Sound, Fine-Grained Traversal Fusion for Heterogeneous Trees

Laith Sakka
Kirshanthan Sundararajah
Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana, USA
{lsakka,ksundar}@purdue.edu

Ryan R. Newton
Computer Science
Indiana University
Bloomington, Indiana, USA
rrnewton@indiana.edu

Milind Kulkarni
Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana, USA
milind@purdue.edu

Abstract

Applications in many domains are based on a series of traversals of tree structures, and *fusing* these traversals together to reduce the total number of passes over the tree is a common, important optimization technique. In applications such as compilers and render trees, these trees are heterogeneous: different nodes of the tree have different types. Unfortunately, prior work for fusing traversals falls short in different ways: they do not handle heterogeneity; they require using domain-specific languages to express an application; they rely on the programmer to aver that fusing traversals is safe, without any soundness guarantee; or they can only perform coarse-grain fusion, leading to missed fusion opportunities. This paper addresses these shortcomings to build a framework for fusing traversals of heterogeneous trees that is automatic, sound, and fine-grained. We show across several case studies that our approach is able to allow programmers to write simple, intuitive traversals, and then automatically fuse them to substantially improve performance.

CCS Concepts • **Software and its engineering → Recursion**; **Compilers**; *Software performance.*

Keywords Fusion, Tree traversals, Locality

ACM Reference Format:
Laith Sakka, Kirshanthan Sundararajah, Ryan R. Newton, and Milind Kulkarni. 2019. Sound, Fine-Grained Traversal Fusion for Heterogeneous Trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314626

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314626

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 example, a compiler is written as dozens of individual AST-rewriting passes [18, 25]—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 [18].

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 [16–18, 20–22]. In the world of functional programs, *deforestation* techniques rewrite data structure operations to avoid materializing intermediate data structures, either through syntactic rewrites [5, 27] or through the use of special combinators that promote fusion [7]. For web browsers, render-tree passes can be expressed in high-level, attribute grammar–like languages [16] and then passed to a compiler that generates fused passes [17]. For solving differential and integral equations, computations on spatial functions can be expressed using high-level numerical operators [21] that are then fused together into combined kd-tree passes by domain-specific compilers [20, 21]. 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 [18].

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 [22]. 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 [20, 21], or Petrashko et al. who require very specific traversal structures in miniphases [18]). 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.

1.1 Contributions

This paper 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 paper 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 1. Grafter in comparison to prior work. Note that Grafter provides finer-grained fusion than TreeFuser. We exclude syntactic rewrites of functional programs (e.g., [27]), as their rewrites are not directly analogous to fusion.

Approach	Heterogeneous trees	Fine-grained fusion	General expressivity	Dependence analysis
Stream fusion [7]	✓	✗	✗	NA
Attribute grammars [17]	✓	✗	✗	✓
Miniphases [18]	✓	✗	✗	✗
Rajbhandari et al. [20]	✗	✗	✗	✗
TreeFuser [22]	✗	✓	✓	✓
Grafter	✓	✓	✓	✓

Table 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.

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.

1.2 Outline

The remainder of the paper is organized as follows. Section 2 provides an overview of our fusion framework Grafter. Section 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 4 details the prototype implementation of Grafter. Section 5 evaluates Grafter across several case studies. Section 6 discusses related work and Section 7 concludes.

2 Grafter Overview

Grafter adopts a strategy for fusing traversals where a programmer writes individual tree traversals in a standard, C++-like language (Section 3.1), as functions 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 2, but for illustration purposes, we use a simple example shown in Figure 1a, 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 1a into one call to function f_{12} that executes the two functions back-to-back in their original order as shown in Figure 1b (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 [22].) In Figure 1b, 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 3.2 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 1c shows the results of such re-ordering.

