Reasoning About Recursive Tree Traversals

Yanjun Wang
Purdue University
West Lafayette, Indiana, USA
wang3204@purdue.edu

Dalin Zhang
Beijing Jiaotong University
Beijing, China
dalin@bjtu.edu.cn

Jinwei Liu
Beijing Jiaotong University
Beijing, China
12251187@bjtu.edu.cn

Xiaokang Qiu
Purdue University
West Lafayette, Indiana, USA
xkqiu@purdue.edu

Abstract
Traversals are commonly seen in tree data structures, and performance-enhancing transformations between tree traversals are critical for many applications. Existing approaches to reasoning about tree traversals and their transformations are ad hoc, with various limitations on the classes of traversals they can handle, the granularity of dependence analysis, and the types of possible transformations. We propose Retreet, a framework in which one can describe general recursive tree traversals, precisely represent iterations, schedules and dependences, and automatically check data-race-freeness and transformation correctness. The crux of the framework is a stack-based representation for iterations and an encoding to Monadic Second-Order (MSO) logic over trees. Experiments show that Retreet can automatically verify optimizations for complex traversals on real-world data structures, such as CSS and cyctrees, which are not possible before. Our framework is also integrated with other MSO-based analysis techniques to verify even more challenging program transformations.

CCS Concepts:
• Theory of computation → Abstraction; Program verification; Automated reasoning; • Software and its engineering → Compilers;Recursion; • General and reference → Verification.

Keywords: tree traversals, iterations, program equivalence, monadic second-order logic

ACM Reference Format:

1 We call it an iteration because it is akin to a loop iteration in a loop.

1 Introduction
Trees are one of the most widely used data structures in computer programming and data representations. Traversal is a common means of manipulating tree data structures for various systems, as diverse as syntax trees for compilers [19], k-d trees for scientific simulation [11, 12, 21, 22], and DOM trees for web browsers [16]. Due to dependence and locality reasons, these traversals may iterate over the tree in many different orders: pre-order, post-order, in-order or more complex, and in parallel for disjoint regions of the tree. A tree traversal can be regarded as a sequence of iterations (each executing a code block on a tree node) and many transformations essentially tweak the order of iterations for better performance or code quality, with the hope that no dependence between iterations is violated.

Matching this wide variety of applications, orders, and transformations, there has been a fragmentation of mechanisms that represent and analyze tree traversal programs, each making different assumptions and tackling a different class of traversals and transformations, using a different formalism. For instance, Meyerovich and Bodik [15] and Meyerovich et al. [16] use attribute grammars to represent webpage rendering passes and automatically compose/parallelize them, but the traversals representable and fusible are limited, as the dependence analysis is coarse-grained at the attribute level. TreeFuser [26] uses a general imperative language to represent traversals, but the dependence graph it can build is similarly coarse-grained. In contrast, the recently developed PolyRec [27] framework supports precise instance-wise analyses for tree traversals, but the underlying transducer representation limits the traversals they can handle to a class called perfectly nested programs, which excludes mutual recursion and tree mutation. All these mechanisms are ad hoc and incompatible, making it impossible to represent more complex traversals or combine heterogeneous code transformations. For instance, a simple, mutually recursive tree traversal is already beyond the scope of all existing approaches (see our running example later).
Therefore, toward automated reasoning about tree traversals arising from emerging computing applications, we believe that there are two fundamental research questions. First, how to generally represent tree traversals and analyze the dependences between iterations? An expressive language in which one can freely write and combine complex tree traversals is a precursor of handling many new applications. Second, how to automatically verify the validity of subtle transformations between tree traversals? From the perspective of static analyses, the key challenge is how to design an appropriate abstraction of the program such that it is as precise as possible yet amenable for automated reasoning. Our answers to these questions are RETREET, a general framework (as illustrated in Figure 1) in which one can write almost arbitrary tree traversals, reason about dependences between iterations of fine granularity, and check correctness of transformations automatically. This framework features an abstract yet detailed characterization of iterations, schedules and dependences, which we call Configuration, as well as a powerful reasoning algorithm.

In this paper, we first present RETREET ("REcursive TREE Traversal") as an expressive intermediate language that allows the user to flexibly describe tree traversals in a recursive fashion (Section 2). Remarkably, RETREET can express mutually recursive traversals, which cannot be handled by existing techniques. Second, we propose Configuration as a detailed, stack-based abstraction for dynamic instances in a traversal (Section 3). Intuitively, a configuration profiles the call stack maintained during the execution; it preserves the full computation history except for function calls, i.e., recursive calls become abstract and may return arbitrary values. Furthermore, this abstraction can be encoded to Monadic Second-Order (MSO) logic over trees, which allows us to reason about dependences and check data-race-freeness and equivalence of RETREET programs (Section 4). The encoded formulae can be checked using MSO-based decision procedures such as the one implemented in MONA [7]. Our framework is sound and incomplete. In other words, all verified programs are indeed data-race-free/equivalent, but there is no guarantee that all data-race-free/equivalent programs can be verified. Therefore, finally, we show our framework is practically useful by synthesizing or verifying provably-correct optimizations for four different classes of programs, including real-world applications such as CSS minification and Cyctletree routing, for the first time. One of these case studies also shows how RETREET is integrated with other MSO-based analysis techniques to verify list-traversal transformations that cannot be handled by RETREET alone (Section 5).

2 A Tree Traversal Language

In this section, we present RETREET, our imperative, general tree traversal language. While RETREET is syntactically simple and not intended to serve as an end-user programming language, we envision RETREET as an intermediate language for automatic analyses and many language features commonly used in practice should be translated to RETREET through a preprocessor. See more discussion in Section 2.1.

RETREET programs execute on a tree-shaped heap which consists of a set of locations. Each location, also called node, is the root of a (sub)tree and associated with a set dir of pointer fields and a set f of local fields. Pointer fields dir contain the references to the children of the original location; local fields f store the local Int values.

The syntax of RETREET is shown in Figure 2. A program consists of a set of functions; each has a single Loc parameter and optionally, a vector of Int parameters. We assume every program has a Main function as the entry point of the program. The body of a function comprises Blocks of code combined using conditionals, sequentials and parallelizations.

