GRAFTER uses `new` and `delete` C++ language constructs to support leaf mutations (constructs 8 and 9 in Figure 3.3(b)). The `new` statement allows a new tree node to be constructed and assigned to a specific child field of the current tree node, and the standard C++`delete` statement is permitted for deleting child fields (subtrees). GRAFTER accepts such statements only if the the trivial constructor or destructor is called, i.e., user-defined constructors or destructors are not permitted.

GRAFTER allows traversals to invoke pure functions; those functions can have an object or primitive return values, and accepts object and variables as parameters. The bodies of those functions are not analyzed, and the pure annotation indicates to GRAFTER that those functions can be considered as read-only functions. Equations 11, and 15 shows the usage of such functions.

[3]This restriction on return values simplifies GRAFTER's design, but is mostly an implementation detail.

[4]GRAFTER could allow more general assignment statements, coupled with a sophisticated alias analysis, but such support is orthogonal to the goals of this work, so we do not provide it.

$s \in$ <data-ref> ::= $s_1 \mid s_2 \mid \ldots$ $\quad t \in$ <tree-ref> ::= $t_1 \mid t_2 \mid \ldots$

$c \in$ <child-ref> ::= $c_1 \mid c_2 \mid \ldots$ $\quad f \in$ <traversal-ref> := $f_1 \mid f_2 \mid \ldots$

$l \in$ <alias-ref> := $l_1 \mid l_2 \mid \ldots$ $\quad p \in$ <pure-func> := $p_1 \mid p_2 \mid \ldots$

<prim> ::= **int** | **float** | **bool** | **double** | **char** (1)

<data-def> ::= (<prim> | <c++-class-ref>) s (2)

<tree-def> ::= **_tree_ class** t [: **public** t(, **public** t)*]{
(**_traversal_** <traversal>
| **_child_** t * c ; | <data-def>)*}; (3)

(a) Types definition in Grafter.

<traversal> := **[virtual] void** f (
[<data-def>(, <data-def>)*]) { (4)
<stmt>+ }

<stmt> := <traverse-stmt> | <simple-stmt> (5)

<simple-stmt> := <if-stmt> | <delete-stmt> |<new-stmt>
| <assignment> | <local-def> | <alias-def>
| <pure-call> | **return** ; (6)

<traverse-stmt> := **this**[->c]->f([<expr>(, <expr>)*]) (7)

<delete-stmt> := **delete** <tree-node> ; (8)

<new-stmt> := <tree-node> **=** **new** t() ; (9)

<assignment> ::= <data-access> **=** <expr> ; (10)

<pure-call> ::= p ([<expr>(, <expr>)*]); (11)

<if-stmt> := **if** (<expr>) **then** { <simple-stmt>* }
else { <simple-stmt>* } (12)

<local-def> ::= <data-def>; (13)

<alias-def> ::= t * **const** l **=** <tree-node> ; (14)

<pure-func> := (<prim> | <c++-class-ref>) p (
[<data-def>(, <data-def>)*]) {*c++ code.* } (15)

(b) Traversals definition and statements in Grafter.

(16)

<tree-node> := (**this** | l | <cast-expr>)(->c)+ (17)

<data-access> := <on-tree> | <off-tree> (18)

<off-tree> := s(.s)* (19)

<on-tree> := (**this** | l | <cast-expr>)(->c)*(.s)+ (20)

<cast-expr> := **static_cast** < t * > (<tree-node>) (21)

<bin-expr> := <expr> <c++-binary-op> <expr> (22)

(c) Expressions in Grafter.

Fig. 3.3. Language of Grafter.

The key operation performed by GRAFTER traversal methods is *accessing data and child fields*. Reads and writes to these fields, whether directly at a node or through a chain of pointer dereferences.

Accessing a variable in GRAFTER is done through an *access path*. An access path is a sequence of member accesses starting from a field or a node. Accesses paths can be classified into ¡tree-node¿ and ¡data-accesses¿ (see Figure 3.3(c)). Data accesses can be further classified, based on the location of the accessed variable, into $< on-tree >$ and $< off-tree >$ accesses. The former are accesses that start at member fields of the current node (and hence are parameterized on `this`, the node the function is called on), while the latter are to global data (meaning that all invocations of this function will access the same location, regardless of which node the function is executing at).

Any local alias variables can be recursively inlined in the access path until the access path is only a sequence of member accesses. Access paths can also be classified to reads and writes in the obvious way. Note that ¡tree-node¿ accesses appear as writes only in the new and delete statements.

Figure 3.2 shows an example of a program written in GRAFTER (we elide the annotations for brevity). This program consists of a tree of `Element`s that can be `TextBox`es or `Group`s of `TextBox`es (with a special sentinel `End` type representing the end of a chain of siblings). Each `Element` can point to a sibling `Element`, and a `Group` element can contain content elements. All elements have heights and widths, that are computed by the traversals `computeHeight` and `computeWidth`, respectively.

3.3.2 Dependence Graphs and Access Representations

The primary representation that GRAFTER uses to drive its fusion process is the dependence graph [16]. As described in Section 3.2, this graph has one vertex for each *top level* statement[5] and edges between statements if there are dependences between them. More precisely, an edge exists between two vertices v_1 and v_2, arising

[5]In other words, one vertex for each ¡stmt¿ construct, as shown in Figure 3.3(b).

from functions f_a and f_b (f_a and f_b could be the same function) if, when invoking f_a and f_b *on the same tree node* (i.e., when `this` is bound to the same object in both functions), either:

1. v_1 and v_2 may access the same memory location with one of them being a write; or

2. v_1 is control dependent on v_2 (in GRAFTER's language, this can only happen if v_1 and v_2 are in the same function and either v_1 or v_2 could `return` from the function).

So how does GRAFTER compute these data dependences?

Access automata

To compute dependences between statements in different traversals, the first step is for GRAFTER to capture the set of accesses made by any statement or call in a given traversal function. To do so, GRAFTER builds *access automata* for each statement. These can be thought of as an extension of the regular expression–based access paths used by prior work [16, 24] to account for the complexities of virtual function calls and mutual recursion.

An access path for a simple statement such as `n.x = n.l.y + 1` is straightforward. The statement *reads* $n.l.y$ and *writes* $n.x$. A simple abstract interpretation suffices to compute these access paths (intuitively, we perform an alias analysis on the function using access paths as our location abstraction [21, 27]). The abstract interpretation associates with each local variable an access path, or set of access paths when merging across conditionals, aliased to that variable. At each read (or write) of a variable, the access path(s) are added to the read (or write) set of access paths for that statement. GRAFTER collects the set of access paths for each top level simple statement in each traversal function. We do not elaborate further on this process, as this analysis is standard (and is similar to TreeFuser [16]).

The more complicated question is how to deal with building access paths for traversing function calls. Our goal is to build a representation that captures *all possible access paths* that could arise as a result of invoking the function. Rather than trying to construct path expressions to summarize the behavior of function calls, GRAFTER directly constructs *access automata* to account for this complexity. Note that these access automata are not quite like the aliasing structures computed by Larus and Hilfinger [21], because GRAFTER's representation is deliberately parameterized on the current node that a function is invoked on. We describe how GRAFTER builds these automata next.

Building access automata for statements

Each top level statement in GRAFTER has six automata associated with it that represent reads and writes of local, global and tree accesses (including $< on - tree >$ and $< tree - node >$ accesses) that can happen during the execution of the statement.

GRAFTER starts by creating primitive automata. For each access path, a primitive automaton is constructed which is a simple sequence of states and transitions. Transitions in the automata are the member accesses in the primitive access path except for two special transitions: (1) the *traversed-node* transition which appears only at the start of an $< on - tree >$ access and replaces **this**, and (2) the "*any*" transition that happens on any member access.

If a primitive access path is a read, then each prefix of the primitive access is also being read, and accordingly, each state in the primitive automata is an accept state except the initial state. If a primitive access is being written, then only the full sequence is written to while the prefixes are read.

There are some special cases to deal with while constructing primitive automata. If an access ends with a non-primitive type (a C++ object), then accessing that location involves accessing any possible member within that structure. Such cases are handled by extending the last state with a transition to itself on any possible member using an

any transition. Likewise, tree locations that are manipulated using *delete* and *new* statements, writes to any possible sub-field accessed within the manipulated node and their automata uses *any* transition to capture that.

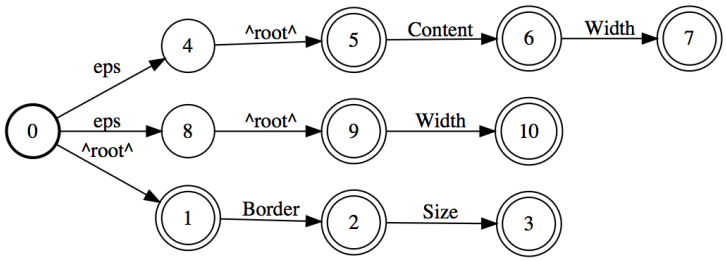


Fig. 3.4. Automata that represents summary of read accesses for the simple statement (width = Content.Width + Border.Size*2).

After the construction of the primitive automata, access automata of simple statements can be constructed from the union of the primitive automata. For example, the tree reads automaton for a simple statement is the union of the primitive automata of the tree read accesses in the statement. Figure 3.4[6] shows the tree read automaton for the statement:

```
Width = Content->Width + Border.Size*2;
```

Finding dependences between statements These automata provide the information needed to find dependences between statements. Because each statement's automata captures the full set of access paths read (or written) for a statement, and we are interested in whether the statements have a dependence *when invoked on the same tree node*, we can simply intersect the write automaton for a statement with the read and write automata for another statement to determine if a dependence could exist—a non-empty automaton means the two statements could access the same location.

[6] *eps* is epsilon transition, and *root* is the traversed node transition.

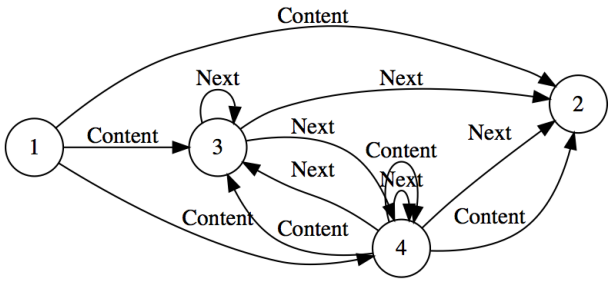
Building access automata for traversing calls

Representing accesses of traversal calls is not as simple. For a given call statement we want to construct a finite automaton that captures *any possible access path that could arise during the call relative to the tree node being traversed by caller*—including the fact that a call may invoke more traversals.

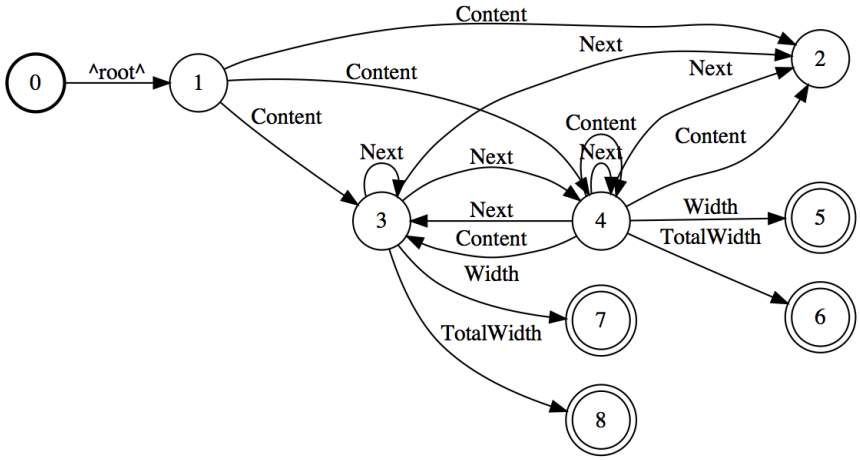
When building the access automaton for a traversal call, GRAFTER first creates a call graph that includes all the possibly (transitively) reachable functions from that call. We first note that any *off-tree* data accesses made by any of these reachable functions are, inherently, not parameterized by the receiver of the traversal calls—regardless of when and where the function gets called, those access paths will be the same. Thus, we can simply union those automata together for the functions in the call graph to capture those accesses.

The situation is more complicated for *on-tree* accesses, as those are parameterized by the receiver of the call, `this` (i.e., the node that is being traversed). To find the accessed locations relative to `this`, we need to find two things: the functions that are reachable during the call (to know which statements are executed), and the tree nodes that those functions are invoked on, relative to `this`.

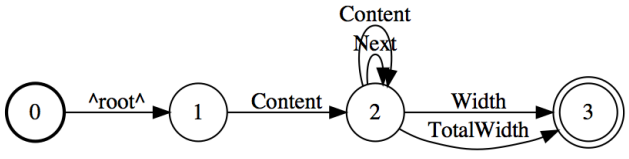
Consider building the access paths for some function f. For each function q reachable from f, the sequence of children traversed to get to the invocation of q gives an access path for the node that q is invoked on, relative to the receiver of f. This access path can be pre-pended to the statements access paths in q to produce the access paths relative to `this` (the traversed node in f). For example, when a function f invokes another function g on a child x of `this`, it invokes `x.g()`; the receiver object of g is `this.x`. Thus, to incorporate the effects of g in to the access paths of f, we can prefix the access paths of g with `this.x`. If g in turn calls h through child y, then we can prefix the access paths of h with `this.x.y` and add them to the access paths of f.



(a) Step 1: Create labeled call graph. States 2, 3 and 4 corresponds to computeWidth() functions for End, TextBox and Group types respectively.



(b) Step 2: Attach simple statements' automata and traversed node transition.



(c) Step 3: Minimize automata.

Fig. 3.5. Construction of tree writes automata for the traversing statement *content.computeWidth()*. *root* is the traversed node transition.

To account for *on-tree* accesses, GRAFTER takes the call graph for the traversing call, and labels each edge with the traversed field that is the receiver for that call. If the receiver for the call is the currently traversed node (this), the edge is labelled with the epsilon transition *eps*. Figure 3.5(a) shows the call graph that is generated for Content-¿computeWidth() from our running example. Paths in this graph thus correspond to possible sequences of child-node accesses to reach each function in the graph. For each function, GRAFTER then attaches the *statement* automata of each simple statement in that function (see the previous section) to the corresponding node in the call graph. This has the effect of treating the regular language from the statement automata as the suffix attached to the prefix that designates the receiver object. Figure 3.5(b) shows the resulting automaton, and Figure 3.5(c) shows the reduced version.

The pseudo code in Figure 1 illustrates the construction process in detail starting from a traversing statement. First, the algorithm adds the *traversed-node* transition. After that, for each possible called function that corresponds to each possible dynamic type of the called child, a state that represents all the accesses with in that function is created, and a transition on the traversed child is added. Accesses within a function are the union of the accesses of all the statements in the body of the function. Hence, for the function's traversing statements, the process will be called recursively. To guarantee the termination of such process, accesses of a unique function do not need to have more than one corresponding state in the automata. If a function is already in the automaton, then a transition to the existing state is added. Since there is only a finite number of function definitions the process is guaranteed to terminate.