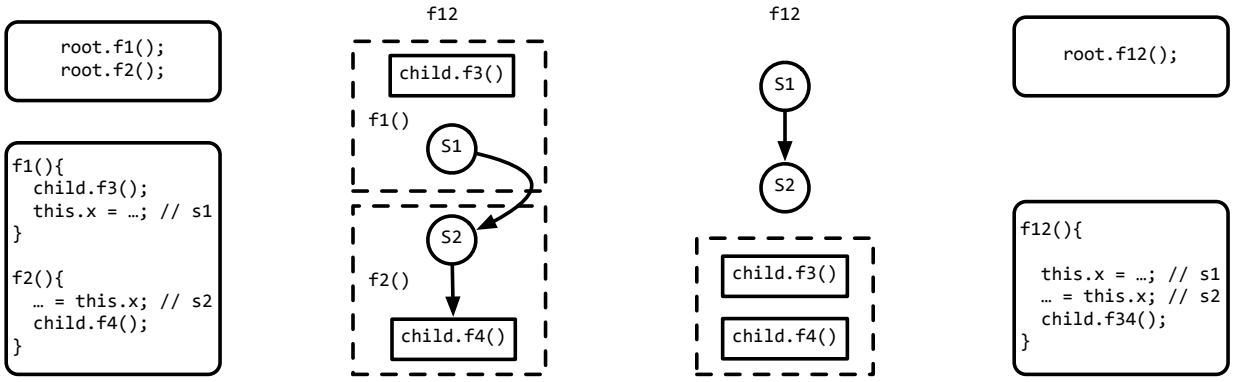
Note that in this example the reordering step changes the traversal f_1 from a post-order traversal to a pre-order traversal ($s1$ is executed at parents before their children in f_{12}, while it is executed at the children before their parents in the original function $f1$). In other words this reordering 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 1c 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 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.



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

Figure 1. Fusing mutual traversals in Grafter.

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

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 3.2). Then we explain how Grafter uses the dependence representation to synthesize new, fused functions (Section 3.3). But first, we explain the language that Grafter uses to express its traversals.

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 3a 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* [1]. In order to support tree children with abstract types, a tree in Grafter can inherit fields and virtual traversal methods from other tree types and can specialize inherited virtual traversals by overriding them.

Figure 3b 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[2]. 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[3].

Grafter uses `new` and `delete` C++ language constructs to support leaf mutations (constructs 8 and 9 in Figure 3b). 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

[1] 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.

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

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

```
int CHAR_WIDTH;

class Element {
  Element *Next;
  int Height = 0, Width = 0;
  int MaxHeight = 0, TotalWidth = 0;
  virtual void computeWidth(){};
  virtual void computeHeight(){};
};

class TextBox: public Element {
  String Text;
  void computeWidth(){
    Next->computeWidth();
    Width = Text.Length;
    TotalWidth = Next->Width + Width;
  };
  void computeHeight(){
    Next->computeHeight();
    Height = Text.Length * (Width/CHAR_WIDTH) +
        1;
    MaxHeight = Height;
    if(Next->Height > Height)
      MaxHeight = Next->Height;
  };
};

class Group: public Element {
  Element *Content;
  BorderInfo Border;
  void computeWidth(){
    Content->computeWidth();
    Next->computeWidth();
    Width = Content->Width + Border.Size*2;
    TotalWidth = Width + Next->Width;
  };
  void computeHeight(){
    Content->computeHeight();
    Next->computeHeight();
    Height = Content->MaxHeigh + Border.Size*2
    MaxHeight = Height;
    if(Next->Height > Height )
      MaxHeight = Next->Heigh;
  };
};

class End: public Element {
};

int main(){
  Element *ElementsList = ...;
  ElementsList->computeWidth();
  ElementsList->computeHeight();
}
```

Figure 2. An example of a program written in Grafter.

as read-only functions. Equations 15, and 11 shows the usage of such functions.

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.

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

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

$l \in$ <alias-ref> := $l_1 \mid l_2 \mid \ldots$ $p \in$ <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>)*]**)** **{**
<stmt>+ **}** (4)

<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.