A block of code is either a function call or a straight-line sequence of assignments. A function call takes as input a LExpr which can be the current Loc parameter or any of its descendants, and a sequence of AExprs of length as expected. Each AExpr is an integer expression combining Int parameters and local fields of the Loc parameter. Non-call assignments compute values of AExprs and assign them to Int parameters, fields or special return variables. Note that the functions in RETREET can be mutually recursive, i.e., two or more functions call each other. However, there is a special syntactic restriction: every function g(n, 0) should not call, directly or indirectly through inlining, itself, i.e., g(n, . . .) with arbitrary Int arguments (see more discussion below).

The semantics of RETREET is common as expected and we omit the formal definition. In particular, all function parameters are call-by-value; the parallel execution adopts the statement-level interleaving semantics (every execution is a serialized interleaving of atomic statements).

Example 2.1. Figure 3 illustrates our running example, which is a pair of mutually recursive tree traversals. Odd(n) and Even(n) count the number of nodes at the odd and even layers of the tree n, respectively (n is at layer 1, n.i is at layer 2, and so forth). Odd and Even recursively call each other; and the Main function runs Odd and Even in parallel, and returns the two computed numbers. Note that the mutual recursion is beyond the capability of all existing automatic frameworks that handle tree traversals [1, 15, 16, 26, 27, 32].

2.1 Discussion of the Language Design

We remark about some critical design features of RETREET. Served as an intermediate language for analyses, RETREET is semantically expressive but syntactically simple. In a nutshell, RETREET has been carefully designed to be maximally
Figure 1. RETREET reasoning framework

```
dir ∈ Loc Fields  v ∈ Int Vars  n ∈ Loc Vars
f ∈ Int Fields     g : Function IDs
LExpr ::= n | LExpr.dir
AExpr ::= 0 | 1 | n.f | v | AExpr + AExpr | AExpr = AExpr
BExpr ::= LExpr == nil | true | AExpr > 0 | 1 | BExpr
         | BExpr && BExpr
Assgn ::= n.f = AExpr | v = AExpr
Block ::= v = g(LExpr, AExpr) | Assgn+  
Stmt ::= Block | if (BExpr) Stmt else Stmt | Stmt ; Stmt
Func ::= g(n, v) { Stmt }
Prog ::= Func+
```

Figure 2. Syntax of RETREET

Odd(n)
if (n == nil) return 0
else return Even(n.l) + Even(n.r) + 1

Even(n)
if (n == nil) return 0
else return Odd(n.l) + Odd(n.r)

Main(n)
{ o = Odd(n) || e = Even(n) }
return (o, e)

Figure 3. Mutually recursive traversals (original)

permissible of describing tree traversals, yet encodable to the MSO logic.

Key Language Restrictions. Three major design features make possible our MSO encoding presented in Section 4:

obviously terminating, single node traversal and no-tree-mutation. Despite these restrictions, RETREET is still more general and more expressive than the state of the art—to the best of our knowledge, all the restrictions we discuss below can be seen in all existing approaches (find more discussion in Section 6).

Termination: RETREET allows obviously terminating tree traversals. Any function \( g(n, v) \) should not contain recursive calls to \( g(n, \ldots) \), regardless of directly in \( Stmt \) or indirectly through inlining arbitrarily many calls in \( Stmt \). The restriction guarantees not only the termination, but also a bound of the steps of executions. With this restriction, every function call makes progress toward traversing the tree downward. Hence, the height of the call stack will be bounded by the height of the tree, and every statement \(^2\) is executed on a node at most once. Therefore, running a RETREET program \( P \) on a tree \( T \) will terminate in \( O(|P|h(T)) \) steps where \( h(T) \) is the height of the tree. This bound is critical as it allows us to encode the program execution to a tree model, with only a fixed amount of information on each node. In contrast, RETREET excludes the following program: \( A(n, k) \): if \( (k <= 0) \) return 0; else return \( A(n, k-1) + \ldots \). The program terminates, but the length of execution on node \( n \) is determined by the input value \( k \), which can be arbitrarily large and makes our tree-based encoding impossible.

Single node traversal: In RETREET, all functions take only one Loc parameter. Intuitively, this means the tree

\(^2\) Notice that two different call sites of the same function are considered two different statements. So the number of statements is bounded by the size of the program.
traversal is not allowed to manipulate more than one node at one time. For example, traversing two trees at the same time to find the max height is not allowed in RETREET. This is a nontrivial restriction and necessary for our MSO encoding. The insight of this restriction will be clearer in Section 4.

No tree mutation: Mutation to the tree topology is generally disallowed in RETREET. General tree mutations will possibly affect the tree-ness of the topology, where our tree-based encoding cannot fit in.

Other Restrictions for Simplification. As an intermediate language, RETREET has more syntactic restrictions, which are not fundamental and does not jeopardize the expressivity. In particular, RETREET does not support loops, global variables, return statements or integer arguments. These restrictions are not essential because loops or global variables can be rewritten to recursion and local variables, respectively. Return values or integer arguments can also be rewritten to local fields. As long as the rewritten program satisfies the real restrictions we set forth above, it can be handled by our framework. See our discussions below.

Loop-freeness: The RETREET language does not allow iterative loops. Recall that RETREET is meant to describe tree traversals, and the no-loop guarantee guarantees that the program manipulates every node only a bounded number of times, and hence the termination of the program. That said, most typical loops or even nested loops traversing a tree only compute a limited number of steps on each node, and hence can be naturally converted to recursive functions in RETREET.

No global variables: We omit global variables in RETREET. However, it is not difficult to extend for global variables. Note that when the program is sequential, i.e., no concurrency, one can simply replace a global variable with an extra parameter for every function, which copies in and copies out the value of the global variable. In the presence of concurrency, we need to refine the current syntax to reason about the schedule of manipulations to global variables. Basically, every statement accessing a global variable forms a separate Block, so that we can compare the order between any two global variable operations.

No return statements and no integer arguments: We handle recursive calls with return values with the following preprocessing. For every function, we introduce a special local field with the function name (if no conflict occurs) in each node to store the return value of the function call. Each return statement can be rewritten to a writing to the special local field in the callee (the unique Loc argument of the call). For each recursive call to a function in the program, we ignore the return value from the call and instead read from the corresponding special local field of the callee. Recursive calls with integer arguments are handled with similar preprocessing: the caller writes to special local fields of the callee such that the callee can read them as integer arguments. After this preprocessing step, the running example shown in Figure 3 are rewritten to the one in Figure 4.