Note that the constructed automata handle the possibility of non-statically-bounded trees; whenever we encounter a function we already have created a state for, we add a "back edge" in the automaton to the state that corresponds to that function. Unbounded recursion is hence represented by loops in the automaton.

```
Function buildExtendedOnTreeAutomata (CallStmt) :
    addTraversedNodeTransition (0, 1)
    appendCallAccesses (CallStmt, 1)

Function appendCallAccesses (CallStmt, State) :
    for T : CallStmt.VisitedNode.PossibleTypes() do
        F = getActualCalledFunction (T, CallStmt)
        if ! FunctionToState.find(F) then
            FunctionToState[F] = NewState = addState ()
            for Stmt : F.body() do
                if isCallStmt (Stmt) then
                    appendCallAccesses (Stmt, NewState);
                else
                    appendStmtAccesses (Stmt, NewState);
                end
            end
        end
        addTransition (State, FunctionToState[F], CallStmt.VisitedNode);
    end
```

Algorithm 1: Construction of tree access automata for traversing statements.

Finding dependences between statements and calls The access automata constructed for calls are no different than those constructed for statements. By using the call graph to construct access paths for receiver objects of functions, all the access paths generated by the final access automata are rooted at the same receiver object as the automata for statements. Hence, finding dependences between statements and calls (or calls and calls) can be achieved by intersecting the automata and testing for emptiness.

3.3.3 Fusing Traversals

Overview At a high level, GRAFTER performs fusion by repeatedly invoking the following steps:

1. Find a sequence L of consecutive traversal functions invoked on the same tree node n.

2. *Outline* these traversal functions into a new function f_L that is called on n. If such an f_L has already been created in an earlier iteration of this process, simply call the existing f_L.

3. *Inline* each of the individual traversal functions in f_L to expose the work done on the node n.

4. *Reorder* the statements in f_L to bring statements that access the same fields closer together *and* to create new sequences of traversal functions invoked on the same tree node (typically, some child node of n).

5. Repeat the process for these newly created sequences of calls.

These steps constitute the fusion algorithm of GRAFTER. In particular, every time a sequence L is encountered *more than once*, and hence an existing f_L can be reused, GRAFTER has exploited an opportunity for fusion. We now explain this fusion process in more detail, and also sketch a proof of correctness.

Details Fusion starts with a sequence of traversal functions that are invoked at the same tree node (e.g., the root). GRAFTER searches for such candidates in the compiled program and initiates the fusion process for each of them. For example, the sequence followed by in Figure 3.2 line 51.

Because a given function may be virtual, GRAFTER first computes all possible sequences of *concrete* functions that may be invoked as a result of a sequence of function calls. For each type T that the sequence of calls can be invoked on, GRAFTER constructs a sequence L of concrete calls. In our example, there are three, depending on whether @ElementList@ points to a @TextBox@, @Group@ or @End@.

For each function sequence L a fused function with label f_L is created (if one has not already been generated). If the label f_L does not already exist, then its corresponding function needs to be generated. A dependence graph G_L is constructed for

the statements in the traversals in L. In other words, the fused function is essentially a function containing the *inlined* statements from each call in the sequence L, in order. Note that a sequence L may contain the same static function more than one time (i.e., the same function can be invoked on a given node of a tree more than once). In this case, G_L contains statements from multiple copies of that function, and the statements from the two copies are treated as coming from different traversal functions.

Once G_L is constructed, the statements (nodes) in the statement can be reordered as long as no dependences are violated (as long as a pair of dependent statements are not reordered). Grafter thus reorders the statements and try to group invocations on the same node together. It then generates the fused function code, as explained in the next section. This newly generated function has grouped traversals invocations on the same node together (and these invocations may have come from different functions than the original sequence L), creating new sequences of functions. Grafter then process these new sequences of functions to generate more fused functions. Whenever Grafter encounters a sequence of functions it has seen before, it does not need to generate a new function, but instead inserts a call to the already generated function. Crucially, if this new sequence is *the same as for a function currently being generated*, Grafter just inserts a recursive call to that function.

The end result is a set of mutually recursive fused functions, each for a different set of traversals that are executed together at some point. Furthermore each of those functions is fused independently of the others. This process introduces *type-specific-partial-fusion*, since for an invocation of a traversals on a super type, the set of the called functions that corresponds to each dynamic type are fused independently, and hence some of them actually might be fusible while others are not.

Note that Grafter only generates new functions for sequences it has not seen before. Because Grafter limits the number of functions that can be fused together (see Section 3.4), the number of sequences of functions is finite, and hence fusion is guaranteed to terminate.

Proof sketch of soundness The argument for the soundness of GRAFTER's fusion procedure is straightforward. First, we note that the outlining and inlining steps (steps 2 and 3) in the fusion process are trivially safe because they do not reorder any computations, and hence cannot break any dependences. Step 4 could potentially break dependences, but because GRAFTER performs a dependence analysis, it can ensure that statements are only reordered if dependences are preserved. Hence, this step is also clearly sound.

The tricky step in GRAFTER's fusion algorithm is the step where it gains the advantage of fusion: if a sequence of calls to a particular sequence of traversal functions matches a sequence that GRAFTER has already generated a fused function for, GRAFTER immediately replaces the original sequence of calls with a call to the fused function rather than generating another new function. This is only safe if the already-fused function will do the same thing as the original function sequence.

To see that this is safe, we note that the process of outlining followed by inlining means that the code of the fused function f_L is not dependent on the node f_L is invoked on—in other words, if the original sequence of traversal functions are invoked on @root.left@, after outlining and inlining, the statements within f_L are relative to the formal parameter @n@ of f_L, and will be exactly the same as if f_L were produced from the same sequence of functions invoked on a different tree node, such as @root.right@. In other words, two identical sequences of traversal functions, L and L' that are invoked on different tree nodes will yield *identical* functions f_L and $f_{L'}$ after outlining and inlining. Because the dependence graph for these functions are identical, any reordering GRAFTER does to create a fused function can be applied to both f_L and $f_{L'}$. It is obvious, then, that, upon encountering the same sequence of traversal functions L, even if those functions are invoked on different nodes, GRAFTER can reuse an existing synthesized function.

However, there remains one gap: if, while fusing a sequence of functions L to generate f_L, GRAFTER encounters the same sequence of invocations L that is reachable (transitively) from f_L, GRAFTER will substitute a call to f_L. In the simplest case,

if f_L contains L, then a new invocation to f_L will be inserted into the body of f_L. Hence, in these situations, GRAFTER is changing the behavior of f_L while using it to replace L. This process feels circular. However, a straightforward inductive argument on the depth of the call stack (in other words, the number of recursive invocations of f_L before reaching the end of the tree or some base case) shows that this new invocation of (the rewritten) f_L behaves the same as the original sequence L. This argument mirrors the proof for the soundness of TreeFuser [16, Section 7].

3.3.4 Traversal Code Generation

As described in the previous section, to fuse a sequence of functions L, GRAFTER generates a graph G with statements and groups of calls that represents the fused function f_L. The body of the function f_L is generated from the graph in a way similar to TreeFuser [16], yet it incorporates several changes to account for mutual recursions and virtual calls.

The generated function f_L is a global function that takes a pointer to the traversed node as the first parameter (this function will be called from a virtual function switch placed in the tree classes). Since the functions in L can be defined in different classes, those traversals might be operating on different types, however since they are all invoked on the same child, there must be a super type that encloses all of them. This type is used for the traversed node parameter in the generated function f_L. A lattice for the types traversed in the functions in L is created to find such type. Line 3 in figure 3.6 shows the result of fusing the two functions that computes the width and the height for @TextBox@ element in the program shown in figure 3.2.

The traversed node parameter is followed by the traversal's parameters, and an integer that represents the set of active traversals, @active_flags@. This parameter can be seen as a vector of flags where each bit determines whether a specific traversal function is active or truncated at any point during the execution of the fused function.

```
1  ...
2  void _fuse__F3F4(TextBox *_r, int active_flags) {
3    TextBox *_r_f0 = (TextBox *)(_r);
4    TextBox *_r_f1 = (TextBox *)(_r);
5    if (active_flags & 0b11) /*call*/ {
6      unsigned int call_flags = 0;
7      call_flags <<= 1;
8      call_flags |= (0b01 & (active_flags >> 1));
9      call_flags <<= 1;
10     call_flags |= (0b01 & (active_flags >> 0));
11     _r_f0->Next->__switch1(call_flags);
12   }
13   if (active_flags & 0b1) {
14     _r_f0->Width = _r_f0->Text.Length;
15     _r_f0->TotalWidth = _r_f0->Next->Width + _r_f0->Width;
16   }
17   if (active_flags & 0b10) {
18     _r_f1->Height = _r_f1->Text.Length *
19         (_r_f1->Width / CHAR_WIDTH) + 1;
20     _r_f1->MaxHeight = _r_f1->Height;
21     if (_r_f1->Next->Height > _r_f1->Height) {
22       _r_f1->MaxHeight = _r_f1->Next->Height;
23     }
24   }
25 };
26 void TextBox::__switch1(int active_flags) {
27     _fuse__F3F4(this, active_flags);
28 }
29 void Group::__switch1(int active_flags) {
30     _fuse__F5F6(this, active_flags);
31 }
32 void End::__switch1(int active_flags) {
33     _fuse__F1F2(this, active_flags);
34 }
35 int main() {
36     Group *ElementsList;
37     //ElementsList->computeWidth();
38     //ElementsList->computeHeight();
39     ElementsList->__switch1(0b11);
40 }
```

Fig. 3.6. Sample output code generated by GRAFTER.

Those flags are needed because the fused traversals can have different termination conditions.

To *call* the the generated function f_L, GRAFTER replaces the original sequence of calls with a call to a newly created virtual function that acts as a switch and calls the corresponding f_L for each possible traversed type T. Lines 26-34 in Figure 3.6 shows an example of such switch which is called at line 39. The switch's arguments are the arguments of the fused original call expression in addition to an integer that represents the active traversals. The switch's arguments are passed to the new fused function as well as the traversed node (`this`).

Statements of different traversals in the fused function should see the traversed node type the same way they see it in the original function that they came from. This is needed for accessibility (a field might be defined in a derived type while the type of the traversed node is a base type) and correctness (a derived class can shadow a base class variable of the same name). To handle this, an alias of the traversed node parameter is created with the desired type for each participating traversal using casting (lines 4 and 5 in figure 3.6) and those aliases are used by the statements whenever a tree access happens (lines 15-19).

A topological order of the nodes in the graph G is then obtained that represents the order of the statement in body of the fused function. Each node in the topological order (which corresponds to a top level statement in one of the traversals) is then written to the function in order. A simple statement is only executed if the traversal that it belongs to is not terminated. Return statements terminate a traversal and updates the corresponding active flags as in TreeFuser [16]. Appropriate flags should be passed when a fused call is invoked within a traversal. The @call_flags@ variable, defined at line 7 in Figure 3.6, holds the active flags that are passed to the next traversing call. Lines 7–11 fill in the appropriatef flags from the @active_flags@ in the @call_flags@ based on which traversals the outlined calls belong to.

3.3.5 Limitations of Grafter

Limitations of Grafter's dependence analyses, fusion procedure, and language mean that it cannot exploit all possible fusion opportunities for all possible traversal implementations. Here, we group the limitations of Grafter into three categories.

First, Grafter's language and implementation have been limited in some ways merely to simplify the dependence analysis. For example, Grafter does not support pointers other than to nodes of the tree, but relaxing this simply requires enriching the access analysis with standard alias analysis techniques. The dependence analysis can similarly be extended to support loops within traversal functions (that do not themselves invoke additional traversal functions). In these scenarios, Grafter's basic fusion principles need not change. Finally, adding a shape analysis to Grafter could allow it to avoid annotating data structures to establish that they are trees.

Second, some extensions to Grafter would require extending the machinery of code generation to handle them. For example, supporting conditional traversal invocation can be done through syntactic manipulation (pushing the condition into an unconditionally-invoked traversal function that immediately returns if the condition is false), but this introduces instruction overhead. Managing conditional calls, traversal functions invoked within loops, or return values from traversal functions will require some new strategies for generating fused traversal code, but likely would not require substantial changes to the rest of the fusion machinery.

Finally, some extensions to Grafter may require devising new theories of fusion: new principles for how a fused traversal actually operates. In this category are extensions like functions that operate over multiple trees (*e.g.*, traversals that zip together two trees), or traversals that perform more sophisticated tree mutation such as rotations.

3.4 Implementation

Grafter is implemented as a Clang tool that performs source to source transformation for input programs [7].

Grafter uses Clang's internal AST to analyze the annotated traversals and tree structure, and performs checks to validate that the annotated traversals satisfies Grafter's restrictions. Any traversal that does not adhere to Grafter's language is excluded from being fused. Grafter uses OpenFST library [34] to construct the automata that represent accesses of statements and perform operations on these automata.

Different criteria can be used to perform grouping of call nodes in the dependence graph during fusion. Grafter uses a greedy approach for grouping: it selects an arbitrary un-grouped call node, and tries to maximize the size of the group by accumulating other un-grouped call nodes. The process continues until there is no more grouping left. This criteria is sufficient to show significant improvements (Section 3.5), thus we did not investigate any other approach for grouping.

As mentioned earlier, in order to control the fusion process Grafter must limit the number of functions that can be fused together. It may seem odd that these cutoffs are needed at all, since there are only a finite number of function definitions in the program. However, if a traversal function calls *multiple* functions on the same child node (say, two), and each of *those* functions call two functions on the same child node, then it is clear that at each level of the tree, there are *more active traversal functions* than at the previous level. Because each step of Grafter's fusion process essentially descends through one level of the tree to expose more fusion opportunities, we will systematically uncover more and more functions to fuse together. Hence the need for a cutoff.

[7] Grafter is available at https://bitbucket.org/plcl/grafter_pldi2019/.

GRAFTER limits fusion in two ways: by limiting the length of a sequence of functions to fuse, and by limiting the number of times any one static function can appear in a group.

3.5 Evaluation

We evaluate GRAFTER through four case-studies from different domains, demonstrating its ability to express traversals over heterogeneous tree structures without compromising efficiency, to perform fusion efficiently, and to significantly enhance the performance of traversals, even when processing small trees. The four case-studies are:

- Fusing multiple traversals over a render tree.
- Fusing multiple AST optimization passes.
- Fusing multiple operations on piecewise functions.
- Fusing two fast multipole method traversals.

Experimental platform . Since rendering is a common task performed on mobile phones, yet the memory on such devices is relatively small, we evaluated the render tree traversals on a smartphone with Qualcomm Snapdragon 425 SoC. The main platform, which is used for the other case studies, is a dual 12-core, Intel Xeon 2.7 GHz Core with 32 KB of L1 cache, 256 KB of L2 cache, and 20 MB of L3 cache. All cache lines are of size 64 B. L1, L2 caches are 8-way associative and L3 cache is 20-way associative. We have used single-threaded execution throughout our evaluation. Clang++ was used for compilation with "-O2" optimization level for all case studies.

For each experiment, we measure four quantities:

1. The number of node visits. This measures the number of times any traversal function is called on any node in the tree. This provides a performance-agnostic

measure of fusion effectiveness: the more fusion is performed, the fewer functions are called per tree node.

2. The number of instructions executed. Synthesizing fused functions requires additional work to keep track of when various fused traversals truncate, additional stub virtual functions, etc. On the other hand, fusion reduces call and memory instructions. Measuring instructions executed provides an estimate of GRAFTER's overall instruction overhead.

3. The number of cache misses. One benefit of fusion we expect to see is improved locality and reduced memory accesses. Cache misses provide a good proxy for this.