<expr> := <const> | <data-access> | <bin-expr>| <pure-call> (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.

Figure 3. Language of Grafter.

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 3c). 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 2 shows an example of a program written in Grafter (we elide the annotations for brevity). This program consists of a tree of `Elements` that can be `TextBoxes` or `Groups` of `TextBoxes` (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.

3.2 Dependence Graphs and Access Representations

The primary representation that Grafter uses to drive its fusion process is the dependence graph [22]. As described in Section 2, this graph has one vertex for each *top level* statement[4] and edges between statements if there are dependences between them. More precisely, an edge exists between two vertices v_1 and v_2, arising 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?

3.2.1 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 [22, 28] 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 [13, 29]). 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 [22]).

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 [13], 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.

[4] In other words, one vertex for each <stmt> construct, as shown in Figure 3b.

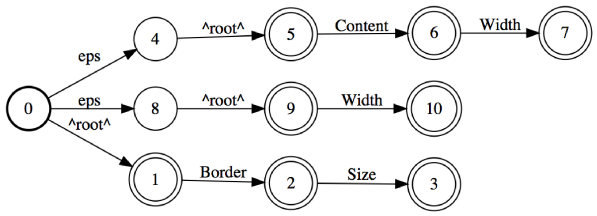


Figure 4. Automata that represents summary of read accesses for the simple statement. *eps* is epsilon transition, and *root* is the traversed node transition.

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.

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 4 shows the tree read automaton for the statement:

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

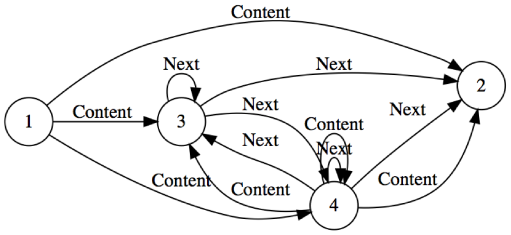
Finding dependences between statements These automata provide the information needed to find dependences between statements. Because each statement's automata captures the full set of access paths read (or written) for a statement, and we are interested in whether the statements have a dependence *when 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.

Building access automata for traversing calls

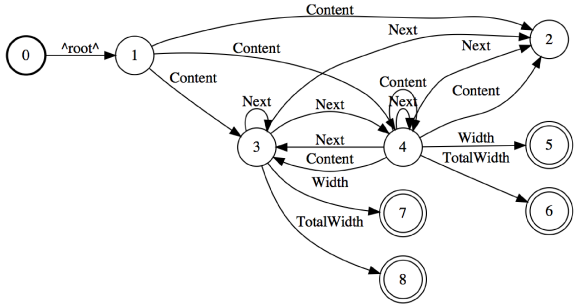
Representing accesses of traversal calls is not as simple. For a given call statement we want to construct a finite automaton that captures *any possible access path that could arise during the call relative to the tree node being traversed by caller*—including the fact that a call may invoke more traversals.

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.

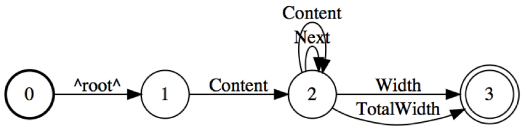
The situation is more complicated for *on-tree* accesses, as those are parameterized by the receiver of the call, `this` (i.e., the node that is being traversed). To find the accessed locations relative to `this`, we need to find two things: the functions that are reachable during the call (to know which



(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.

Figure 5. Construction of tree writes automata for the traversing statement *content->computeWidth()*. *root* is the traversed node transition.

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.