As an intermediate language, RETREET has more syntactic restrictions, which are not fundamental and does not jeopardize the expressivity. In particular, RETREET does not support loops, global variables, return statements or integer arguments. These restrictions are not essential because loops or global variables can be rewritten to recursion and local variables, respectively. Return values or integer arguments can also be rewritten to local fields. As long as the rewritten program satisfies the real restrictions we set forth above, it can be handled by our framework. See our discussions below.

Figure 4. Mutually recursive traversals (no-return-value)

Odd(n)
if (n == nil) // c0
  n.Odd = 0 // s0
else
  Even(n.l) // s1
  Even(n.r) // s2
  n.Odd = n.l.Odd + n.r.Even + 1 // s3
Even(n)
if (n == nil) // c1
  n.Even = 0 // s4
else
  Odd(n.l) // s5
  Odd(n.r) // s6
  n.Even = n.l.Odd + n.r.Odd // s7
Main(n) // s8
{ Odd(n) || Even(n) } // s9
n.Main = (n.Odd, n.Even) // s10
their positions in the syntax tree. In particular, when two blocks \( s \sim t \) belong to the same function \( f \), there are three possible relations, determined by the least common ancestor (LCA) node of \( s \) and \( t \) that is a sequential, conditional or parallel.

**Example 2.2.** In our running example (Figure 4), there are 11 blocks. We number the blocks with \( s_0 \) through \( s_{10} \), as shown in the comment following each block. There are six call blocks: \( \text{AllCalls} = \{ s_1, s_2, s_5, s_6, s_8, s_9 \} \); and five non-call blocks: \( \text{AllNonCalls} = \{ s_0, s_3, s_4, s_7, s_{10} \} \). Take \( s_6 \) for example. \( \text{Path}(s_6) \) is just the path from the beginning of function \( \text{Even} \) (which \( s_6 \) belongs to) to \( s_6 \), i.e., from \( \sim c_1 \) to \( s_5 \) then \( s_6 \). The relation holds between any two blocks from the same group: \( s_0 \) through \( s_3 \), \( s_4 \) through \( s_7 \), or \( s_8 \) through \( s_{10} \). \( s_2 \sim s_7 \) because \( s_2 \) calls \( \text{Even} \) and \( s_7 \in \text{Blocks} (\text{Even}); s_5 \prec s_7 \) because \( s_5 \) precedes \( s_7 \); \( s_0 \uparrow s_1 \) because \( s_0 \) belongs to the if-branch and \( s_1 \) belongs to the else-branch; \( s_8 \parallel s_9 \) because they are running in parallel.

**Lemma 2.3.** For any two statement blocks \( s \) and \( t \), \( s \sim t \) if and only if exactly one of the following relations holds: \( s < t \), \( s \uparrow t \), \( t < s \), \( t \parallel s \) or \( s \parallel t \).

Read&Write analysis. In our framework, data dependences are represented and analyzed at the block level. We perform a static analysis over the program to extract the sets of local fields and variables being accessed in each non-call block. Intuitively, we use several read sets and write sets to represent local fields and global variables being read or written, respectively, in each statement block.

For every non-call block \( s \), we build the read set \( R_s \) by adding all data fields and local variables that occur on the RHS of an assignment. The data fields can be from the current node (such as \( n.i.v \)) or a neighbor node (such as \( n.l.v \)). The write set \( W_s \) can be built similarly: all data fields and local variables that occur on the LHS of an assignment are added.

### 3 Iteration Representation

As we mentioned above, code blocks (function calls or straight-line assignments) are building blocks of \( \text{Retreet} \) programs and are a key to our framework. In our running example (Figure 4), there are 11 blocks. Then the execution of a \( \text{Retreet} \) program is a sequence of iterations, each running a non-call code block on a tree node. For example, consider executing our running example on a single-node \( u \) (i.e., \( u.l = u.r = \text{nil} \)), one possible execution is a sequence of iterations (also called instances in the literature): 

\[
(s_0, u.l), (s_0, u.r), (s_7, u), (s_4, u.l), (s_4, u.r), (s_3, u)
\]

Note that every iteration is unique and appears at most once in a traversal, as per the obviously terminating restriction of \( \text{Retreet} \). However, this representation is not sufficient to reason about the dependences between steps. For example, if the middle steps \( (s_7, u), (s_4, u.l) \) were swapped, is that still a possible sequence of execution? The question can’t be answered unless we track back the contexts in which the two steps are executed: \( (s_7, u) \) is executed in the call to \( \text{Even}(u) \) (block \( s_9 \)); \( (s_4, u.l) \) is executed in the call to \( \text{Even}(u.l) \) (block \( s_1 \), which is further in \( \text{Odd}(u) \) (block \( s_8 \)). As the two calls are running in parallel, swapping the two steps yields another legal sequence of execution. Automating this kind of reasoning is extremely challenging. In fact, even determining if an iteration exists is already undecidable:

**Theorem 3.1.** Determining if an iteration may occur in a \( \text{Retreet} \) program execution is undecidable.

**Proof.** We prove the undecidability through a reduction from the halting problem of 2-counter machines [17]. We can build a \( \text{Retreet} \) program to simulate the execution of a 2-counter machine. Given a 2-counter machine \( M \), every line of non-halt instruction \( c \) in \( M \) can be converted to a function in a \( \text{Retreet} \) program. The function is of the form \( f_c(n, v_1, v_2) \); \( n \) is a \( \text{Loc} \) parameter and \( v_1, v_2 \) are \( \text{Int} \) parameters. It treats \( v_1, v_2 \) as the current values of the two counters, updates the two counter values to \( u_1, u_2 \) by simulating the execution of \( c \), then recursively calls \( f_{c'}(n.i) \) if \( c' \) is the next instruction. For the halt instruction, a special function \( f_{\text{halt}} \) will pass up the signal by recursive calls, and finally run a special line of code \( s \) on the root. Then \( M \) halts if and only if the iteration \((s, \text{root})\) occurs.

\[\Box\]
As precise reasoning about return values is expensive and leads to undecidability as per Theorem 3.1, we make two key assumptions below which make it possible for configurations to be abstractions of real call stacks in an execution:

1. all function calls not shown in the stack can return arbitrary values;
2. a call stack is valid if every pair of adjacent records in the stack are consistent.