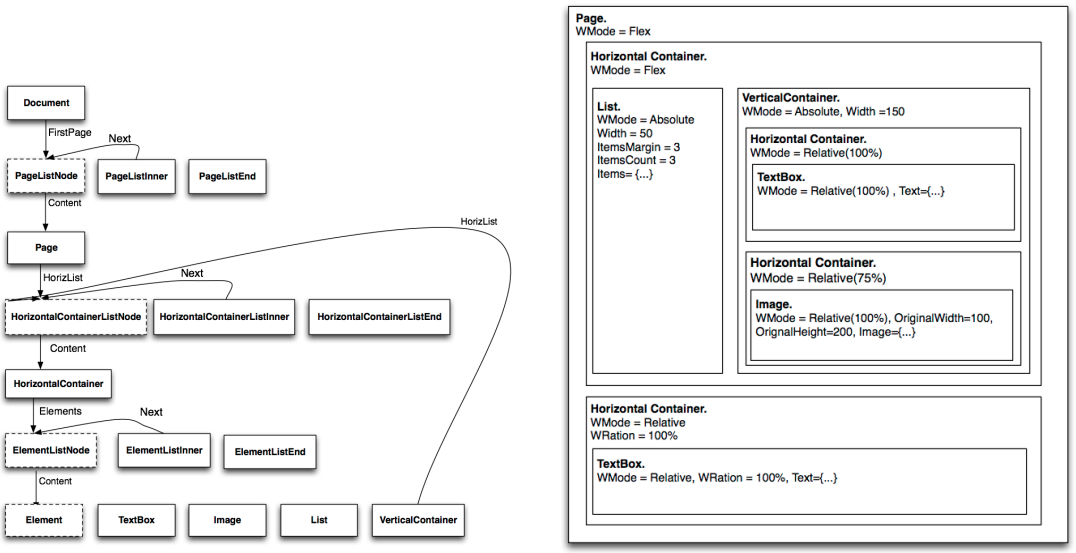
4. Overall runtime. Fusion improves locality, but potentially at the cost of increased instruction overhead (as reported by Sakka et al. [16]). The final effectiveness of fusion is hence determined by runtime.

3.5.1 Case Study 1: Render Tree

Tree traversal is an integral part of document rendering. A render tree that represents the organizational structure of the document is built and traversed a number of times to compute visual attributes of elements of the document. We implemented a render tree for a document that consists of pages composed using nested horizontal and vertical containers with leaf elements (TextBox, Image, etc.).

Figure 3.7(a) shows the structure and the class hierarchy of the render tree. The tree nodes are of 17 different types; boxes in the figure represent types, arrows represent the fields and point to type of the child node. For instance a `HorizontalContainer` contains a list of elements that can be accessed through the field `Elements`. Boxes with dashed borders are of super-types of the boxes to the right of them. For instance, an `Element` can be a `TextBox`, a `Image` or a `VerticalContainer`.

Figure 3.7(b) shows an example of a page that can be represented using this class hierarchy.



(a) Class hierarchy of the render tree.

(b) The layout of the rendered page.

Fig. 3.7. Class hierarchy of the render tree, and layout of the rendered page.

Five rendering passes are implemented and listed in Table 3.2. These passes are dependent on each other. For example, computing the height of an element depends on computing the width and font style. In Grafter, passes are implemented as fine-grained, stand-alone functions for each type.

In the first experiment we compared the effectiveness of Grafter and TreeFuser [16], prior work that can also perform (partial) fusion for general recursion. We produced a baseline that performs no fusion, as well as a version that uses Grafter's full fusion capabilities. We also implemented the same passes in TreeFuser [16]. To accommodate the limitations of TreeFuser, that implementation collapses the types into a single type, using conditionals to determine which code path to take. Again,

we evaluate a baseline that uses the TreeFuser language to implement the passes, and a version that performs as much fusion as possible with TreeFuser.

The page in figure 3.7(b) is replicated to create documents of different sizes and are used for the experiments.

Table 3.2.
Render-tree and AST fused passes names.

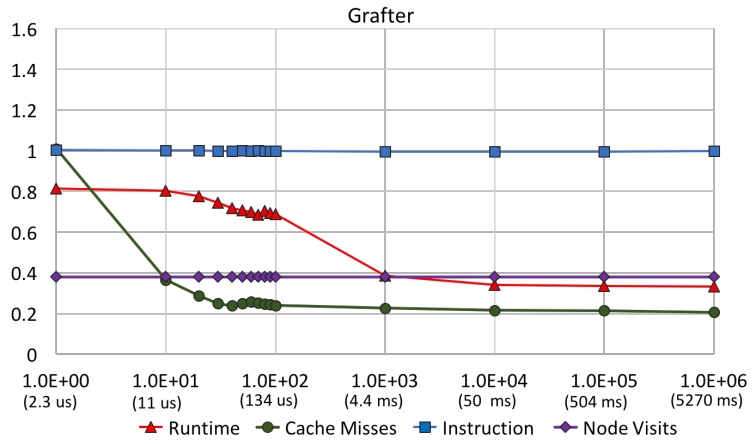
Render-tree traversals	**AST traversals**
1. Resolve flex widths	1. De-sugar increment
2. Resolve relative Widths	2. De-sugar decrement
3. Set font style	3. Constant propagation
4. Compute height	4. Replace variable references
5. Compute positions	5. Constant folding
	6.Remove unused branches

For this experiment we created documents of various sizes by replicating the page shown in Figure3.7(b). Figure 3.8(a) evaluates the Grafter-fused implementation of the render-tree case study, normalized to the unfused Grafter baseline, while Figure 3.8(b) evaluates the TreeFuser-fused implementation, normalized to *its* unfused baseline[8] [9].

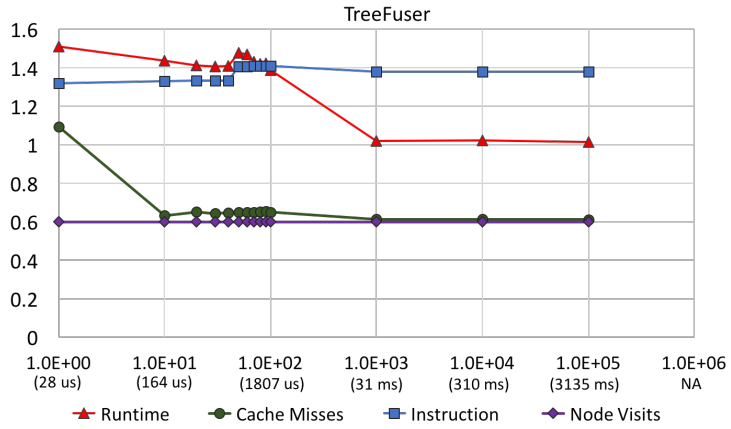
Both systems show the expected results of fusion: reduced node visits after fusion, and reduced cache misses. Nevertheless, on both metrics, Grafter does better than TreeFuser: due to its finer-grained representation and fusion, it can more aggressively fuse traversals, resulting in 60% fewer node visits than the baseline, compared to 40% fewer node visits for TreeFuser. Note that because both the Grafter and TreeFuser

[8]Number in parentheses is runtime of the baseline. Number of pages on the x axis and normalized measurement on the y axis.

[9]For small input sizes, each experiment is run in a loop to achieve a reasonable overall runtime, then divided through by the number of loop iterations to find the per-run metrics. 95% confidence intervals for render tree experiments are within ±5% for all measurements except for cache misses of trees smaller than 50 pages in figure 3.8. For those the intervals expands as the tree size decreases down to ±60%. Note that for such small trees, the absolute number of misses is quite small.



(a) GRAFTER-fused implementation normalized to unfused GRAFTER implementation.



(b) TreeFuser-fused implementation normalized to unfused TreeFuser implementation. (1M page baseline does not complete.)

Fig. 3.8. Performance comparison of render-tree passes written in GRAFTER and TreeFuser for different tree sizes.

implementations do the same work, the baselines have exactly the same absolute number of node visits. This increased fusion is reflected in cache misses: TreeFuser's fusion reduces cache misses by 40% while GRAFTER reduces misses by 80%.

We also see the advantage of GRAFTER's type-specific fusion approach: because fused functions are on a per-type basis, GRAFTER is able to leverage dynamic dispatch

to specialize traversals. As a result, it exhibits virtually no instruction overhead relative to the baseline. TreeFuser, in contrast, has a 30–40% instruction overhead.

All told, these effects mean that TreeFuser cannot achieve performance improvements for the inputs we investigate: the improved memory system behavior cannot outweigh the instruction overhead. In contrast, Grafter sees substantial performance improvements: 20% even for the smallest input (a single page), and 60% for inputs of 1000 pages or more. And note that this is despite the fact that Grafter's *baseline* is already substantially faster than TreeFuser's, as seen in the baseline runtimes shown in Figures 3.8(a) and 3.8(b).

Programmability-wise, the overall logical lines of code (LLOC) [10] for the body of the traversals is the same for both TreeFuser and Grafter. However, those LLOC are distributed among 55 different simple functions in Grafter while TreeFuser requires one function per traversal, with complex conditionals to disambiguate types.

Grafter's expressive language lets us write simple, efficient code; and Grafter's aggressive, fine-grained fusion lets us automatically wring substantial performance improvements out of even this efficient code.

We also evaluated Grafter's render tree traversal on multiple different documents configurations (input trees), with different properties. The results are summarized in Table 3.3: Grafter achieves speedups between 1.5× and 4.5×[11] [12].

3.5.2 Case Study 2: AST Traversals

Abstract Syntax Tree (AST) representation of programs is common in modern compiler frontend. Various validation and optimization passes are performed on AST representation. We implemented AST passes for a simple imperative language that has assignments, if statements, functions, and allows certain syntactic sugars. Fig-

[10] We use Fenton's definition of LLOC for C++: the number of instructions with the semantic delimiter [35].

[11] Doc1 consists of a large number of simple pages. Doc2 is a very large single page with dense deeply nested components. Doc3 consists of 150 pages of different sizes and complexities.

[12] The results in Table 3.3 were collected on the main platform, rather than the mobile platform.

Table 3.3.
Performance of traversals fused by GRAFTER normalized to unfused ones for different render tree configurations.

	runtime	**cache misses**	**node visits**	**tree size**	**description**
Doc1	0.22	0.17 (L2) 0.14 (L3)	0.38	90MByte	10^5 simple pages
Doc2	0.65	0.28 (L2)	0.4	4MByte	1 dense page
Doc3	0.47	0.24 (L2) 0.18 (L3)	0.4	58MByte	150 pages of different sizes

ure 3.9 shows the language constructs and the class hierarchy of the AST that represents programs in the language. Node types in the AST belong to different hierarchy levels and passes are implemented as virtual functions defined at the top level type.

Table 3.2 shows the six different AST traversal passes we implemented in GRAFTER: two de-sugaring passes for increment and decrement operations, and three optimization passes. The constant propagation pass is written as two traversals and works as follows: the constant propagation traversal looks for constant assignments and for each of them it initiates a traversal to replace variable references with constants. The AST passes depends on each other; de-sugaring must happen before the optimization passes, and removing unused branches depends on constant folding and propagation since they may produce constant branch conditions. Furthermore, those passes mutate the tree (e.g., to de-sugar an expression, one part of the AST is deleted and another part is constructed).

We wrote a function that consists of different statement types, and expressed it as an AST. This function was replicated in order to obtain bigger trees for the evaluation. Figure 3.10 shows the performance of the fused AST traversals with respect to the un-fused ones [13] [14]. GRAFTER reduces L2 cache misses by 75% and

[13]Number of functions on the x axis, and normalized measurements on the y axis.

[14]95% confidence intervals for the AST experiments are within ±5% for all measurements, except for the point 10^3 in Figure 3.10, where confidence intervals are within ±33% for L3 cache misses and ±15% for runtime.

once the tree is big enough, it reduces L3 misses by 70% as well. The fused traversals have instruction overhead (between 15% and 4%), but that overhead is overcome by the reduction in cache misses. The fused traversals are 1.25× to 2.5× faster than the un-fused traversals depending on the tree size.

The instruction overhead is caused by different AST passes having different truncation conditions, unlike the render tree traversals, which all completely traverse the tree and get truncated at the same time. For instance, the "replace variable references" pass gets truncated once the reference is reassigned. Because this truncation is dynamic, parameters of the truncated traversals will kept being passed, and the truncation flags will continue to be checked, until all traversals truncate, increasing overhead.

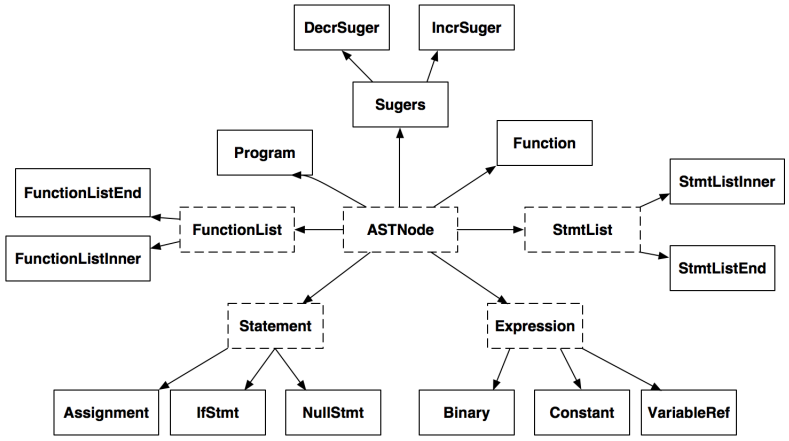


Fig. 3.9. Class hierarchy of the AST used in evaluation with 20 different types.

Table 3.4 shows the performance of the fused traversals for different AST inputs with respect to the unfused ones. *Prog1* consists of a large number of normal-sized functions. Since all the traversals are fusible across the function list, *Prog1* has the highest reduction in node visits, 34%. On the other hand *Prog2* consists of only one large function, and hence has less reduction in node visits, 8%. *Prog3* consists of

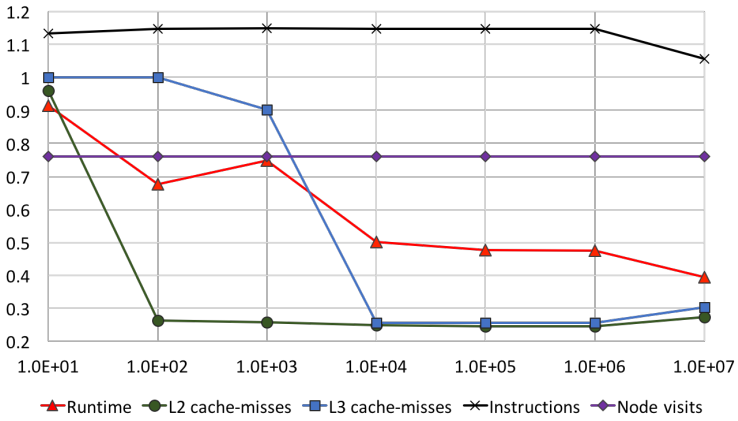


Fig. 3.10. Performance of AST traversals fused by GRAFTER normalized to the unfused ones for different tree sizes.

multiple functions with long live ranges. The tree of *Prog3* is the largest and thus reductions in L3 cache misses are achieved along with a 70% reduction in the runtime.