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 5a shows the call graph that is generated for `Content->computeWidth()`

```
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.

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 5b shows the resulting automaton, and Figure 5c 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.

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 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 `ElementsList->computeWidth();` followed by `ElementsList->computeHeight()` in Figure 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 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 [22, Section 7].

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 [22], 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 6 shows the result of fusing the two functions that

```
1
2  ...
3  void _fuse__F3F4(TextBox *_r, int active_flags) {
4    TextBox *_r_f0 = (TextBox *)(_r);
5    TextBox *_r_f1 = (TextBox *)(_r);
6    if (active_flags & 0b11) /*call*/ {
7      unsigned int call_flags = 0;
8      call_flags <<= 1;
9      call_flags |= (0b01 & (active_flags >> 1));
10     call_flags <<= 1;
11     call_flags |= (0b01 & (active_flags >> 0));
12     _r_f0->Next->__switch1(call_flags);
13   }
14   if (active_flags & 0b1) {
15     _r_f0->Width = _r_f0->Text.Length;
16     _r_f0->TotalWidth = _r_f0->Next->Width + _r_f0
           ->Width;
17   }
18   if (active_flags & 0b10) {
19     _r_f1->Height = _r_f1->Text.Length * (_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 }
```

Figure 6. Output of Grafter.

computes the width and the height for `TextBox` element in the program shown in figure 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. 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 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 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 [22]. Appropriate flags should be passed when a fused call is invoked within a traversal. The `call_flags` variable, defined at line 7 in Figure 6, holds the active flags that are passed to the next traversing call. Lines 7–11 fill in the appropriate flags from the `active_flags` in the `call_flags` based on which traversals the outlined calls belong to.

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.

4 Implementation

Grafter is implemented as a Clang tool that performs source to source transformation for input programs [5]. 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 [1] 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 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. 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.

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.

This paper presents results for the first two case studies, which exploit the novel features of Grafter. The latter two case studies are Grafter versions of benchmarks from Rajbhandari et al. [20, 21] and TreeFuser [23], respectively; the results are in the extended version of the paper [24].

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. [22]). The final effectiveness of fusion is hence determined by runtime.

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

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

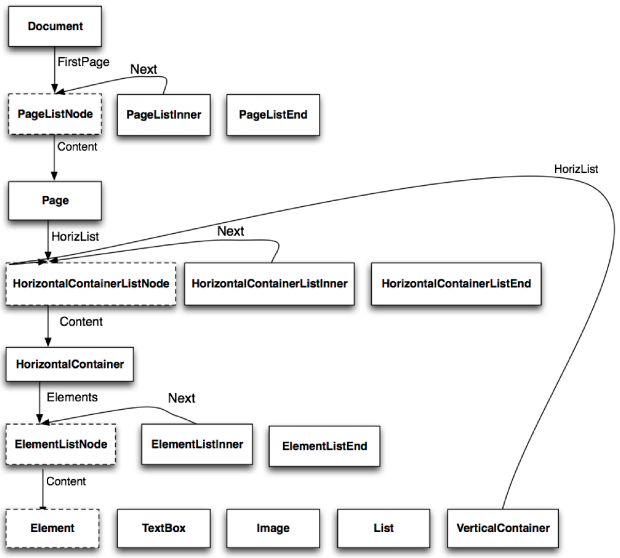


Figure 7. Class hierarchy of the render tree.

pages composed using nested horizontal and vertical containers with leaf elements (TextBox, Image, etc.)[6].

Figure 7 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`.

Five rendering passes are implemented and listed in Table 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 [22], 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 [22]. 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.