With these two assumptions, we can now formally define configuration:

**Definition 3.3 (Configuration).** A configuration of length $k$ on a tree $T$ is a mapping $C : [k] \rightarrow \text{AllBlocks} \times \text{Nodes}(T) \times (\text{AllParams} \cup \text{AllCalls} \rightarrow \mathbb{Z})$ such that:

- For any $0 \leq i < k$, $C(i)$ is of the form $(s, u, M)$ where $s \in \text{AllCalls}$ is a call to a function $f$, and $M$ is only defined on $\text{Params}(f) \cup \text{Blocks}(f)$.
- The last record $C(k)$ is of the form $(\text{main}, u, 0)$, where $s \in \text{AllNonCalls}$.
- The first record $C(0)$ is of the form $\text{main}$, $\text{root}_f$, $\ldots$).
- For any two adjacent records $C(i) = (s, u, M)$, $C(i) = (t, v, N)$, $s$ is a call to the function that $t$ belongs to (denoted as $s \ast t$, see Figure 6). Moreover, $(s, u, M)$ speculatively reaches $(t, v, N)$.

The speculative reachability mentioned in the last condition of the definition above does not relate to any concrete run of a program and is a key notion that captures the second assumption we made above. In other words, we consider two adjacent records consistent if the first one can speculatively reach the second one. We next define speculative reachability formally.

**3.2 Speculative Reachability**

Intuitively, a record $(s, u, M)$ speculatively reaches (or just reaches for short) another record $(t, v, N)$ if an execution triggered by $(s, u, M)$ can lead to the next record $(t, v, N)$. More concretely, if $s$ is a call to a function $f$, then one can run $f$ on node $u$, with initial integer arguments from $M|_{\text{Params}(f)}$. Whenever a function call within the body of $f$ is encountered, one just skips the call and returns the speculative output from $M|_{\text{Calls}(f)}$. The execution should lead to a run of block $t$ on node $v$. If $t$ is also a function call, the input arguments for the call should match the expected, speculative inputs from $N$. We call this execution process a speculative execution.

**Definition 3.4 (Speculative Execution).** Given a function $f$, a group of initial values $I : \text{Params}(f) \rightarrow \mathbb{Z}$ and a group of speculative outputs $O : \text{Calls}(f) \rightarrow \mathbb{Z}$, a speculative execution of $f$ with respect to $I$ and $O$ follows the following steps:

1. initialize each parameter $p$ with value $I(p)$, and let the current block $c$ be the first block in $f$;
2. if $c$ is not a call, then simulate the execution of $c$, and move to the next block;
3. if $c$ is a call of the form $v = q(l,e, i)\bar{e}$, then update the special field $eq.g$’s value with $O(c)$.

With the formal definition above, we can formulate the speculative reachability using logical formulae. Note that
the weakest precondition may be nondeterministic due to the concurrency. However, there are only finitely many possible paths with statement-level interleaving and each path is of finite length. Then for each concrete path, the speculative execution of a function is completely deterministic as all initial parameters and return values from function calls are determined by $M$. More specifically, for every code snippet $l$ without branching and every logical constraint $\varphi$ that should be satisfied after running $l$, we can compute the weakest precondition $wp(l, \varphi, M)$ that must be satisfied before running $l$. The definition of $wp$ is shown in Figure 8.

Now if $s$ is a call to function $g$, we can determine if the speculative execution of $g$ with respect to $M$ hits block $t$. The path from the entry point of $g$ to $t$ will be a straight-line sequence of statements of the form

$$\ell_1; \text{assume}(c_1); \ldots; \text{assume}(c_{n-1}); \ell_n; \text{Block } t$$

where every branch condition is converted to a corresponding $\text{assume}(c_i)$. Then we can compute the path condition for $t$ by computing the weakest precondition for every condition $c_i$ on the path:

$$WP(c_i, M) \equiv wp(\ell_1; \ldots; \ell_i; \text{assume}(c_i); \ell_{i+1}; \ldots; \ell_n, M)[M(\tilde{p})/\tilde{p}]$$

where $\tilde{p}$ is the sequence of arguments for $g$.

Moreover, when $t$ is another call block, we also need to make sure that the initial parameters in $N$ match the speculative execution of the above code sequence w.r.t. $M$. We denote this condition as $\text{Match}_{\ell,t}(u, v, M, N)$.

**Lemma 3.5.** Let $(s, u, M)$ and $(t, v, N)$ be two records such that $s \prec t$ (as defined in Figure 6). Then $(s, u, M)$ speculatively reaches $(t, v, N)$ if $(u, v, M, N)$ satisfies

$$\text{PathCond}_{\ell,t}(u, v, M, N) \equiv \text{Match}_{\ell,t}(u, v, M, N) \land \bigvee_{P \in \text{Paths}(t)} \bigwedge_{c \in P} WP(c, M)$$

**Examples.** We present several examples to illustrate how the paths and path conditions are determined.

**Example 3.6.** Consider a code block $s$ calling a function $\text{foo}(n, p, r0) \{ n.f = p + 1 \ ; \ r1 = r0; \}$ if $(n.f < r1) \} [...]$ else $\{ \text{foo}(n, l, p, r0) / \text{Block } t \}$. For record $(s, u, M)$ to reach record $(t, v, N)$, there is only one path on which there is one condition, $n.f < r1$, which occurs negatively. In other words, the code sequence reaching $t$ is $n.f = p + 1 \ ; \ r1 = r0; \text{assume}(n.f \geq r1); \text{Block } t$. In addition, since code blocks $s$ and $t$ invoke function foo on nodes $n$ and $n.l$, respectively, $\text{Match}(u, v, M, N)$ should ensure that $v$ is the left child of $u$, i.e. in this case, $\text{Match}(u, v, M, N) \equiv u.l = v$. Therefore the path condition can be represented as

$$\text{PathCond}_{\ell,t}(u, v, M, N) \equiv M(p) + 1 \geq M(r0) \land u.l = v$$

**Example 3.7.** This example illustrates how paths are determined in the presence of concurrency. Consider a function $\text{foo}(n) \{ v = 0; \text{if } (v == 1) \{ \text{foo}(n.l) / \text{Block } t \} \} \}$ in which the recursive call to $\text{foo}(n.l)$ is parallel to the assignment $v = 1$. Since every possible statement-level interleaving is considered, weakest preconditions for all the three possible paths are computed: 1) $v=0; v=1; \text{assume } v==1; \text{foo}(n.l); 2) v=0; \text{assume } v==1; v=1; \text{foo}(n.l); 3) v=0; \text{assume } v==1; \text{foo}(n.l);$ $v=1$. The recursive call $\text{foo}(n.l)$ is reachable in the first possible path.