Table 3.4.
Performance of fused traversals normalized to unfused ones for different AST configurations.

	Runtime	**Cache misses**	**Node visits**	**Tree size**	**Description**
Prog1	0.71	0.26 (L2)	0.76	8MByte	Small functions
Prog2	0.87	0.58 (L2)	0.92	6MByte	One large function
Prog3	0.31	0.48 (L2) 0.26 (L3)	0.93	31MByte	Long live ranges

3.5.3 Case Study 3: Piecewise Functions

MADNESS (Multiresolution, Adaptive Numerical Environment for Scientific Simulation) [20] uses kd-trees to compactly represent piecewise functions over a multi-dimensional domain. The inner nodes of the tree divide the domain of the function into different sub-domains, while leaf nodes store the coefficients of a polynomial that

estimates the function within the node's sub-domain. MADNESS uses this representation to solve differential and integral equations: mathematical operations on these functions (*e.g.*, differentiation, scaling, sqaring) can be captured as traversals of the tree.

In this case study, we implemented kd-trees for single variable functions, and different traversals to perform computations on these functions. Table 3.5 shows these traversals. Some of these traversals requires structural changes to the tree. For example, the *multXRange(a, b)* traversal multiplies the function in the range $[a, b]$ by x where x is the variable representing the domain. A leaf node that has a non-empty intersection interval of its domain and the interval $[a, b]$, needs to be split into multiple nodes if its domain does not lie completely within the interval $[a, b]$.

Table 3.5.
Description of operator traversals used in piecewise functions case study.

Function	**Description**
1.scale(c)	$f(x) = cf(x)$
2.add(c)	$f(x) = f(x) + c$
3.square()	$f(x) = f(x) \cdot f(x)$
4.differentiate()	$f(x) = f^{(1)}(x)$
5.addRange(c, a, b)	$f(x) = f(x) + c(u(a) - u(b))$
multXRange(a, b)	$f(x) = xf(x) \cdot (u(a) - u(b))$ $+f(x) \cdot (1 - u(a) + u(b))$
6.addXRange(a, b)	$f(x) = f(x) + x(u(a) - u(b))$
7.integrate(a, b)	$\int_a^b f(x)$
8.project(x_0)	$f(x_0)$

Unlike the previous case studies, the schedule of traversals in this case-study depends on the constructed equation and differs from one another. Hence, manual fusion of such traversals is not practical, because it needs to be done for each equation separately based on the different operations in the equation. We constructed three

different equations that use different schedules of traversals, shown in Table 3.6. We evaluated the performance of the fused traversals for each of these equations on a balanced kd-tree constructed by uniformly partitioning the interval $[10^5,10^{-5}]$.

Figure 3.11 shows the performance of fused traversals corresponding to the first equation in Table 3.6 for different depths of kd-tree, normalized to the unfused ones.[15] [16] The fused traversals reduce node visits by 83%, and we see a 90% reduction in L2 cache misses. Overall, the fused traversals are faster, with a runtime improvement ranging from 15% for small trees, to 66% for large ones. Table 3.6 summarizes the performance of the corresponding fused traversals for each equation normalized to the unfused ones when performed on a balanced kd-tree of depth 20.

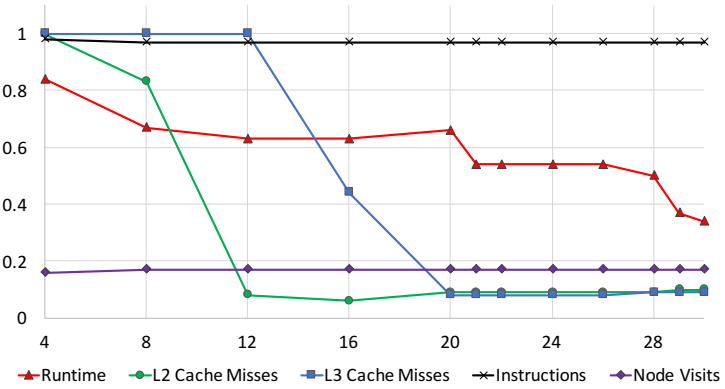


Fig. 3.11. Performance measurements for fused kd-tree traversals normalized to the unfused ones for different tree sizes.

3.5.4 Case Study 4: Fast Multipole Method

The fast multipole method (FMM) is a numerical technique used in evaluating pairwise interactions between large number of points distributed in a space (*e.g.* Long-

[15] Depth of the tree on the x axis, and normalized measurement on the y axis.

[16] 95% confidence intervals for kd-tree experiments are within ±1% for all reported measurements except for L3 cache misses of trees of depth 16, reported in Figure 3.11, where intervals are ±16%.

Table 3.6.
Performance improvement of different piecewise functions kd traversals fused by GRAFTER.

Equation	Runtime	Cache misses	Node visits
$x^4(f^{(2)}(x))^2 + \sum_{i=0}^{3} x^i$	0.66	0.09(L2) 0.51(L3)	0.17
$f^{(5)}(x)\|_{x=0}$	0.49	0.20(L2) 0.51(L3)	0.20
$\int_{-10^5}^{10^5} x^3(f(x) + .5)^2 \cdot u(0)$	0.88	0.33(L2) 0.65(L3)	0.33

ranged forces in the n-body problem, computation of gravitational potential, and computational electromagnetic problems) [28].

In this case study, we reimplement the FMM benchmark from TreeFuser in GRAFTER, which is based on the implementation from the Treelogy benchmark suite [31].

Figure 3.12 shows the performance of fused traversals for different input sizes. Grafter was able to fully fuse the two passes and yield a performance improvement up to 22% over the unfused version.[17] [18]

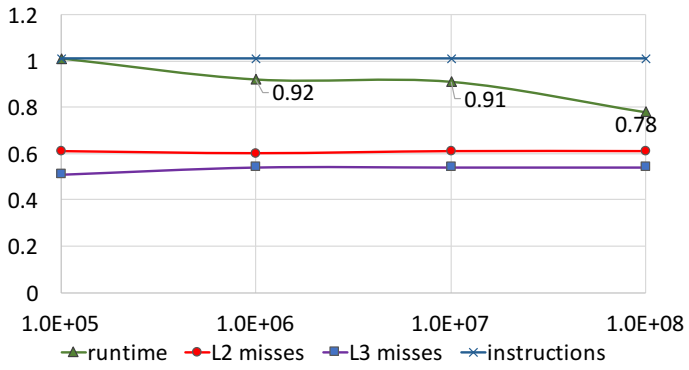


Fig. 3.12. Performance measurements for fused FMM traversals normalized to the unfused ones for different number of points.

[17] Number of points on the x axis and normalized measurement on the y axis.
[18] 95% confidence intervals for the FMM experiments are within ±1% for all measurements.

3.6 Conclusions

This chapter introduced Grafter, a framework for performing fine-grained fusion of generic tree traversals. In comparison with prior work, Grafter is either more general (in its ability to handle general recursion and heterogeneity), more effective (in its ability to perform more aggressive fusion), sound (i.e., without relying on programmer assumptions of fusion safety), or some combination of all three. We showed that Grafter is able to effectively fuse together traversals from two domains that rely on repeated tree traversals: rendering passes for document layout, and AST traversals in compilers. Not only can Grafter perform more aggressive fusion than prior work, it also delivers substantially better performance. Grafter allows programmers to write simple, ergonomic tree traversals while relying on automation to produce high-performance fused implementations.

4. GENERAL DEFORESTATION USING FUSION, TUPLING AND INTENSIVE REDUNDANCY ANALYSIS

This the third part of this thesis, unlike the previous two chapters, this work targets general fusion in functional programs. Fusion in functional programming has its challenges that are discussed in this section.

4.1 Abstract

Deforestation, is a classic optimization in functional programming that eliminates intermediate structures which arise during the evaluation of programs. Traditional deforestation techniques impose harsh syntactic restrictions on transformation candidates to guarantee fusion safety (i.e., termination and runtime complexity preservation). Due to those restrictions, compiler writers adopted less general, combinator-based fusion approaches, such as shortcut fusion.

Tupling is another transformation that combines functions traversing the same structure into one function. Previous work studied the relationship between tupling and fusion and suggested that tupling can be used to clean up the redundant work that results from *unsafe fusion* [36,37] that might otherwise not preserve complexity. However, none of the previous work that we know of implemented and evaluated such a transformation.

In this work, we demonstrate a practical implementation of combined fusion and tupling, in addition to a novel intensive redundancy analysis that is capable of fusing fairly complicated traversals, achieving significant speedup over unfused programs when compiled with the Glasgow Haskell Compiler using both lazy and strict evaluation.

4.2 Introduction

Fusion and *deforestation* are well-known transformations that can eliminate temporary intermediate data structures during the evaluation of functional programs. When performed correctly, fusion reduces data traversal overhead and memory usage, resulting in significant speedup.

In general, the goal of fusion is to take functions $f_1 : A \to B$ and $f_2 : B \to C$ where f_2 consumes the output of f_1 and produce a function $f_{12} : A \to C$. The simplest fusion example might be the composition of two functions *map* f_1 and *map* f_2, over some regular structure. The unfused program applies *map* f_1 on its input resulting in a new temporary structure that is then consumed by *map* f_2 to generate the final output. A fused version, on the other hand, directly generates the output from the input structure by applying *map* $(f_1 \circ f_2)$ on the input.

Fusion/deforestation transformations can be classified into two types. The first class is transformations that rely on the usage of predefined combinators that have well defined compositional behavior (the earlier map function for example), and a set of rewrite rules that optimize them. Shortcut fusion and stream fusion are examples of such transformations [11–15]. Those approaches are extremely useful when dealing with lists and simple data structures, where programs can be composed of fusable combinators. Such transformations are used in modern functional compilers such as the Glasgow Haskell Compiler, and in their popular data structure libraries. We term these approaches *shallow fusion*, because they do *not* reason about the recursive definitions of the combinators themselves, thereby greatly simplifying the problem at the cost of generality.

The second class, *deep fusion*, deals directly with fusing recursive functions, without baking in known traversals as primitive combinators. Deep fusion is general, but these techniques have proved difficult to automate in a practical way [10], and, as such, they have remained comparatively unexplored for the last two decades. The most popular deep approach is Wadler's deforestation [8], which guarantees programs

in *treeless* form can be fused safely. Treeless form, however, is very strict: functions must be linear, and no intermediate data structures are created during a single function evaluation—ensuring termination and complexity preservation. In his conclusion, Wadler states *"further* practical *experience is needed to better assess the ideas in the paper"*.

Without these restrictions, Wadler's program rewrites might not reach a fixed point and hence never terminate, or can generate programs that have higher runtime complexity. In follow-on work, Chin relaxed the treeless form by imposing more precise requirements on the fused functions [38]. Yet the scope of the programs satisfying those requirements remains narrow, with linearity restrictions and restrictions on intermediate structures. In his conclusion, Chin stated: *"The syntactic criteria proposed in this paper are based on safe approximations. They do not detect all possible opportunities for effective fusion, merely a sub-class of them"*.

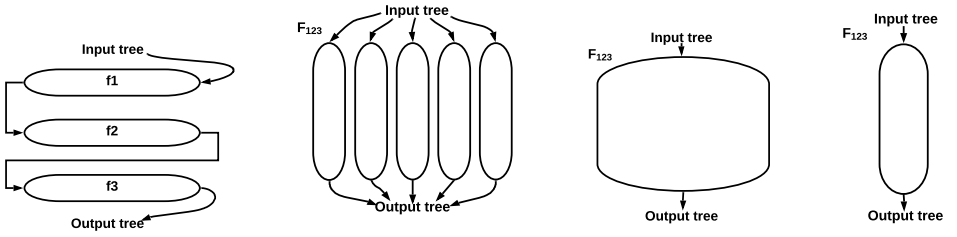
Complementing fusion, *tupling* is a transformation that combines functions that consume the same input into a single one [39, 40]: it transforms $f_1 : A \to B$ and $f_2 : A \to C$ into $f_{12} : A \to B \times C$. Chin studied the relationship between tupling and fusion, and suggested that tupling transformation can enable effective application of fusion with fewer restrictions on applicability [37]. More specifically, fusion may introduce multiple traversals of the same subtree when preformed on non-linear terms, but these additional traversals can be eliminated using tupling. Different combined transformation were suggested [9, 37]. However, the proposed approaches were not implemented or evaluated. Furthermore, syntactic restrictions for termination exist in those works.

In this paper, we propose a fusion transformation that does not require these syntactic restrictions—expanding the applicability of deep fusion. To avoid increasing runtime complexity, tupling and a novel intensive redundancy analysis are used to clean up redundant computations introduced during (unsafe) fusion. To guarantee termination, cut-off parameters limit the transformation and force it to backtrack under certain conditions.

This paper makes the following contributions:

- We propose a deforestation transformation that combines fusion, tupling, and intensive redundancy analysis, and which is guaranteed to terminate and may be incorporated in practical compilers.
- We implemented and evaluated our transformation in a real compiler that operates on a first-order language, with a Haskell backend, showing significant speedups on a large set of programs. This includes difficult-to-fuse examples such as rendering tree-structured documents (like HTML).
- We introduce a novel redundancy analysis techniques that are crucial to eliminating redundant work introduced by fusion and tupling for complicated programs.
- We show that the general, *deep* fusion is still a promising technique, one that—with good engineering—can fuse complicated programs which cannot be fused using competing techniques.

4.3 Background and Motivation



(a) Input program (b) Fusion (c) Tupling (d) Redundancy analysis

Fig. 4.1. Intuition behind the proposed transformation structure showing the major steps.

As mentioned in the introduction, shallow, combinator-based fusion approaches are useful when regular structures are traversed by standard patterns. More specifically, such fusion has shown great success in fusing list operations. However, the usage of such an approach is more limited when dealing with more complicated and less regular structures and traversals: e.g., document render trees and their traversals.

On the other hand, deep fusion approaches have either been unautomated [10], or work with a narrowly defined class of input programs that restricts their usage. Nevertheless, if in some way those restrictions can be relaxed, then we can achieve fusion for those more complicated and irregular traversals. To revive and make practical deep fusion requires surmounting two challenges: termination, and preserving runtime complexity.

Termination Non-termination during fusion can happen when the number of traversals that need to be fused increases with the depth of the traversed structure. For example, consider the program in Figure 4.2. The function $\text{mul}_{2\text{pd}}$ multiplies each element in the input list by 2^i, where i is it's index in the list. It does that by calling mul_2 in each recursion. Hence, each suffix of the list at index n is traversed n times by mul_2, and there are n traversals to fuse at each level.

Fusion does not terminate naturally when applied to this program, because after each step in the fusion process a new fusion opportunity that was not encountered before is created. When fusion is applied to this program it will start by fusing $\text{mul}_{2\text{pd}}$ with mul_2 and generates $\text{mul}_{2\text{pd}}\_\text{mul}_2$. In the body of the new fused function $\text{mul}_{2\text{pd}}\_\text{mul}_2$, an application $\text{mul}_{2\text{pd}}\_\text{mul}_2$ consumes an application of mul_2, fusion will try to fuse those and the process is repeated ad infinitum.