Figure 8a evaluates the Grafter-fused implementation of the render-tree case study, normalized to the unfused

[6]The extended version of the paper presents results for other types of input documents [24].

Table 2. Render-tree and AST passes.

Render-tree traversals	AST traversals
Resolve flex widths	De-sugar increment
Resolve relative Widths	De-sugar decrement
Set font style	Constant propagation
Compute height	Replace variable references
Compute positions	Constant folding
	Remove unused branches

Grafter baseline, while Figure 8b evaluates the TreeFuser-fused implementation, normalized to *its* unfused baseline[7].

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 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%.

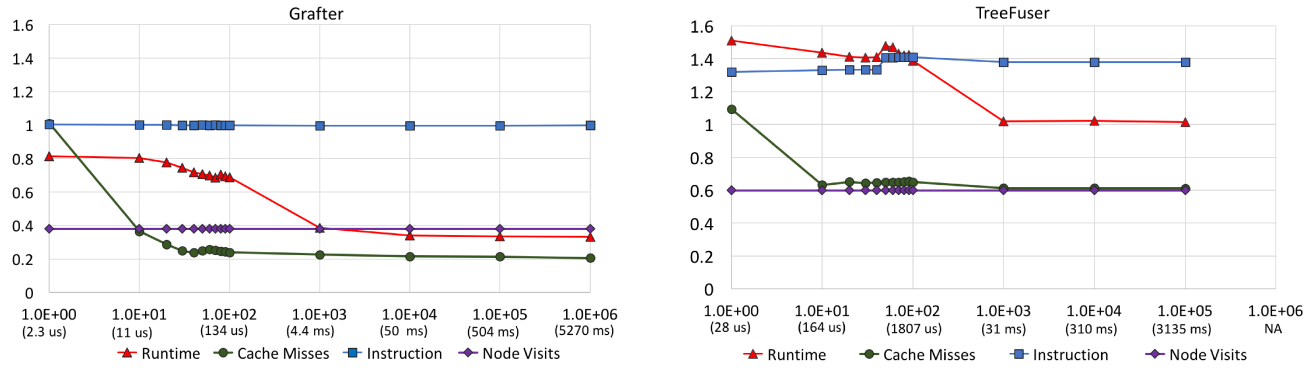
We also see the advantage of Grafter's type-specific fusion approach: because fused functions are on a per-type basis, Grafter is able to leverage dynamic dispatch to specialize traversals. As a result, it exhibits virtually no instruction overhead relative to the baseline. TreeFuser, in contrast, has 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 8a and 8b.

Programmability-wise, the overall logical lines of code (LLOC) [8] 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.

[7]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 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.

[8]We use Fenton's definition of LLOC for C++: the number of instructions with the semantic delimiter [11].



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

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

Figure 8. Performance comparison of render-tree passes written in GRAFTER and TreeFuser for different tree sizes. Number in parentheses is runtime of the baseline. Number of pages on the x axis and normalized measurement on the y axis.

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. Figure 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 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[9]. Figure 10 shows the performance of the fused AST traversals with respect to the un-fused ones [10]. GRAFTER reduces L2 cache misses by 75% and once the tree is big enough,

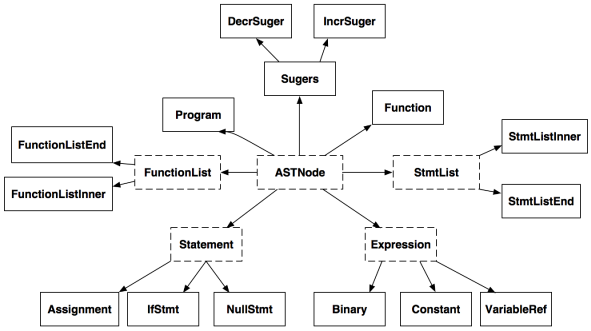


Figure 9. AST class hierarchy with 20 different types.

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.

6 Related Work

The overall goal of this work is to enable programmers to achieve efficient tree traversals. Thus it is obliquely related to prior work on that improves the performance of a *single* tree traversal by adjusting memory layout or encoding [3, 4, 14, 26]. The most directly related work, however, optimizes series of traversals, typically by fusing together multiple

[9] The extended version of the paper presents additional results for different input configurations [24].

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

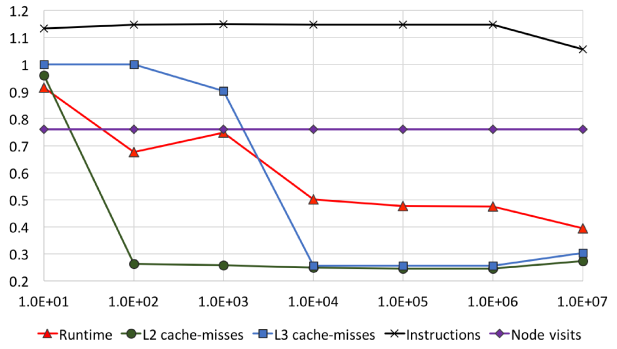


Figure 10. Performance measurements for fused AST traversals normalized to the unfused ones for different tree sizes. Number of functions on the x axis, and normalized measurements on the y axis.

passes. This work can be sub-divided based on the *source representation* it targets:

Specialized or domain-specific programs In these systems, the programmer aids the tool by expressing their tree traversals in a structured form. For example, in *miniphases* [18], 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* [25].

Another specialized way to express tree traversals is by using attribute grammars to specify *tree transducers*—automata that traverse trees and create output [6, 9]. Using this specialized representation of programs, subsequent authors were able to improve efficiency of compositions of tree transducers [10, 15]. 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.