**Example 3.8.** This example illustrates that non-recursive calls can be precisely handled without any speculation. Consider the code snippet $\text{foo}(n) \{ n.f = 0; \text{bar}(n); \text{if } (n.f == 1) \{ \text{foo}(n.l) / \text{Block } t \} \} \text{bar}(n) \{ n.f = 1; \}$ where function $\text{bar}$ is indeed a single assignment manipulating the local field $f$ of $n$. As we mentioned in Section 2.1, during preprocessing of function $\text{foo}$, calls to other functions on $n$ are always inlined to make all operations on fields of $n$ explicit. Function call to $\text{bar}(n)$ in $\text{foo}(n)$ will be inlined to $n.f = 1$, thus the recursive call to $\text{foo}(n.l)$ is obviously reachable.

## 4 Encoding to Monadic Second-Order Logic

The configuration-based abstraction described above allows us to encode the schedules and dependences between configurations to Monadic Second-Order (MSO) logic over trees, a well-known decidable logic. Furthermore, some common dependence analysis queries can be checked by checking MSO formulae. We show the encoding in this section. The syntax of the logic contains a unique root, two basic operators $\text{left}$ and $\text{right}$. There is a binary predicate $\text{reach}$ as the transitive closure of $\text{left}$ and $\text{right}$, and a special $\text{isNil}$ predicate with constraint $\forall v (\text{isNil}(v) \rightarrow \text{isNil}卫视(v)) \land \text{isNil}(卫卫(v))$.

### 4.1 Configurations, Schedules and Dependences

First of all, we need to encode configurations we presented in Section 3. Given a RETREAT program, we define the following labels (each of which is a second-order variable):

- for each code block $s$, introduce a label (a second-order variable) $L_s$ such that $L_s(u)$ denotes that there exists a record $(s, u, \ldots)$ in the configuration;
- for each branch condition $c$, introduce a label $C_c$ such that $C_c(u)$ denotes that $WP(c, M)$ is satisfied by a record of the form $(s, u, M)$;
- for each pair of blocks $s$ and $t$ such that $s \preceq t$, introduce a label $K_{s,t}$ such that $K_{s,t}(u, v)$ denotes that $\text{Match}_{s,t}(u, v, M, N)$ is satisfied by records $(s, u, M)$ and $(t, v, N)$.
Note that these labels allow us to build an MSO predicate \( \text{PathCond}_{\text{ast}} \) as an abstracted version of the path condition \( \text{PathCond}_{\text{ast}} \) defined in Lemma 3.5:

\[
\text{PathCond}_{\text{ast}}(u, v) \equiv K_{\ast}(u, v) \land \bigvee_{p \in \text{Path}(t)} \left( \bigwedge_{c \in p} C(c)(u) \right)
\]

**Example 4.1.** The configuration in Figure 7a can be encoded to labels on the tree in Figure 7b. Note that the labels \( C_0 \) and \( C_3 \) are labeled on nil nodes only. If a node has a particular label, the node belongs to the set represented by the corresponding second-order variable. For example, node \( u \) is in \( L_{\text{ss}} \) but nodes \( r, v \) and \( w \) are not.

As the set of blocks and the set of conditions are fixed and known, we can simply represent these second-order variables using labeling predicates \( L \subseteq \text{AllCalls} \times \text{Nodes}(T) \) and \( C \subseteq \text{AllConds} \times \text{Nodes}(T) \) such that \( L(s, u) \) if and only if \( L(s, u) \) if and only if \( C(c, u) \). In other words, \( L(s, u) \) is the syntactic sugar for \( L(s, u) \) and \( C(c, u) \) is the syntactic sugar for \( C(c) \).

Now we are ready to encode configurations to MSO. We define a formula \( \text{Configuration}(L, C, q, v) \) below, which means \( L \) and \( C \) correctly represent a configuration with \( (q, v, \ldots) \) as the current record, for some non-call block \( q \):

\[
\text{Configuration}(L, C, q, v) \equiv L(\text{main}, \text{root}) \\
\land \text{Current}(L, q, v) \land \forall u. (u \neq v \rightarrow \bigwedge_{s \in \text{AllNonCalls}} \neg L(s, u)) \\
\land \forall u. \bigvee_{s \in \text{AllCalls}} \left( L(s, u) \rightarrow \bigvee_{s \in \text{AllConditions}} \left( \bigvee_{t, t'. s \,\in \, \text{AllConditions}} \neg \text{Next}(L, C, u, s, t, t') \right) \right) \\
\land \forall u. \bigwedge_{t \in \text{AllConditions}} \left( L(t, u) \rightarrow \text{Prev}(L, C, u, t) \right) \\
\land \forall u. \bigwedge_{c \in \text{ConsistentCondSet}} \left( \bigwedge_{c \in C} L(c, u) \land \bigwedge_{c \in C} \neg L(c, u) \right)
\]

The first three lines claim that main is marked on the root, and \( q \) is the only non-call block marked on the tree, where \( \text{Current}(L, q, v) \) is a subformula indicating that for the current node \( v \), a record \((q, v, \ldots)\) is in the stack for exactly one non-call block \( q \):

\[
\text{Current}(L, q, v) \equiv L(q, v) \land \bigwedge_{q' \in \text{AllNonCalls}, q' \neq q} \neg L(q', v)
\]

The next two lines, intuitively, say that every record has a unique successor (and predecessor) that can reach to (and from). Predicates \( \text{Next} \) and \( \text{Prev} \) are defined as below:

\[
\text{Next}(L, C, u, s, t) \equiv \exists v. \left( L(t, v) \land \text{PathCond}_{\ast}(u, v) \right)
\]

\[
\text{Prev}(L, C, u, t) \equiv \exists v. \left( \bigvee_{s \in C} \left( L(s, v) \land \text{PathCond}_{\ast}(v, u) \right) \\
\land \bigwedge_{s \in C, s' \neq s} \neg (L(s', v) \land \text{PathCond}_{\ast}(v, u)) \right)
\]