Our proposed transformation handles non-termination by cutting off the transformation at some certain threshold that is controlled by either the depth of the transformation or the number of fused functions. Although such a cutoff means that the fused code is not *treeless* (it contains intermediate structure at the cut-off location), we can still achieve speedups from performing such *uncompleted* fusion. Especially if

Input program:

```
mul2 []     = []
mul2 (x:xs) = (x * 2) : (mul2 xs)

mul2pd []    = []
mul2pd (x:xs) = (x * 2) : (mul2pd (mul2 xs))
```

Code after fusion:

```
mul2pd (x:xs) =
  (x * 2) : (mul2pd_mul2 xs)
mul2pd_mul2 (x:xs) =
  (x * 2) : (mul2pd_mul2_mul2 xs)
mul2pd_mul2_mul2 (x:xs) =
  (x * 2) : (mul2pd_mul2_mul2_mul2 xs)
 ...
```

Fig. 4.2. Example of a program with non-termination.

the structure is not linear; for example performing fusion up to depth 10 in a binary tree can eliminate all intermediate trees that are generated to compute the upper 2^{10} nodes in the tree.

Runtime complexity The other complication that arises from relaxing the syntactic restriction, is preserving runtime complexity. The treeless form imposed by Wadler [8] enforces that terms in the fused functions are linear. Such linearity guarantees that no repeated work gets introduced during the fusion process.

Non-linearity can cause fusion to replicate work or create traversals that did not exist in the original program, which can change the runtime complexity of the program. Consider for example the program in Figure 4.3. A list of integers is traversed by two functions; the first traversal (sum) aggregates the values in the list and the

next traversal shifts the elements to the left by dropping the first element and adding a zero to the tail of the list. This program has a runtime complexity of O(N).

Figure 4.3(d), shows the final result of fusion when performed on those two functions, the details of the fusion process is explained in the next section. Note that the fused function `shift_sum` has a runtime complexity of O(N^2)! For every element in the list, `shift_sum` calls `head_sum` (which traverses the complete list again) to sum the elements in the tail. Although the resulting program does not have explicit intermediate structures, it has additional traversals that did not exist in the input program.

Such increase in work can occur when the functions being fused are not linear. After fusion, all compositions of functions are eliminated. This means that all the computations operate directly on the input tree to generate some part of the output tree. And if the original functions are not linear it's possible that there would be multiple consuming locations for the input tree, and each of them can become a traversal. In the example above, `shift` is non-linear – it consumes the tail of the list in a recursive call and consumes it to get the first element in the tail. After fusion those two consuming locations became traversals, (`shift_sum`) and (`head_sum`). So what can we do to eliminate such overhead?

One key observation is that those traversals traverse the same structure and we can eliminate redundant work if we can combine them into a single large traversal that traverses the structure only once. Furthermore, since those are compositions constructed from the same set of original functions, there is a good probability of repeated work within these traversals that we need to get rid off; for example fxfy, fzfy, fyfz.. all might include some repeated work from fy! Merging those traversals into a single one makes it easier to detect and eliminate such redundancy.

Tupling is the transformation that combines functions traversing the same input structure into single function with tupled output. Tupling can eliminate the redundant traversals by traversing the input structure once rather than multiple times; furthermore, it brings work from different traversals closer to each and makes it eas-

```
data List = Cons Int List | Sing Int
sum :: List → List
sum ls = case ls of
    Cons value tail →
        let tail' = sum tail in
        let tailValue = head tail' in
        let value' = value + tailValue in
        Cons value' tail'
    Sing value  → Sing value
shift :: List → Int
shift ls = case ls of
    Cons value tail →
        let tail' = shift tail in
        let value' =  head tail in
        Cons value' tail'
    Sing value       → Sing 0
head :: List → Int
head ls = case ls of
  Cons value tail → value
  Sing   value      → value
prog = let ls = .. in
       let tmp = sum ls in
       shift tmp
```

Fig. 4.3. Program used as running example through out the paper.

ier to detect and eliminate redundancy. Our proposed transformation performs a tupling pass after the fusion pass. Figure 4.1(c) illustrates the effect of tupling, where the input is only traversed once, creating coarser-grained traversals.

Back to our example, we have two functions that traverse the tail: `head_sum` and `shift_sum`. The tupling pass combines those into one function, with the result shown in Figure 4.9(a). This tupled function brings the computations closer to each other

and, once simplified, yields the function in Figure 4.9(b). Details of the tupling pass are explained in the next section.

In the final tupled function, every list tail is consumed only once and the repeated computation is eliminated using a simple common subexpression elimination (CSE) pass that is integrated with tupling. The final runtime complexity is O(N) with no intermediate structures.

CSE is not always sufficient to eliminate redundancy, in more complicated programs an intensive redundancy analysis followed by several cycles of tupling might be needed, which is discussed in the next section.

Figure 4.3 visualizes the idea behind the proposed transformation. Fusion eliminates compositions of functions making all the operation in the fused function be written as consumers of the input tree. Tupling then combines those consumers to eliminate redundant traversals and brings computations closer to each other. Finally, redundancy analysis runs to unneeded computations.

4.4 Design

This section describes the details of the proposed transformations, we will start first by defining the language that the transformation assumes:

Our program transformation is defined on a first-order, pure language, with the grammar shown in Figure 4.4. We use the notation $\overline{x}$ to denote a vector $[x_1, \ldots, x_n]$, and $\overline{x_i}$ to denote the item at position i. To simplify presentation, primitives are dropped from the presented language. The language permits recursive data types, but as a strict, side-effect free language there can be no cyclic data values.

A program consists of a set of data definitions, function definitions, and the main expression. A function is a single case expression that destructs the first argument, which is assumed to be the dominant, traversed input.

The branches of the case expressions are sequences of flattened let expressions ending with leaf expressions – either a variable, a function application, or a constructor

$K \in$ Data Constructors, $\tau \in$ Types,

$f \in$ Function names $x, v \in$ Variables

Top-Level Programs	top	$dd\,;fd\,;ee'$
Type Scheme	ts	$\tau \rightarrow \tau$
Datatype Declarations	dd	$data\ \tau = K\ \tau$
Function Declarations	fd	$f : ts\ ;\ fb$
Function Definition	fb	$fx = case\ x_1\ of\ pat$
Pattern	pat	$K\ \ (x : \tau)\ \rightarrow\ ee'$
Let Expression	e	$let\ x : \tau = e'\ in\ ee'$
Leaf Expressions	e'	$fvK\ v\ v$

Fig. 4.4. Language definition

expression with variable arguments. This language is a simplified version of the actual language used in the implementation, which supports literals and primitives, and expressions need not be flattened.

4.4.1 Overview

Figure 4.5 illustrates the high level structure of the proposed transformation. The transformation converts an input expression to an optimized expression and generates a new set of functions during that process. The transformation consists of three main stages; fusion, tupling and redundancy analysis.

During the fusion step *compositions* of functions are fused and new functions that represent the compositions of fused functions are generated, the fusion process is then performed recursively on the new generated functions. Tupling runs next to combine traversals and eliminate redundant traversals, it's performed on each func-

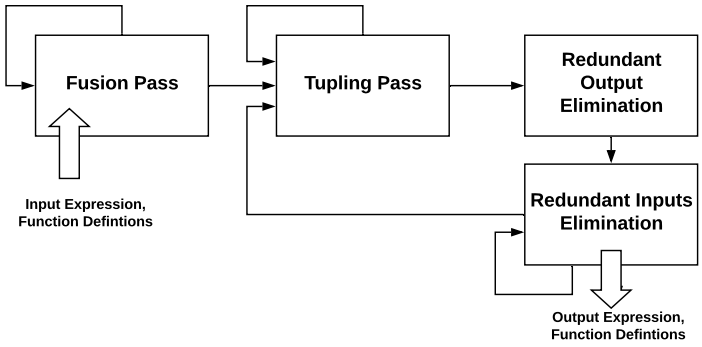


Fig. 4.5. High-level structure of the transformation showing the main passes.

tion that is generated during fusion, the tupling is also performed recursively on the new generated tupled functions. Redundancy analysis is then performed to eliminate redundant work. The process consists of two passes: output and input redundancy eliminations, respectively. Eliminating redundancy can allow more functions to be tupled, and hence tupling runs back-to-back with redundancy analysis until the process converges. A simplification pass serves to cleanup, and runs several times during the transformation. It performs common subexpression elimination and eliminates unused let bindings which is important to avoid optimizing dead code.

Cycles in this process can result in non-termination; the fusion recursion, the tupling recursion, and the tupling-redundancy cycle are some of them. We use a standard "fuel" approach to guarantee termination: the depth of the transformation is tracked along each cycle and threshold values are used to cut off and move to the next step.

In the rest of this section, each of the transformation steps is described in more detail.

4.4.2 Fusion

The goal of the fusion pass is to eliminate intermediate structures in the program. Figure 4.6 shows the structure of the fusion pass. It takes an expression and function definitions as input and returns a new fused expression and a possibly-larger set of function definitions.

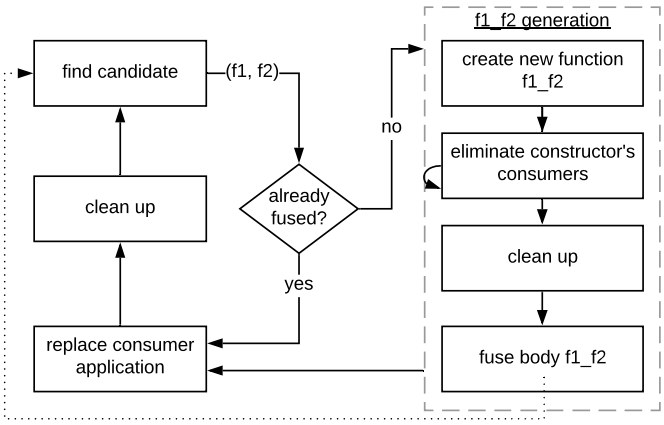


Fig. 4.6. Fusion pass diagram

The pass starts by identifying a fusion candidate in the processed expression, to this end it maintains a *def-use* table that tracks variables that are bound to function applications, and their consumers. More specifically a candidate for fusion (f_1, f_2) is a pair of functions that satisfies the following pattern:

```
let y = f1 x  ···  in
 ···  f2 x  ···
```

In such a case, (f_1, f_2) represents a fusion candidate and a new function f_2_f_1 that represents the composition is generated. We call f_1 *inner* (the producer) and f_2 *outer* (the consumer). Generating the fused function draws on previous fusion techniques [8,38]. However it's slightly altered to handle non-treeless expressions, and preserve the invariant that every function is a single case expression. This invariant makes the implementation of the optimization easier and more regular.

To illustrate the fusion process consider the previous example from Figure 4.3. Functions `sum` and `shift` are candidates for fusion. In this example, `sum` is the producer and `shift` is a consumer. As described in Figure 4.6 the first step is to create a new function `shift_sum` that represents the composition of `shift` and `sum`. It's created according to the following rules:

1. The output type of fused function is the output type of the consumer function.
2. The input type of fused function is a concatenation of the inputs of producer and the consumer excluding the first input of the consumer function.
3. The body of the fused function is the body of the producer with the consumer function applied to the output of every branch in the producer.

Figure 4.7(a) shows the result of the first step of fusion, a new function shift_sum is created with the appropriate types, and the body of the function is the the body of `sum`, with `shift` applied at each output location.

Next we partially-evaluate the body of the generated function with a pass that eliminates constructor consumers (similar to "case of known constructor"). This pass uses its def-use table to look for patterns of the form `let` x = (K ..) in ... f x

For each such pattern, the function application is replaced with the branch in f that corresponds to the constructor K after the appropriate instantiations. In our example `(shift out1)` and `(shift out2)` in Figure 4.7(a) are eliminated the result is shown in Figure 4.7(b). This sub-pass will keep running on the function until there are no further applications to known constructors.

After the new, fused function is generated, a cleanup pass will run, removing common subexpressions and unused let bindings: e.g. value', out1, and shift_value' in our Figure 4.7(b) example. The output of this step is shown in Figure 4.7(c).

Fusion is then performed recursively on the body of the new function, e.g., `shift_sum`. There are two candidates in this case (head, sum) and (shift, sum), the first will be generated through the same cycle while the later is already generated. For each candidate, after generating the new fused function, the consumer application is replaced

```
shift_sum :: List → List
shift_sum ls = case ls of
    Cons value tail →
        let tail' = sum tail in
        let tailValue = head tail' in
        let value' = value + tailValue in
        let out1 = Cons value' tail' in
        shift out1
    Sing value  →
        let out1 = Sing value in
        shift out1
```

(a) Create a new composed function

```
shift_sum :: List → List
shift_sum ls = case ls of
    Cons value tail →
        let tail' = sum tail in
        let tailValue = head tail' in
        let value' = value + tailValue in
        let out1 = Cons value' tail' in
        let shift_tail' = shift tail' in
        let shift_value'  =  head tail' in
        Cons shift_value' shift_tail'
    Sing value  →
        let out2 = Sing value in
        Sing 0
```

(b) Eliminate constructor's consumers

```
shift_sum :: List → List
shift_sum ls = case ls of
    Cons value tail →
        let tail' = sum tail in
        let tailValue = head tail' in
        let shift_tail' = shift tail' in
        Cons tailValue shift_tail'
    Sing value  → Sing 0
```

(c) Clean expression

```
shift_sum :: List → List
shift_sum ls = case ls of
    Cons value tail →
        let tailValue = head_sum tail in
        let shift_tail' = shift_sum tail in
        Cons tailValue shift_tail'
    Sing value  → Sing 0

head_sum :: List → Int
head_sum ls = case ls of
    Cons value tail →
        let value' = head_sum tail in
        value' + value
    Sing value → value
```

(d) Fuse the body of the new function

Fig. 4.7. Different steps of fusion pass performed on the running example.

with an application of the new fused function and a clean pass runs to eliminate the producer application when its *last* consumer is eliminated. The final result of fusion is shown in Figure 4.7(d).

There are two cycles in the fusion pass, the recursive fusion, and the recursive constructor inlining. Each of those is tracked and bounded as described earlier to guarantee termination.

4.4.3 Tupling

Tupling combines traversals that traverse the same structure and bring computations closer to each other. Tupling is performed after fusion to eliminate redundant work that is introduced during fusion. For tupling, we extend the intermediate language to include operations on tuples. New expression forms are added for constructing tuples and projecting elements from tuples, plus a new product type.

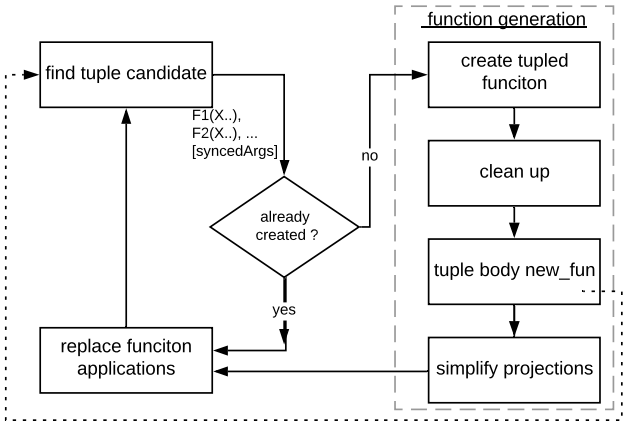


Fig. 4.8. Tupling pass diagram

Figure 4.8 summarizes the tupling pass, which begins by finding a tupling candidate. A candidate is a set of *independent* function applications that all traverse the same input (have the same first argument in our language).