Enabling fusion based on a high-level representation of traversals has likewise been explored in the HPC domain: for instance the MADNESS simulation framework uses this approach for fusing kd-tree passes Rajbhandari et al. [20, 21]. This abstraction supports a more restrictive notion of fusion than Grafter. All traversals must visit the same children and must visit them in either a pre-order or post-order; partial fusion is not possible. Moreover, the framework relies on programmer-provided dependence information.

Functional Programs As discussed in the introduction, the prior research explores fusing general recursive functional programs and eliminating intermediate data structures (deforestation) [27]. In practice, functional programming languages have settled on using libraries of combinators with known fusion transformations (like `map` and `fold`). For example, in the Haskell ecosystem, many everyday libraries use the *stream fusion* approach [7] 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.

Imperative Programs The most closely related works to Grafter are systems that target imperative or object-oriented source programs. Aside from TreeFuser (which we have already described) most work on fusion in imperative programs focuses on *loop fusion*, merging the bodies of adjacent loops. Typically this is applied to programs operating on arrays and matrices [8, 12, 19]. The most well known work in this area targets the further restricted class of programs with *affine* indexing expressions [2]. These approaches do not generalize to recursive programs operating over irregular data such as the tree traversals we explore in this paper.

7 Conclusions

This paper 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.

Acknowledgments

This work was supported in part by National Science Foundation awards CCF-1725672 and CCF-1725679, and by Department of Energy award DE-SC0010295. We would like to thank our shepherd, Ben Titzer, as well as the anonymous reviewers for their suggestions and comments.

References

[1] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. OpenFst: A General and Efficient Weighted Finite-State Transducer Library. In *Implementation and Application of Automata*. Springer Berlin Heidelberg, Berlin, Heidelberg, 11–23.

[2] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[3] TM Chilimbi, MD Hill, and JR Larus. 1999. Cache-conscious structure layout. *ACM SIGPLAN Notices* (1999). http://dl.acm.org/citation.cfm?id=301633

[4] Trishul M. Chilimbi and James R. Larus. 1999. Using generational garbage collection to implement cache-conscious data placement. , 37–48 pages. https://doi.org/10.1145/301589.286865

[5] Wei-ngan Chin, Zhenjiang Hu, and Masato Takeichi. 1999. A Modular Derivation Strategy via Fusion and Tupling. (12 1999).

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

[7] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. ACM, New York, NY, USA, 315–326. https://doi.org/10.1145/1291151.1291199

[8] Alain Darte. 1999. On the Complexity of Loop Fusion. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*. IEEE Computer Society, Washington, DC, USA, 149–. http://dl.acm.org/citation.cfm?id=520793.825721

[9] John Doner. 1970. Tree Acceptors and Some of Their Applications. *J. Comput. Syst. Sci.* 4, 5 (Oct. 1970), 406–451. https://doi.org/10.1016/S0022-0000(70)80041-1

[10] Joost Engelfriet and Sebastian Maneth. 2002. Output String Languages of Compositions of Deterministic Macro Tree Transducers. *J. Comput. Syst. Sci.* 64, 2 (March 2002), 350–395. https://doi.org/10.1006/jcss.2001.1816

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

[12] Ken Kennedy and Kathryn S. McKinley. 1994. 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*. Springer-Verlag, London, UK, UK, 301–320. http://dl.acm.org/citation.cfm?id=645671.665526