**Example 4.2.** Let us continue on Figure 7b. The labels on the tree show a valid instance of configuration for the running example in Figure 4. The root node \( r \) belongs to the second order variable \( L_{\text{main}} \). Block \( s3 \) running on node \( w \) is the only non-call block marked on the tree and node \( w \) is the only node that is running a non-call block. Along with the execution path \((r \rightarrow p \rightarrow q \rightarrow w)\), each record has a unique successor and predecessor. For example, node \( w \) labeled \( L_{s5} \) is the only successor of label \( L_{s1} \) running on node \( q \) and \( s1 \neq s5 \). In contrast, if the label \( L_{s5} \) on \( w \) is changed to \( L_{s2} \), the whole model is no longer a configuration because \( s1 \) does not call \( s2 \) directly (hence the third line of the formula is violated).

### 4.2 Schedules and Dependences

The definition and encoding of configurations above have paved the way for reasoning about RETREET programs. Given two configurations, a basic query one would like to make is about their order in a possible execution: can the two configurations possibly coexist? If so, are they always ordered? Or can they occur in arbitrary order due to the parallelization between them? To answer these questions, intuitively, we need to pairwisely compare the records in the two configurations from the beginning and find the place where they
Figure 10. Relations between consistent configurations

diverge. We define the following predicate:

\[
\text{Consistent}_{t_1, t_2}(L_1, L_2, C_1, C_2) \equiv \exists z, \forall u. \left( \text{reach}(u, z) \rightarrow \left( \bigwedge_{s} (L_1(s, v) \leftrightarrow L_2(s, v)) \right) \land \left( \bigwedge_{c} (C_1(c, v) \leftrightarrow C_2(c, v)) \right) \land L_1(s, z) \land L_2(s, z) \land \text{Next}(L_1, C_1, s, z, t_1) \land \text{Next}(L_2, C_2, z, z, t_2) \right)
\]

The predicate assumes that there are two sequences of records represented as \((L_1, C_1)\) and \((L_2, C_2)\), respectively, and indicates that there is a diverging record \((s, z, \ldots)\) in both sequences such that: 1) the two configurations match on all records prior to the diverging record; 2) the next records after the diverging one are \((t_1, \ldots)\) and \((t_2, \ldots)\), respectively, and they can be reached at the same time (i.e., \(C_1\) and \(C_2\) agree on the diverging node \(z\)).

Blocks \(t_1\) and \(t_2\) are obviously in the same function. If \(t_1 \neq t_2\), there are two possible relations between them: a) if \(t_1\) precedes \(t_2\) (or symmetrically, \(t_2\) precedes \(t_1\)), then configuration \((L_1, C_1)\) always precedes \((L_2, C_2)\) (or vice versa); b) otherwise, \(t_1\) and \(t_2\) must be two parallel blocks, then the two configurations occur in arbitrary order. Both relations can be described in MSO (see Figure 10).

Example 4.3. Let the configuration shown in Figure 7b be denoted as \((L_3, C_3)\). Consider another configuration \((L_4, C_4)\) shown in Figure 9a with execution path \(r \rightarrow p \rightarrow q \rightarrow m\). Instead of labeling \(L_{35}\) and \(L_{33}\) on node \(w\), \(L_{56}\) and \(L_{33}\) is labeled on node \(m\). All the other labels on nodes \(r, p, q\) in \(L_4, C_4\) are the same with the ones in \(L_3, C_3\). In this case, \(\text{Consistent}_{t_1, s_5, s_6}(L_3, L_4, C_3, C_4)\) and \(q\) is the node where two configurations diverge. Since \(s_1 < s_5, s_1 < s_6\) and \(s_5 < s_6\), \(\text{Ordered}(L_3, L_4, C_3, C_4)\). In other words, configurations \((L_3, C_3)\) and \((L_4, C_4)\) are ordered.

Another set of relations is necessary to describe the data dependencies. Recall that we use a read&write analysis to compute the read set \(R_s\) and write set \(W_s\) for each non-call block \(s\). These sets allow us to define two binary predicates: \(\text{Write}(u, v)\) if running \(s\) on \(u\) will write to \(v\); \(\text{ReadWrite}(u, v)\) if running \(s\) on \(u\) will read or write to \(v\). The following predicate describes two configurations \((L_1, C_1, s, u)\) and \((L_2, C_2, t, v)\) with data dependence if both last records \((s, u, \ldots)\) and \((t, v, \ldots)\) access the same node \(z\) and at least one of the accesses is a write:

\[
\text{Dependence}_{t, s}(u, v, L_1, L_2, C_1, C_2) \equiv \\
\text{Configuration}(L_1, C_1, s, u) \land \text{Configuration}(L_2, C_2, t, v) \\
\land \exists z. \left( \text{ReadWrite}(u, z) \land \text{Write}(v, z) \right) \\
\lor \left( \text{Write}(u, z) \land \text{ReadWrite}(v, z) \right)
\]

Example 4.4. Considering another configuration \((L_5, C_5)\) with execution path \(r \rightarrow p \rightarrow q \rightarrow s\) as shown in Figure 9b. The labels in configuration \((L_5, C_5)\) on nodes \(r, p, q\) are the same with the ones in configuration \((L_3, C_3)\). Labels \(L_3\) and \(L_7\) are on node \(q\). Thus \(\text{Dependence}_{s_3, s_5}(w, q, L_5, L_3, C_3, C_5)\) is true since \(s_3\) is writing \(n.Odd\) on node \(w\) while \(s_7\) is reading \(n.Odd\) on \(w\).