By independent we mean that none of them directly nor indirectly consumes the other. For example in the code below, calls to f_1 and f_2 are not tupleable because f_2 indirectly consumes f_1 through the intermediate variables y.

```
let x = f1 tree in
let y = x + 1 in
let z = f2 tree y in
```

where as f_1 and f_2 are candidates for tupling in this program.

```
let x = f1 tree in
let y = 1 in
let z = f2 tree y in
```

In the running example, the two applications, `(shift_sum tail)` and `(head_sum tail)` in the body of fused function `shift_sum` shown in Figure 4.7(d) are a candidate. For each candidate, a tupled function is generated. Figure 4.9(a) shows the tupled function `shift_sum_T_head_sum` from the running example. The tupled function is generated according to the following rules:

```
shift_sum_T_head_sum::List→(List, Int)
shift_sum_T_head_sum ls = case ls of
    Cons value tail →
        -- computations from shift_sum
        let tailValue = head_sum tail in
        let tail' = shift_sum tail in
        let o1 = Cons tailValue tail' in
        -- computations from head_sum
        let value' = head_sum tail in
        let o2 = value' + value in
        (o1, o2)
    Sing value →
        let o1 = Sing 0 in
        (o1, value)
```

(a) Create merged function

```
shift_sum_T_head_sum::List→(List, Int)
shift_sum_T_head_sum ls = case ls of
  Cons value tail →
     let p = shift_sum_T_head_sum tail in
     let o1 = Cons (proj 1 p) (proj 0 t) in
     let o2 = (proj 1 p) + value in
     (o1, o2)
  Sing value →
     let o1 = Sing 0 in
     (o1, value)
```

(b) Clean and tuple recursively

Fig. 4.9. Code generated during tupling the running example

1. The input type of the tupled function is the type of the traversed tree followed by the remaining inputs of each of the participating functions. move a note about doing this during tupling to the redudancy analysis to save space

   One important optimization at this point is to analyze shared inputs, and include them only once in the input list of the tupled function. This guarantees that shared inputs are bound to the same local variable, and hence allows better redundancy analysis. In some cases that also can avoid non-termination by eliminating identical tupleable applications that won't be seen identical otherwise.

2. The output type of the tupled function is a tuple of the output types of the participating functions, with nested tuples flattened.

3. The body of the tupled function is a single case expression that destructs the traversed tree. For each case branch the body of the corresponding branch in each of the tupled functions is bound to a variable and a tuple of those variables is returned. Figure 4.9(a) illustrate the final result of this step.

Next, this new function is optimized through a cleanup pass. In our example the redundant application `head_sum tail` which appears twice in the new tupled function (Figure 4.9(a)) is eliminated. Tupling is then performed recursively on the body of the new function.

At the end of the process, the original function applications that are tupled are eliminated by replacing the first application with the tupled function and the rest with projections to extract the corresponding output. Figure 4.9(b) shows the final result of tupling in our running example.

It's possible for tupling to diverge when the number of the tupleable applications increases with the depth of the tree. Thus, the depth of the recursive tupling cycle has to be tracked to guarantee termination.

Tupling reverts the runtime complexity of the fused program in our running example back to O(N), since redundant traversals are eliminated. However, it's not always enough for the cleanup pass to eliminate the redundant computations. Furthermore, as the traversals get more and more complicated, certain redundancies can create avoidable dependencies that prohibit tupling. The next section describes the redundancy analysis pass which performed a more intensive analysis to overcome those issues.

4.4.4 Redundancy Analysis

Following tupling, redundancy analysis is performed to further optimize the tupled functions. The optimizations performed during this pass are classified into two types; *redundant outputs* and *redundant inputs*. Each is described in detail in this section.

Note that as illustrated in Figure 4.5, tupling is performed again after redundancy analysis, since eliminating redundancy can enable more tupling to be done by eliminating some dependences that prohibit tupling.

Redundant outputs

The redundant outputs pass eliminates outputs of functions that appear at different indices in the tupled output but always have the same value.

Function f_t in the code bellow illustrates such redundancy in its simplest form. The output of f_t is always the same for positions 0 and 1. We will use the notation $f_t^{0=1}$ to refer to that property throughout the section.

```
f_t :: List → (Int, Int)
f_t ls = case ls of
    Cons value tail →
        let ret = depth tail in
        (ret, ret)
    Sing value → (0, 0)
```

Different circumstances can cause such redundancy to originate. For example, consider tupling two fused functions, $f_x f_y$ and $f_x f_z$. If the result of f_x does not depend on f_z nor on f_y, then both functions would have the same output.

Eliminating such redundancy is important for two reasons. First, if this function is called recursively, then the memory and runtime overhead of creating such a tuple is eliminated. The second important effect of such elimination is that it allows more optimizations on the caller side by leveraging the fact the the two outputs are the same to further eliminate redundant traversals and expressions.

For example, consider the function f_x below that calls function f_t (illustrated earlier), where $f_t^{0=1}$. Function f_x calls f_y twice on each of the outputs of f_t.

```
f_x :: List → (Int, Int)
f_x ls = case ls of
  Cons value tail →
    let p  = f_t tail in
    let o1 = f_y (proj 0 p) in
    let o2 = f_y (proj 1 p) + 100 in
    (o1, o2)
```

However after eliminating the redundant output of f_t, a cleanup pass will be able to identify that the two calls to f_y are the same and hence eliminate one of them. The code below shows the result of the optimization. The second output is eliminated from f_t, and the caller of f_x is redirected to read the second output from the first. A cleanup pass then is used to eliminate the redundant application of f_y.

```
f_t :: List → (Int)
f_t ls = case ls of
    Cons value tail →
```

```
        let ret = depth tail in
        (ret)
    Sing v → (0)

f_x :: List → (Int, Int)
f_x ls = case ls of
  Cons value tail →
    let p = f_t tail in
    let y = f_y (proj 0 p) in
    (y, y + 100)
```

Redundant output elimination consists of three steps:

1. Identify redundant outputs.
2. Create a new function with redundant outputs eliminated.
3. Fix callers to call the new function and optimize them.

For each tupled function, each two output positions are checked for redundancy, then the function is rewritten by eliminating the redundant locations. This is done by updating the output type and the output expressions at each output tuple. Next, call sites of the original functions are updated such that projections of the redundant position are projecting the retained position rather than the eliminated redundant position.

Steps two and three are direct manipulation and rewrites. However, identifying redundant outputs is not always as trivial as in the previous example.

Inductive Redundant Output Analysis. In the previous example, it was easy to identify that the outputs at positions 0 and 1 are the same, by simply inspecting the output of each branch. However the process is not always that simple. Due to mutual recursion and complicated traversal patterns, a more rigorous inductive analysis is needed.

Consider the following slightly more complicated example of two mutually-recursive functions, f_1 and f_2.

```
f1 :: List → (List, List)
f1 ls = case ls of
  Cons value tail →
    let p = f2 tail in
    let o1 = Cons (v+1) (proj 0 p)in
    let o2 = Cons (v+1) (proj 1 p) in
    (o1, o2)
  Sing 0 → (Sing 0, Sing 0)

f2 :: List → (List, List)
f2 ls = case ls of
  Cons v tail →
    let p= f1 tail in
    let y1 = Cons (v*2) (proj 0 p) in
    let y2 = Cons (v*2) (proj 1 p)  in
    (y1, y2)
  Sing 0 → (Sing 0, Sing 0)
```

Looking closely at those two functions, we observe that the second output of f_1 and f_2 is redundant and matches the first output. But how can we verify that soundly and systematically?

We want to check if f_1 always returns the same output at indices 0 and 1. In other words, if $f_1^{0=1}$ is satisfied. We can do that by checking the output at each branch. In this example the the following two equalities should be satisfied: (Sing 0 == Sing 0) and (o1 == o2).

If the application of f_1 is a leaf function application (with respect to the execution call stack) then (Sing 0 == Sing 0) should be satisfied, otherwise if it is a non leaf application, then (o1 == o2) should hold.

Verifying that (o1 == o2) is equivalent to verifying that Cons (v+1) (proj 0 p) == Cons (v+1) (proj 1 p), which is true only if (proj 0 p == proj 1 p)—in other words, if $f_2^{0=1}$ is satisfied (since p is bound to f_2 function application).

More precisely, for $f_1^{0=1}$ to be satisfied during a non-leaf application at depth l, $f_2^{0=1}$ need to be satisfied for depth $l+1$. In a similar a way $f_2^{0=1}$ is satisfied if $f_1^{0=1}$ is satisfied for the next depth.

We can use induction to show that $f_1^{0=1}$ is satisfied, under the assumption that the program terminates, as follows:

Base Case: $f_1^{0=1}$ and $f_2^{0=1}$ are satisfied during a leaf function application, since (Sing 0 = Sing 0).

Induction hypothesis: Assume that $f_1^{0=1}$ and $f_2^{0=1}$ holds at depth ¿ l.

Induction step: $f_1^{0=1}$ and $f_2^{0=1}$ are satisfied during non leaf application at depth l as a consequence of the induction hypothesis as discussed earlier.

We propose a process through which a compiler can conclude that two outputs of a given functions at two different locations are always the same. The process checks all the conditions that are needed to construct an inductive proof similar to the previous proof.

We will first describe the process and then explain its soundness. We will use the example above to illustrate the process, to verify $f_1^{0=1}$.

The process tracks two sets of properties, $S1, S2$, the former for properties that need to verified and the latter for the properties that are already verified. A single property is of the form $f_1^{0=1}$. In our example, at the beginning of the process $S1 = \{ f_1^{0=1} \}$ and $S2 = \{\}$.

The process will keep pulling properties from $S1$ and checks for two things:

*Check*1**:** Whether the property is satisfied during a leaf application of the function (leaf with respect to to the call stack).

*Check*2**:** Wether the property is satisfied during a non-leaf application at level l under the assumption that all properties that need to be satisfied at depth $l+1$ are satisfied.

If the two checks are satisfied, then the set of properties that need to be satisfied at depth $l+1$ of the call stack for the current property to be satisfied at level l (the assumptions in *check2*) are extracted. Those properties will then be added to $S1$ and the condition that was checked will be moved to $S2$. If a condition already exists in $S2$, then it does not need to be added to $S1$ again since it is already verified.

Let us explain how the process work more specifically: For each condition $f_x^{y=z}$ in $S1$, the following will be performed (in our example those are performed first on $f_1^{0=1}$):

1. All let bindings in f_x are inlined in the output tuples except variables that are bound to function application of other tupled functions. In our example, this will result in the following function body:

```
Cons value tail →
  let p = f₂ tail in
   (Cons (v+1) (proj 0 p),
   Cons (v+1) (proj 1 p))
Sing 0 → (Sing 0, Sing 0)
```

2. Each expression in the output tuple is parametrized around the indices of the projections that appear inside it. The code below shows the result of this step.

```
Cons value tail →
  let x = f₂ ls in
  (Cons (v+1) (proj p0 x where p0 = 0),
   Cons (v+1) (proj p0 x where p0 = 1))
Sing 0 → (Sing 0, Sing 0)
```

   We do such parametrization by traversing the expression in a deterministic manner such that if two expressions are the same but have different projection indices then those indices will have place holders of the same id (p0 in the code below).

3. Check that the expression at indices y and z from the property ($f_x^{y=z}$) are the same in each output in the function. Place holders in the parametrized ex-

pressions (p0 in our example) are considered equivalent if they have the same id. If this check fails then our algorithm considers the output locations as not identical and terminates. Otherwise, this step will satisfy the two checks *check*1 and *check*2 that are mentioned earlier for the property $f_x^{y=z}$. More specifically, a leaf application of f_x satisfies $f_x^{y=z}$ since we have checked that the outputs are the same on each possible output branch. And for non-leaf application of f_x, $f_x^{y=z}$ is satisfied assuming that the properties that need to be satisfied during the next level of the call stack are satisfied. Those properties are determined in a way that guarantees that projections from the same place holders are always the same, and in such cases, the output expressions are the same. (The next step explains how those required properties are determined.)

4. Determine the set of properties that need to be satisfied at level $l + 1$. This is done by inspecting the corresponding pairs of place holders in the output expressions at indices y and z. In our example, we only have one (p0 = 0 and p0 = 1, and they are projecting from x, which is bound to f_2). Hence the property $f_2^{0=1}$ is generated as a requirement for $l + 1$.

5. The checked property is then moved from $S1$ to $S2$, and each property that is generated in the previous step is added to $S1$ if its not already in $S2$.

6. The process is repeated on the next property in $S1$ until either $S1$ becomes empty or an unsatisfied condition is encountered. If $S1$ becames empty, then the starting property $f_x^{y=z}$ and all properties in $S2$ hold.

In our example, the process will be repeated for the condition $f_2^{0=1}$, then this condition will be moved to $S2$ and $f_1^{0=1}$ wont be added to $S1$ since its already in $S2$. Hence, at the end of the process the compiler knows that $f_1^{0=1}$ and $f_2^{0=1}$ hold.

Theorem 4.4.1 *At the end of the process described earlier, if* $S1$ *is empty then all the properties in* $S2$ *are satisfied.*

Proof We will prove the theorem by induction on the depth of the call stack during the evaluation of any of the functions in $S2$. The call stack is considered to expand at applications of tupled function. The proof assumes that the program terminates.

Base Case. Every property in $S2$ is satisfied if its evaluation is a leaf application in the call stack. This follows directly from *check*1 mentioned earlier, which is performed during step 3 in the process above on each property before it is moved to $S2$.

Induction hypothesis. Assume that each property in $S2$ holds at depth ¿ l.

Induction step. We can show that at depth l any property in $S2$ holds as the following:

(1) Any property in $S2$ is satisfied if all properties that need to be satisfied at $l+1$ hold. This is done at step 3 for every property before it is moved to $S2$.

(2) For any property in $S2$, all the properties that need to be satisfied at $l+1$ are also in $S2$. During step 4 all such needed properties are generated and added to $S1$ unless they are already in $S2$. This means that any property that needs to be satisfied is already in $S2$, since everything in $S1$ eventually moves to $S2$ and $S1$ is empty.

(3) By the induction hypothesis, all properties in $S2$ hold at depth $l+1$, thus from (1) and (2) all properties in $S2$ holds at depth l. ■

The output of the optimized mutual recursion is shown in the code bellow. The output program traverses the tree on time instead of traversing every branch twice recursively.

```
f₁ :: List → (List)
f₁ ls = case ls of
  Cons value tail →
    let p = f₂ ls in
    let o1 = Cons (v+1) (proj 0 p)in
    (o1)
```

```
  Sing 0 → (Sing 0)

f₂ :: List → (List, List)
f₂ ls = case ls of
  Cons v tail →
    let p = f₁ ls in
    let y1 = Cons (v*2) (proj 0 p) in
    (y1)
  Sing 0 → (Sing 0)
```

Redundant inputs

The redundant inputs pass targets eliminating inputs of functions when they are not needed. Eliminating such inputs removes the overhead of passing them, especially in recursive functions. It also allows better optimization on the callee and the caller site by possibly eliminating related computations. Furthermore, it can eliminate dependences and allow more tupling. This section will describe several types of redundant inputs that are handled in our transformation.

Shared inputs Function applications that consume the same input at different input positions can be optimized by unifying such arguments into one argument. Doing so enables more optimization in the body of the function, since all arguments will be bound to the same local variable.

Although this optimization is performed during tupling, it is performed here again because the output redundancy pass can result in more shared input opportunities.