[13] J. R. Larus and P. N. Hilfinger. 1988. Detecting Conflicts Between Structure Accesses. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. ACM, New York, NY, USA, 24–31. https://doi.org/10.1145/53990.53993

[14] Chris Lattner and Vikram Adve. 2005. Automatic pool allocation: improving performance by controlling data structure layout in the heap. *ACM SIGPLAN Notices* 40 (2005), 129–142. https://doi.org/10.1145/1065010.1065027

[15] Andreas Maletti. 2008. Compositions of Extended Top-down Tree Transducers. *Inf. Comput.* 206, 9-10 (Sept. 2008), 1187–1196. https://doi.org/10.1016/j.ic.2008.03.019

[16] Leo A. Meyerovich and Rastislav Bodik. 2010. Fast and Parallel Webpage Layout. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. ACM, New York, NY, USA, 711–720. https://doi.org/10.1145/1772690.1772763

[17] Leo A. Meyerovich, Matthew E. Torok, Eric Atkinson, and Rastislav Bodik. 2013. Parallel Schedule Synthesis for Attribute Grammars. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/2442516.2442535

[18] Dmitry Petrashko, Ondřej Lhoták, and Martin Odersky. 2017. Miniphases: Compilation Using Modular and Efficient Tree Transformations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 201–216. https://doi.org/10.1145/3062341.3062346

[19] Apan Qasem and Ken Kennedy. 2006. Profitable Loop Fusion and Tiling Using Model-driven Empirical Search. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS '06)*. ACM, New York, NY, USA, 249–258. https://doi.org/10.1145/1183401.1183437

[20] Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noel Pouchet, Fabrice Rastello, Robert J. Harrison, and P. Sadayappan. 2016. 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 (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 40, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014958

[21] Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, Robert J. Harrison, and P. Sadayappan. 2016. On Fusing Recursive Traversals of K-d Trees. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, New York, NY, USA, 152–162. https://doi.org/10.1145/2892208.2892228

[22] Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. 2017. TreeFuser: A Framework for Analyzing and Fusing General Recursive Tree Traversals. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 76 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133900

[23] Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. 2017. TreeFuser: A Framework for Analyzing and Fusing General Recursive Tree Traversals. In *PACM Progr. Lang. 1 (OOPSLA '17)*. ACM, 31. https://doi.org/10.1145/3133900

[24] Laith Sakka, Kirshanthan Sundararajah, Ryan R. Newton, and Milind Kulkarni. 2019. Sound, Fine-Grained Traversal Fusion for Heterogeneous Trees - Extended Version. arXiv:arXiv:1904.07061

[25] Dipanwita Sarkar. 2008. *Nanopass Compiler Infrastructure*. Ph.D. Dissertation. Indianapolis, IN, USA. Advisor(s) Dybvig, R. Kent. AAI3337263.

[26] D. N. Truong, F. Bodin, and A. Seznec. 1998. Improving Cache Behavior of Dynamically Allocated Data Structures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*. IEEE Computer Society, Washington, DC, USA, 322–. http://portal.acm.org/citation.cfm?id=522344.825680

[27] Philip Wadler. 1990. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science* 73, 2 (1990), 231 – 248. https://doi.org/10.1016/0304-3975(90)90147-A

[28] Yusheng Weijiang, Shruthi Balakrishna, Jianqiao Liu, and Milind Kulkarni. 2015. Tree Dependence Analysis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 314–325. https://doi.org/10.1145/2737924.2737972

[29] Ben Wiedermann and William R. Cook. 2007. Extracting Queries by Static Analysis of Transparent Persistence. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/1190216.1190248