4.3 Data Race Detection and Equivalence Checking

Now we are ready to encode some common dependence analysis queries to MSO. A data race may occur in a Retreet program \(P\) if there exist two parallel configurations between which there is data dependence:

\[
\text{DataRace}[P] \equiv \bigvee_{q_1, q_2 \in \text{NonCalls}} \exists x_1, x_2, L_1, L_2, C_1, C_2. \\
\text{Dependence}_{q_1, q_2}(x_1, x_2, L_1, L_2, C_1, C_2) \\
\land \text{Parallel}(L_1, L_2, C_1, C_2)
\]

**Theorem 4.5.** A Retreet program \(P\) is data-race-free if \(\text{DataRace}[P]\) is invalid.

**Proof.** If \(P\) is not data-race-free, there must exist two iterations, represented as \((L_1, C_1)\) and \((L_2, C_2)\) and running blocks \(q_1\) and \(q_2\) on nodes \(x_1\) and \(x_2\), respectively, such that there is data dependence but no happens-before relation between them. This pair witnesses the validity of the formula \(\text{DataRace}[P]\), as \(\text{Dependence}\) encodes data dependences and \(\text{Parallel}\) encodes the absence of happens-before. \(\square\)

Besides data race detection, another critical query is the equivalence between two Retreet programs, which is common in program optimization. For example, when two sequential tree traversals \(A()\); \(B()\) are fused into a single traversal \(AB()\), one needs to check if this optimization is valid, i.e., if \(A()\); \(B()\) is equivalent to \(AB()\). Again, while the equivalence checking is a classical and extremely challenging problem, we focus on comparing programs that are built on the same set of straight-line blocks and simulate each other. The comparison is sufficient since the goal of the Retreet framework is to automate the verification of common program transformations such as fusion or parallelization, which only reorder the operations of a program.

**Definition 4.6.** Two Retreet programs \(P\) and \(P'\) bisimulate if there exists a mapping between blocks \(\text{Sim}: \text{AllBlocks}(P) \rightarrow \text{AllBlocks}(P')\) such that
blocks and translating (as they end with the same blocks).

Therefore the two executions are equivalent. The configurations keep the same order for all pairs of dependent it-

tations between them is undefined.

Conflict more, as exactly the same set of iterations, and vice versa. Further-

more, as Conflic[P,P′] is invalid, the corresponding execu-

tions keep the same order for all pairs of dependent iter-
ations. Therefore the two executions are equivalent. The

\[ \text{Conflict}[P,P′] \equiv \bigvee \text{AllNonCalls}(q_1,q_2) \] 

\[ \text{Dependence}^P(x_1,x_2,L_1,L_2,C_1,C_2) \] 

\[ \text{Ordered}^P(L_1,L_2,C_1,C_2) \] 

\[ \text{Ordered}^C(L_1,C_1,C_2) \] 

\[ \text{Ordered}^C(L_2,C_1,C_2) \] 

Theorem 4.7. For any two data-race-free RETREET programs P and P′ that bisimulate, they are equivalent if Conflic[P,P′] is invalid.

Proof. According to Definition 4.6, it can be proved by recur-
sion that there is a one-to-one correspondence between the configurations for P and the configurations for P′ such that the corresponding configurations are running the same block of code. Therefore for any execution of P, P′ can run exactly the same set of iterations, and vice versa. Furthermore, as Conflic[P,P′] is invalid, the corresponding executions keep the same order for all pairs of dependent iterations. Therefore the two executions are equivalent. The correctness of the formula encoding can be verified by readers.

Theorem 4.8. The MSO encoding for Theorems 4.5 and 4.7 is incomplete.

Proof. Since the outputs of speculative execution are arbitrary, the precision of the path conditions is lost. Consider a function f as shown in Figure 11 where height and size recursively compute the height and size of the tree, respectively. Due to speculative execution, the call to f(n,l) is considered reachable since arbitrary h and s values are legal. However, f(n,l) is unreachable during real computation since height of a tree can never be greater than the size of the tree.

5 Evaluation

We prototyped the RETREET framework, which implements all techniques presented above and also incorporates other existing MSO-based analysis techniques. We evaluated the effectiveness and efficiency of the framework through four case studies: mutually recursive size-counting traversals, CSS minification, cyclertree routing, and list sum-and-shift traversals. For the first two case studies, we synthesized provably-correct optimizations (parallelizing a traversal and/or fusing multiple traversals) using MSO encoding. More concretely, our prototype constructed a candidate fused program by heuristically enumerating possible mappings that establish the bisimulation relation between the original and fused programs, and finally checked their data-race-freeness and equivalence using the MSO encoding presented in this paper. For cyclertree routing, our prototype automatically verified some manually-crafted optimizations. The list sum-and-shift traversals, our prototype verified known optimizations using a combination of configuration-based abstraction presented in this paper and the Streaming Register Transducer (SRT) techniques for streaming list traversals [20]. To the best of our knowledge, none of these verification tasks can be automatically done by existing techniques before RETREET.

Our framework leverages MONA [7], a state-of-the-art WS2S (weak MSO with two successors) logic solver as our back-end constraint solver. All experiments were run on a server with a 40-core, 2.2GHz CPU and 128GB memory running Fedora 26. The bisimulation checking step is currently done by hand but can be automated in the future. The
We manually verified that the counterexample is a true positive. To verify a CSS minification pattern, the Abstract Syntax Tree (AST) of the CSS code is traversed several times to identify identifiers, reducing whitespaces, etc. In the case that the same AST is traversed multiple times, fusing the traversals together would be desirable to enhance the performance of minification process.

Hence, we consider fusing three CSS minification traversals. Traversal ConvertValues converts values to use different units when conversion result in smaller CSS size. For instance, 100ms will be represented as .1s. Traversal MinifyFont will try to minimize the font weight in the code. For example, font-weight: normal will be rewritten to font-weight: 400. Traversal ReduceInit reduces the CSS size by converting the keyword initial to corresponding value when keyword initial is longer than the property value. For example, min-width: initial will be converted to min-width: 0. Notice that these programs involve conditions on string which are not supported by Retreet. Nonetheless, since the traversals in Figure 13 only manipulate the local fields of the AST, these conditions can be replaced by some simple arithmetic conditions. Moreover, as the ASTs of CSS programs are typically not binary trees and cannot be handled by Mona directly, we converted the ASTs to left-child right-sibling binary trees and then simplify the traversals to match Retreet syntax. The three minification traversals are fused and their fusibility was checked in 6.88s.

We believe Retreet is the first framework to synthesize and verify these CSS traversal fusions. The CSS minification technique proposed by Hague et al. [10] also aims to generate minimized CSS file with the original semantics of the file preserved. However, they focus on one type of CSS minification method, called rule-merging, only, while Retreet can reason about the fusibility of different kinds of CSS minification methods.