Unconsumed inputs Unconsumed inputs are inputs that are not used in the body of the function that consumes it. Fusion can result in such inputs. For example, consider a fused function, $\texttt{f}_x\texttt{f}_y\texttt{a b}$, where b is originally an argument of $\texttt{f}_y$. If the

input b is not needed to construct the output of f_x, then although it is used in f_y, that usage will be eliminated in the body of $f_x f_y$.

Non-recursively consumed inputs This pass eliminates inputs that are returned as output without being further consumed in the function. Thus the caller can be rewritten to use them directly. The code below shows an example of such inputs.

```
f2 :: List → List → List
f2 ls1 ls2 = case ls1 of
  Cons v tail →
    let z = f1 tail in
    (Cons (v*2) z, ls2) in
  Sing 0 → (Sing 0, ls2)
```

4.5 Limitation Discussion

Unlike previous work, one of the goals is of this paper is to show that syntactic restriction needs not to be followed for general fusion, which can help us to get significant performance improvements on a larger set of applications.

The syntactic restriction that establishes limitations has been studied thoroughly for individual optimizations. However, performing such an analysis for a transformation that is composed of different sub-optimizations is extremely hard. Instead, an experimental and intuition-based approach is followed in this work.

Back to discussing the limitations of the transformation; our work (Unlike previous) can handle non-termination and non-linearity in isolation. However, an issue might arise when a program has both of these properties at the same time. The AST benchmark in section 4.7 is an example of such a program.

The argument for that follows from the following intuition, first non-termination by itself is not an issue since our framework handles that by trimming the transformation, also when its combined with linear terms that wont be a problem since no repeated work would be created in that case.

Second nonlinear terms, when combined with fusion that naturally terminates are also not an issue. Since fusion will guarantee that all function applications are consuming the same input as illustrated in figure 4.1(b), and hence tupling will be capable of combining them and eliminating redundant work.

However, when non-linear terms are combined with non-naturally terminated fusion, the following can happen. Fusion does not guarantee in such a case that all the function applications are written in terms of the same input tree (compositions do still exist and the state in figure 4.1(b) might not be achieved). But non-linearity, on the other hand, can introduce redundant work that in such case is not guaranteed to be tupled since not all the function applications are written in terms of the same input. (Figure 4.3) . The failure of tupling any redundant traversals will result in more work to do and worse performance.

4.6 Implementation

The proposed transformation was implemented as a pass in Gibbon [41] a compiler for a small subset of Haskell. Gibbon has a Haskell front end and can be prompted to output the transformed program into Haskell output. Hence, we used Gibbon to perform Haskell source-to-source transformation.

To control termination, all cycles in the transformation are controlled by a maximum depth of 10 in all the reported experiments unless otherwise noted, the max depth threshold can be overridden using a compiler flag.

4.7 Evaluation

We evaluated our transformation on a large set of programs showing its ability to fuse them, achieving better performance and lower memory usage.

We divided the evaluated programs into two sets: a set of programs inspired by previous related work, and a set of more complicated programs that involve larger traversals. For each experiment, we evaluated the output Haskell programs in lazy and

strict modes. Strict mode is achieved via the `Strict` pragma in GHC. We also report an experiment that measures the effect of each major pass in the transformation. And finally, we discuss a case in which our transformation was not able to consistently achieve a speedup.

Experimental platform: We ran our experiments on a Intel Xenon E5-2699 CPU, with 65GB of memory running Ubuntu 18.04 and GHC 8.8.1. All programs are compiled using the `-O3` optimization level and the runtime numbers are collected by taking the average of 10 program executions.

4.7.1 Surveyed Simple Programs

The first set of programs is extracted and inspired from previous work, each one of those programs is a composition of two functions. Table 4.1 contains the evaluated programs – programs 1-4 operate on lists while programs 5-9 operate on trees. Most of the programs are self explanatory, while programs 3,4 and 5 are explained in section 4.2. Program 10 consumes a list of lists.

For each program, we show runtimes that correspond to the fused and the unfused versions in both lazy and strict modes. In addition to that, three properties are shown: natural termination, linearity, and whether the program is in treeless form. Programs not in treeless form are unsafe fusion candidates in Wadler's deforestation [8].

Under strict evaluation, the fused version has better performance for most programs and does not introduce any slowdown, with speedups up to more than 5X . One the other hand under lazy evaluation a runtime regression is caused by fusion for three programs. One reason for that is that for some programs, fusion happens naturally during lazy evaluation where the order of evaluation matches the order of the fused computation. In such cases the overhead that is associated with syntactic fusion is not justified. Specifically the overhead due to tuples packing and unpacking, and the introduced coarser-grained traversals. Program 9 is an interesting case; the function flipRec flips each tree at depth d, d times. Fusion does not terminate

naturally on the program, however when truncated at depth 10 it eliminates all the additional traversal up to that level, and for a tree of depth 13 that is eliminating almost all of the work, achieving more than 100X speedup. Overall for programs in table 4.1 fusion achieves speedups with geo-mean of 2.4 in lazy evaluation and 2.6 in strict evaluation.

Table 4.1.
Comparison of the runtime of the fused and unfused programs under lazy and strict evaluation. Programs in this table are ported or inspired form previous work.

	natural termination	**linear**	**treeless**	**lazy**		**strict**	
				unfused	**fused**	**unfused**	**fused**
1. *append (append ls)*	✓	✓	✓	0.47s	**0.42s**	1.51s	**1.28s**
2. *sum (square ls)*	✓	✓	✓	0.37s	**0.25s**	0.99s	**0.36s**
3. *shift (sum ls)*	✓	✗	✗	**22.9ms**	31.8ms	11.4ms	**7.0ms**
4. *mul2pd ls*	✗	✓	✗	2.56s	**2.38s**	0.60s	**0.58s**
5. *mul2pd tree*	✗	✓	✗	1.56s	**0.42s**	0.89s	**0.32s**
6. *seteven (sumup tree)*	✓	✗	✗	1.11s	**1.07s**	0.95s	**0.52s**
7. *sum (flatten tree)*	✓	✓	✓	**0.69s**	0.8s	1.66s	**1.3s**
8. *flip (flip tree)*	✓	✓	✓	0.53s	**0.28s**	0.68s	**0.48s**
9. *flipRec (flipRec tree)*	✗	✓	✗	3.18s	**2ms**	0.75s	**1ms**
10. *sum (flatten mtrx)*	✓	✓	✓	**1.16s**	1.52s	1.37s	**1.35s**

4.7.2 Larger Programs

In this section we consider another set of programs that are larger and more application oriented:

Render Tree Render trees are used in render engines to represent the visual components of the rendered document. A render tree is consumed by different functions to compute the visual attributes of elements of the document. We implemented a render tree for a document that consists of pages composed of nested horizontal and

vertical containers with leaf elements (TextBox, Image, etc.). We implement five traversals that traverse the tree to compute height, width, positions and font style of the visual elements of the document. Each traversal consists of a set of mutually recursive functions. In total, the program consists of more than 40 functions with more than 400 lines of code. Table 4.2 shows four entries for the render tree, fusing 4 passes and fusing 5 passes with two different inputs. Fusion reduces memory usage and achieves speedups up to 3X for all programs under both lazy and strict evaluation.

Piecewise Functions Kd-trees can be used to compactly represent piecewise functions over a multi-dimensional domain. The inner nodes of the tree divide the domain of the function into different sub-domains, while leaf nodes store the coefficients of a polynomial that estimates the function within the node's sub-domain. In this program, we implemented a kd-tree for single variable functions, and different traversals to construct and perform computations on these functions such as adding constant, multiplying variable and adding two functions. Table 4.2 shows the speedups for three different programs that are expressed using different compositions of those functions along with the corresponding equations. A binary tree of depth 22 is used to represent those functions.

Fusion achieves up to 5X speedups on those programs and significantly reduces the memory usage. The third program has a relatively lower speedup than the first two, and the reason is that the function that adds two piecewise functions consumes two trees, but our fusion performs fusion across one of them only.

Effect of different passes. Table 4.3 shows the runtime of the fused programs when the transformation is truncated at its major three stages: fusion, tupling, and redundancy analysis. Render tree is the most complicated program, and it utilizes both tupling and redundancy analysis to achieve speedups especially in strict mode. Simpler, non-linear programs need tupling only to eliminate redundancies and achieve

Table 4.2.
Comparison of the runtime and total memory allocated of different fused and unfused programs under lazy and strict evaluation.

	lazy		**strict**	
	unfused	**fused**	**unfused**	**fused**
4 render tree passes [input 1]	1.02s — 767MiB	**0.53s** — 551MiB	1.14s — 461MiB	**0.47s** — 228MiB
4 render tree passes [input 2]	3.79s —2.52GiB	**2.26s** — 1.85GiB	2.24s —1.53GiB	**1.23s** — 823MiB
5 render tree passes [input 1]	1.25s — 968MiB	**0.49s** — 590MiB	1.63s — 583MiB	**0.63s** — 291MiB
5 render tree passes [input 2]	4.97s — 3.16GiB	**2.16s** — 2.01GiB	4.24s —1.924GiB	**1.73s** — 1.04GiB
piecewise functions 1 ($f_1 = x^3 + x^2 + x + 1$)	5.06s — 6.37GiB	**0.88s** — 2GiB	4.99s — 4.71GiB	**1.65s** — 2.75GiB
piecewise functions 2 ($f_2 = x^2 + x$)	3.55s — 4.78GiB	**0.76s** — 1.9GiB	3.71s — 3.96GiB	**1.53s** — 2.65GiB
piecewise functions 3 ($f_3 = (f_1)^2 + f_2$)	15.0s — 28GiB	**5.56s** — 19GiB	12.1s — 11.28GiB	**6.00s** — 6.59GiB
5 binary tree traversals	4.12s — 2.5GiB	**3.10s**— 1.46GiB	2.08s — 864MiB	**0.77s** — 480MiB

speedups. Finally, although the piecewise functions program is large and not trivial, due to its linearity it only requires fusion to achieve its speedup.

Does it always work? There is no guarantee that this transformation is always safe from a runtime perspective. Although for strict evaluation the transformation does not reduce the runtime for almost all the benchmarks, we encountered one case where the performance of the fused program varies between 2x speedup and 2x slowdown for different inputs. We implement a sequence of 7 functions that optimize and evaluate first-order lambda calculus expression. The program's traversals are complicated from a fusion perspective and hard to fuse. Specifically because we are dealing with expressions only, not functions, fusion opportunities are less likely to be found at that level.

For that specific program, a threshold of 10 for the depth of the transformation was too large for the transformation to terminate in a reasonable time. Furthermore, the code size grows very quickly since the number of different compositions of functions and traversed structures can get very large. Future work includes more investigation to analyze that benchmark and determine the causes of the slowdowns for some inputs and whether it's something that can be handled by the transformation.

Table 4.3.
Runtime of the fused programs when the transformation is truncated at the its three main satges.

	Unfused		**Fusion**		**Fusion + Tupling**		**Fusion + Tupling + Redundancy Elimination**	
	lazy	**strict**	**lazy**	**strict**	**lazy**	**strict**	**lazy**	**strict**
4 render tree passes [input 2]	3.79s	3.24s	2.23s	3.22s	4.07s	1.68s	**0.76s**	**0.46s**
5 render tree passes [input 1]	1.25s	1.63s	0.74s	1.15s	0.70s	1.05s	**0.49s**	**0.63s**
shift (sum ls)	**22.9ms**	11.4ms	58.6s	45.6s	31.8ms	**7.0ms**	31.8ms	**7.0ms**
5 binary tree traversals	4.12s	2.08s	**1.87s**	2.46	3.10	**0.77**	3.10	**0.77**
piecewise functions 3	15.02s	12.10s	**6.82s**	**5.64s**	**6.82s**	**5.64s**	**6.82s**	**5.64s**

4.8 Conclusion

Deforestation is an important optimization in functional programs due to their stateless nature. Practical fusion optimizations that are adopted by compilers utilize combinator-based fusion techniques. While those are easy to implement, they address a narrow set of fusion opportunities, and require programs to be built using specific combinators.

In this work we propose and implement a practical fusion transformation that operates directly on general recursive functions. We utilize fusion, tupling and redundancy analysis to increase the applicability of such transformations and mitigate or eliminate any performance side effects. The proposed transformation shows significant speedup over GHC optimized Haskell code. We hope that this work will inspire and motivate more work to be done on practical, general deforestation techniques.

5. RELATED WORK

5.1 Fusion for imperative programs

5.1.1 Fusion for regular programs

Fusion and similar transformations form the basis for many crucial compiler optimizations. In the world of *regular* programs (programs that operate over dense matrices and arrays), *loop fusion* is a common optimization that improves locality (by reducing the number of times that an array or matrix is accessed) by merging the bodies of two (or more) loops together [42–44]. Perhaps the most popular framework for performing fusion is the *polyhedral framework* [45–47], which uses a high level dependence representation that allows compilers to reason about the dependences between loops and explore different fusion possibilities prior to synthesizing code to implement a fused schedule. These approaches are not suitable for traversal fusion, as the dependence representation applies to loops over arrays with affine access functions.

5.1.2 Fusion for irregular programs

The most directly related work, however, optimizes series of traversals, typically by fusing together multiple passes. Different systems where proposed, in these systems, the programmer aids the tool by expressing their tree traversals in a structured form.

Miniphases For example, in *miniphases* [3], the programmer writes an AST transformation as a collection of "hooks" or callbacks that observe individual syntax nodes and act on them, while abstracting the recursion over the tree so that it is handled by the underlying framework. The user manually groups these miniphases into

tree traversals, but there is no guarantee of equivalence between fused and unfused executions—that soundness burden is on the programmer. This style of decomposing passes is often approximated in many compiler implementations—strategies for it are part of compiler writer folklore. It also appears explicitly in software frameworks such as the a *nanopass framework* [2].

Attribute grammars Another specialized way to express tree traversals is by using attribute grammars [48]. In recent work, [18] showed how complex web page rendering passes can be expressed using attribute grammars and then scheduled using a combination of fusion and parallelization [18]. While their approach is highly effective, attribute grammars are restrictive in the kinds of traversals they can express: a single complex traversal might require a careful decomposition into several interlocking attributes. Moreover, existing attribute grammar scheduling frameworks do not support partial fusion.

Tree transducers Attribute grammars are a special case of *tree transducers*, which are automata that recognize tree languages (equivalently, traverse trees) and produce an output as they go [49,50]. *Macro* tree transducers allow for more arbitrary computations by allowing the transducer to accumulate context as it traverses the tree [51]. There has been a significant investigation into the composition of various types of tree transducers [52–54]. Tree transducers, like attribute grammars, require expressing traversals in a particular formalism, in contrast to our more natural, code-like expression. Moreover, because our language for fusion admits fairly general computations and interleaving of computation and traversal, we believe that our approach covers a more general class of problems than tree transducers, though this requires further study. Finally, to our knowledge, there is no analog in transducer composition to our notion of partial fusion.

However, to our knowledge, none of these approaches handle as general a class of programs and fusion opportunities as Grafter: for example, we are not aware of a tree transducer solution for partial fusion.

Fusion in MADNESS simulation package [4,5] looked at fusing kd-tree traversals in the MADNESS simulation package. They capture the various kd-tree passes in MADNESS using a high-level representation and then reason about when fusion is legal using a dependence test similar to that of [24]. Rajbhandari et al.'s abstraction only supports pre- and post-order traversal, requires that every node in the traversed tree have the same structure, and requires that each traversal be compatible (i.e., that each traversal visit all the children of a node), and hence is not expressive enough to handle, e.g., arbitrary AST traversals. As a result of these limitations, their optimization framework also does not handle general code motion, and does not support partial fusion.

5.2 Fusion for functional programs

Traversal in functional programs are dense with intermediate structures; each tree traversal in a functional program results in an intermediate tree that is traversed by the following traversal. Hence "fusion" in the functional world is usually associated with eliminating such intermediate structures in addition to combining work from different traversals. Such fusion is usually also referred to as "Deforestation".

5.2.1 Deep fusion approaches

In 1977, Burstall and Darlington [10] provide a a calculation method to transform recursive equations so as to reach a fused program, however decisions for applying transformations are left to the programmer. More recent work [55] unified several previous fusion approaches under one theoretical and notational framework based on recursive coalgebras.

Wadler's deforestation is the most popular deep fusion approach [8], in this work expressions of functions written in treeless can be transformed using a recursive application of some rewrite rules into a treeless form. Different extensions to Walder's

approach came next trying to relax the treeless form requirement using a consumer-producer model [38, 56]. However, the approach did not make its way to mainstream compilers due to its syntactic restriction. Such syntactic restrictions are introduced to guarantee the termination of the transformation and to avoid an increase in runtime complexity.

Tupling Another "fusion" transformation that is more close to the fusion in the imperative word is **tupling** [39]. Tupling targets combining work from different traversals but not eliminating intermediate structures. It looks at functions that traverse the same structure and merge them into one function that returns the outputs of the merged functions in a tuple. While fusion looks at sequential invocations of traversals where each operates on the output of the previous one, tupling targets independent functions that traverse the same structure.

A previous study showed that fusion and tupling are correlated and applying each of them can lead to optimization opportunities of the other [37, 39]. A combined transformation was proposed where the two optimizations interleave, yet we do not know of any practical implementation and evaluation for the proposed optimization.

5.2.2 Shallow fusion

Another approach of fusion that is more popular in practical implementation, especially when optimizing functions operating on lists is *shortcut fusion* [14]. Such approaches operate on high-level constructs(map, fold, filter, build..etc) and are defined as rewrite rules.

In practice, functional programming languages have settled on using libraries of combinators with known fusion transformations. For example, in the Haskell ecosystem, many everyday libraries use the *stream fusion* approach [13] and variations on it. This combinator style works well for collections (arrays, lists, etc) but is sharply more limited than general recursive functions on trees. One example of this fact is that compiler writers cannot use such frameworks for fusing AST traversals.

5.2.3 Additional Related Works

Domain-specific languages [57, 58] and data-parallel libraries [59–61] typically include fusion rules that merge multiple data-parallel transformations of their data collections. For example, these systems frequently provide map and fold operations over multi-dimensional arrays (dense or sparse). These systems typically manipulate an explicit abstract syntax representation to perform fusion optimizations, and can generally be classified with the combinator-based approaches we discussed in section 4.3.

In contrast, libraries that expose *iterator* or *generator* abstractions can often achieve fusion by construction and avoid the necessity of fusion as a compiler optimization (which may not always succeed).

For example Rust (or C++) iterators[1] provide a stream of elements without necessarily storing them within a data structure; likewise a Rust (rayon) parallel map operation, simply returns a new parallel iterator without creating a new data structure. In functional contexts as well, libraries often provide data abstractions where a client can "pull" data, or where a producer pushes data to a series of downstream consumers (as in "push arrays" [62]. All these techniques amount to fusion-by-construction programming. However, in these approaches the programmer often needs to manually intervene if they *do* want to explicitly store a result in memory and share it between consumers.

[1] https://doc.rust-lang.org/book/ch13-02-iterators.html

REFERENCES

REFERENCES

[1] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, Jun. 1970.

[2] D. Sarkar, "Nanopass compiler infrastructure," Ph.D. dissertation, Indianapolis, IN, USA, 2008, aAI3337263.

[3] D. Petrashko, O. Lhoták, and M. Odersky, "Miniphases: Compilation using modular and efficient tree transformations," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 201–216. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062346

[4] S. Rajbhandari, J. Kim, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, R. J. Harrison, and P. Sadayappan, "On fusing recursive traversals of k-d trees," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: ACM, 2016, pp. 152–162. [Online]. Available: http://doi.acm.org/10.1145/2892208.2892228

[5] ——, "A domain-specific compiler for a parallel multiresolution adaptive numerical simulation environment," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 40:1–40:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014958

[6] L. A. Meyerovich and R. Bodik, "Fast and parallel webpage layout," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 711–720. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772763

[7] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodik, "Parallel schedule synthesis for attribute grammars," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 187–196. [Online]. Available: http://doi.acm.org/10.1145/2442516.2442535

[8] P. Wadler, "Deforestation: transforming programs to eliminate trees," *Theoretical Computer Science*, vol. 73, no. 2, pp. 231 – 248, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/030439759090147A

[9] W. Chin, Z. Hu, and M. Takeichi, "A modular derivation strategy via fusion and tupling," 12 1999.

[10] R. M. Burstall and J. Darlington, "A transformation system for developing recursive programs," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 44–67, 1977.

[11] A. Gill, J. Launchbury, and S. L. Peyton Jones, "A short cut to deforestation," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, ser. FPCA '93. New York, NY, USA: ACM, 1993, pp. 223–232. [Online]. Available: http://doi.acm.org/10.1145/165180.165214

[12] J. Svenningsson, "Shortcut fusion for accumulating parameters & zip-like functions," in *ICFP*, vol. 2, 2002, pp. 124–132.

[13] D. Coutts, R. Leshchinskiy, and D. Stewart, "Stream fusion: From lists to streams to nothing at all," in *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '07. New York, NY, USA: ACM, 2007, pp. 315–326.

[14] P. Johann, "A generalization of short-cut fusion and its correctness proof," *Higher-Order and Symbolic Computation*, vol. 15, no. 4, pp. 273–300, Dec 2002. [Online]. Available: https://doi.org/10.1023/A:1022982420888

[15] D. Coutts, "Stream fusion: Practical shortcut fusion for coinductive sequence types," 2011.

[16] L. Sakka, K. Sundararajah, and M. Kulkarni, "Treefuser: A framework for analyzing and fusing general recursive tree traversals," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 76:1–76:30, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133900

[17] L. Sakka, K. Sundararajah, R. R. Newton, and M. Kulkarni, "Sound, fine-grained traversal fusion for heterogeneous trees," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: ACM, 2019, pp. 830–844. [Online]. Available: http://doi.acm.org/10.1145/3314221.3314626

[18] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodik, "Parallel schedule synthesis for attribute grammars," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 187–196. [Online]. Available: http://doi.acm.org/10.1145/2442516.2442535

[19] 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

[20] R. Harrison, G. Beylkin, F. Bischoff, J. Calvin, G. Fann, J. Fosso-Tande, D. Galindo, J. Hammond, R. Hartman-Baker, J. Hill, J. Jia, J. Kottmann, M. Yvonne Ou, J. Pei, L. Ratcliff, M. Reuter, A. Richie-Halford, N. Romero, H. Sekino, W. Shelton, B. Sundahl, W. Thornton, E. Valeev, A. Vazquez-Mayagoitia, N. Vence, T. Yanai, and Y. Yokoi, "Madness: A multiresolution, adaptive numerical environment for scientific simulation," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S123–S142, 2016. [Online]. Available: https://doi.org/10.1137/15M1026171

[21] J. R. Larus and P. N. Hilfinger, "Detecting conflicts between structure accesses," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, ser. PLDI '88. New York, NY, USA: ACM, 1988, pp. 24–31. [Online]. Available: http://doi.acm.org/10.1145/53990.53993

[22] J. Hummel, L. J. Hendren, and A. Nicolau, "A general data dependence test for dynamic, pointer-based data structures," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94. New York, NY, USA: ACM, 1994, pp. 218–229. [Online]. Available: http://doi.acm.org/10.1145/178243.178262

[23] A. Cohen and J.-F. Collard, "Instance-wise reaching definition analysis for recursive programs using context-free transductions," in *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 332–. [Online]. Available: http://dl.acm.org/citation.cfm?id=522344.825716

[24] Y. Weijiang, S. Balakrishna, J. Liu, and M. Kulkarni, "Tree dependence analysis," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 314–325. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737972

[25] 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*, ser. POPL '96. New York, NY, USA: ACM, 1996, pp. 1–15. [Online]. Available: http://doi.acm.org/10.1145/237721.237724

[26] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '99. New York, NY, USA: ACM, 1999, pp. 105–118. [Online]. Available: http://doi.acm.org/10.1145/292540.292552

[27] 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

[28] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *J. Comput. Phys.*, vol. 73, no. 2, pp. 325–348, Dec. 1987. [Online]. Available: http://dx.doi.org/10.1016/0021-9991(87)90140-9

[29] V. Rokhlin, "Rapid solution of integral equations of classical potential theory," *Journal of Computational Physics*, vol. 60, no. 2, pp. 187 – 207, 1985. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0021999185900026

[30] N. Engheta, W. D. Murphy, V. Rokhlin, and M. S. Vassiliou, "The fast multipole method (fmm) for electromagnetic scattering problems," *IEEE Transactions on Antennas and Propagation*, vol. 40, no. 6, pp. 634–641, Jun 1992.

[31] N. Hegde, J. Liu, K. Sundararajah, and M. Kulkarni, "Treelogy: A benchmark suite for tree traversals," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 227–238.

[32] R. Farrow, K. Kennedy, and L. Zucconi, "Graph grammars and global program data flow analysis," in *Proceedings of the 17th Annual Symposium*

on Foundations of Computer Science, ser. SFCS '76. Washington, DC, USA: IEEE Computer Society, 1976, pp. 42–56. [Online]. Available: http://dx.doi.org/10.1109/SFCS.1976.17

[33] L. Sakka, K. Sundararajah, R. R. Newton, and M. Kulkarni, "Sound, fine-grained traversal fusion for heterogeneous trees," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062346

[34] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri, "Openfst: A general and efficient weighted finite-state transducer library," in *Implementation and Application of Automata*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 11–23.

[35] N. E. Fenton, *Software Metrics: A Rigorous Approach*. London, UK, UK: Chapman & Hall, Ltd., 1991.

[36] W.-N. Chin and Z. Hu, "Towards a modular program derivation via fusion and tupling," in *Generative Programming and Component Engineering*, D. Batory, C. Consel, and W. Taha, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 140–155.

[37] W. Chin, "Fusion and tupling transformations: Synergies and conflicts," in *Proceedings of the Fuji International Workshop on Functional and Logic Programming, Susono, Japan*. World Scientific Publishing, 1995, pp. 106–125.

[38] W.-N. Chin, "Safe fusion of functional expressions ii: Further improvements," *Journal of Functional Programming*, vol. 4, no. 4, p. 515555, 1994.

[39] ——, "Towards an automated tupling strategy," in *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, ser. PEPM '93. New York, NY, USA: ACM, 1993, pp. 119–132. [Online]. Available: http://doi.acm.org/10.1145/154630.154643

[40] ——, "Towards an automated tupling strategy," in *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, ser. PEPM '93. New York, NY, USA: ACM, 1993, pp. 119–132. [Online]. Available: http://doi.acm.org/10.1145/154630.154643

[41] M. Vollmer, S. Spall, B. Chamith, L. Sakka, C. Koparkar, M. Kulkarni, S. Tobin-Hochstadt, and R. R. Newton, "Compiling Tree Transforms to Operate on Packed Representations," in *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), P. Müller, Ed., vol. 74. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 26:1–26:29. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7273

[42] K. Kennedy and K. S. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. London, UK, UK: Springer-Verlag, 1994, pp. 301–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=645671.665526

[43] A. Darte, "On the complexity of loop fusion," in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 149–. [Online]. Available: http://dl.acm.org/citation.cfm?id=520793.825721

[44] A. Qasem and K. Kennedy, "Profitable loop fusion and tiling using model-driven empirical search," in *Proceedings of the 20th Annual International Conference on Supercomputing*, ser. ICS '06. New York, NY, USA: ACM, 2006, pp. 249–258. [Online]. Available: http://doi.acm.org/10.1145/1183401.1183437

[45] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 101–113. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375595

[46] P. Feautrier, "Some efficient solutions to the affine scheduling problem: I. one-dimensional time," *Int. J. Parallel Program.*, vol. 21, no. 5, pp. 313–348, Oct. 1992. [Online]. Available: http://dx.doi.org/10.1007/BF01407835

[47] ——, "Some efficient solutions to the affine scheduling problem. part ii. multidimensional time," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, Dec 1992. [Online]. Available: https://doi.org/10.1007/BF01379404

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

[49] J. Doner, "Tree acceptors and some of their applications," *J. Comput. Syst. Sci.*, vol. 4, no. 5, pp. 406–451, Oct. 1970. [Online]. Available: http://dx.doi.org/10.1016/S0022-0000(70)80041-1

[50] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi, "Tree automata techniques and applications," Available on: http://www.grappa.univ-lille3.fr/tata, 2007, release October, 12th 2007.

[51] J. Engelfriet and H. Vogler, "Macro tree transducers," *Journal of Computer and System Sciences*, vol. 31, no. 1, pp. 71 – 146, 1985. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022000085900662

[52] J. Engelfriet, "Bottom-up and top-down tree transformations— a comparison," *Mathematical systems theory*, vol. 9, no. 2, pp. 198–231, Jun 1975. [Online]. Available: https://doi.org/10.1007/BF01704020

[53] A. Maletti, "Compositions of extended top-down tree transducers," *Inf. Comput.*, vol. 206, no. 9-10, pp. 1187–1196, Sep. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.ic.2008.03.019

[54] J. Engelfriet and S. Maneth, "Output string languages of compositions of deterministic macro tree transducers," *J. Comput. Syst. Sci.*, vol. 64, no. 2, pp. 350–395, Mar. 2002. [Online]. Available: http://dx.doi.org/10.1006/jcss.2001.1816

[55] R. Hinze, T. Harper, and D. W. H. James, "Theory and practice of fusion," in *Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages*, ser. IFL'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 19–37. [Online]. Available: http://dl.acm.org/citation.cfm?id=2050135.2050137

[56] D. Van Arkel, J. Van Groningen, and S. Smetsers, "Fusion in practice," in *Proceedings of the 14th International Conference on Implementation of Functional Languages*, ser. IFL'02. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 51–67. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756972.1756976

[57] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier, "Optimising purely functional GPU programs," in *ICFP: International Conference on Functional Programming*. ACM, 2013, pp. 49–60.

[58] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '11. IEEE, 2011, pp. 89–100.

[59] B. Lippmeier, M. Chakravarty, G. Keller, and S. Peyton Jones, "Guiding parallel array fusion with indexed types," in *ACM SIGPLAN Notices*, vol. 47, no. 12. ACM, 2012, pp. 25–36.

[60] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: Easy, efficient data-parallel pipelines," in *Programming Language Design and Implementation*. New York, NY, USA: ACM, 2010, pp. 363–375. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806638

[61] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[62] B. J. Svensson and J. Svenningsson, "Defunctionalizing push arrays," in *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*. ACM, 2014, pp. 43–52.