**Cycletree Routing.** Our most challenging case study is about Cycletrees [29], a special class of binary trees with an additional set of edges. These additional edges serve the purpose of constructing a Hamiltonian cycle. Hence, cycletrees are especially useful when it comes to different communication patterns in parallel and distributed computation. For instance, a broadcast can be efficiently processed by the tree structure while the cycle order is suitable for point-to-point

```plaintext
Figure 12. Fusing two mutually recursive traversals

<table>
<thead>
<tr>
<th>Fused(n)</th>
<th>Fused(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n == nil) return (0, 0) if (n == nil) return (0, 0) else (ls, lv) = Fused(n.l) (ls, lv) = Fused(n.l) (rs, rv) = Fused(n.r) (rs, rv) = Fused(n.r) return (ls + rs + 1, lv + rv) return (ret1, ret2)</td>
<td></td>
</tr>
</tbody>
</table>

(a) A valid fusion (b) An invalid fusion

Figure 13. CSS minification traversals

<table>
<thead>
<tr>
<th>ConvertValues(n)</th>
<th>Main(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (n == nil) return 0 if (n == nil) return 0 else for each child p: ConvertValues(n.p) for each child p: MinifyFont(n.p) if (n.type == &quot;word&quot;</td>
<td></td>
</tr>
</tbody>
</table>
```
communication. Cicletrees are proven to be an efficient network topology in terms of degree and number of communication links [28–30].

We consider two traversals regarding cicletrees. A traversal, called RootMode, is a mutually recursive traversal that constructs the cyclic order on a binary tree to transform the binary tree to a cicletree. Another traversal ComputeRouting computes the router data of each node which are essential for an efficient cicletree routing algorithm presented in [29]. In the event of cyclic order traversal and routing had to be performed repeatedly—in case of link failures—it would be useful to think about ways we can optimize these procedures by fusion or parallelization. Figure 14 shows the code for these two traversals.

We first consider checking the fusibility of these two traversals RootMode and ComputeRouting. Since the mapping relation between the unfused traversals and expected fused one is very subtle and does not satisfy the bisimulation relation defined in Definition 4.6, we designed the fused traversal manually and apply RETREET to verify the correctness of the fusion. The total time spent to verify the fusibility of these two traversals was 490.55s.

Figure 14. Ordered cicletree construction and routing data computation

Figure 15. Two functions traversing a list

We then considered whether the two traversals can run in parallel. This time MONA spent 0.95s and returned a counterexample which allows us to discover a data race that violates a read-after-write dependence. We manually verified that the counterexample is indeed a true positive.

List Sum and Shift. In this case study, we show how RETREET integrates other MSO-based techniques and enables optimizations not possible with any of the techniques alone. Consider the two list traversals discussed in [25] (as shown in Figure 15a). Traversal Sum updates the local fields v in the list to the aggregation of values v in the list. Traversal Shift shifts the element in the list to the left and sets the last element in the list to be 0. A program invokes Sum followed by Shift. Sakka [25] shows the two traversals can be fused at the cost of an extra field for each node. However, if one swaps the order of the two traversals (step 1, from Sum(n);Shift(n) to Shift(n);Sum(n)), they can be fused without introducing the extra field and form the optimal program (step 2, from Shift(n);Sum(n) to Figure 15b). While the core RETREET can verify step 2, unfortunately, it is not sufficient to verify step 1, since there does not exist a relation between the original and the swapped traversals that preserves all data dependences in the original program.

Nonetheless, we extended RETREET to support other existing MSO-based analysis techniques. For example, both Sum and Shift can be described by streaming register transducer (SRT) [20], an automaton-based machine model for what they call streaming transformations with additive operations, which are essentially list traversals. It is also shown in [20] that these traversals are closed under composition and can be defined in MSO. The crux of the proof is: for every node y of the output list, there exists a set of nodes N(y) from the input list such that the data value stored in y is the sum of values stored in N(y). Following their encoding, we can define two MSO predicates:

\[
\text{sum}(x, y) \equiv x \leq y
\]

\[
\text{shift}(x, y) \equiv x.next = y
\]
such that $\text{sum}(x,y)$ (resp. $\text{shift}(x,y)$) means $x$ belongs to the set $N(y)$ for traversal $\text{Sum}$ (resp. $\text{Shift}$). We can further encode similar predicates for “sum then shift” and “shift then sum”, respectively:

$$
\text{sum_shift}(x,y) \equiv \exists z. \text{shift}(x,z) \land \text{sum}(z,y)
$$

$$
\text{shift_sum}(x,y) \equiv \exists z. \text{sum}(x,z) \land \text{shift}(z,y)
$$

Then RETREET verifies the validity of step 1 by checking the validity of the following formula:

$$
\text{shift_sum}(x,y) \iff \text{sum_shift}(x,y)
$$

Furthermore, RETREET verifies the validity of step 2 using an encoding similar to the tree-mutation example. The whole chain of optimization was verified automatically, for the first time, in 0.11s.

6 Related Work

There has been much prior work on program dependence analysis for tree data structures. Using shape analyses [13], Ghia et al. [8] detect function calls that access disjoint sub-trees for parallel computation in programs with recursive data structures. Rugina and Rinard [24] extract symbolic lower and upper bounds for the regions of memory that a program accesses. Instead of providing a framework that describes dependences in programs, these works only focus on detecting the data races and the potential of parallel computing so that is not able to handle fusion or other transformations.

Amiranoff et al. [1] propose instance-wise analysis to perform dependence analysis for recursive programs involving trees. This framework represents each dynamic instance of a statement by an execution trace, and then abstracts the execution trace to a finitely-presented control word. Nonetheless, the framework does not support applications other than parallelization and they cannot handle programs with tree mutation. Weijiang et al. [32] also present a tree dependence analysis framework that reason the legality of point blocking, traversal slicing and parallelization of traversals with the assumption that all traversals are identical preorder traversals. Their framework allows restricted tree mutations including nullifying or creating a subtree but the traversals that they consider are also single node traversals like RETREET. Deforestation [5, 9, 14, 23, 31] is a technique widely applied to fusion, but it either does not support fusion over arbitrary tree traversals, or does not handle reasoning about imperative programs.

The last decade has seen significant efforts on reasoning transformations over recursive tree traversals. Meyerovich and Bodik [15] and Meyerovich et al. [16] focus on fusing tree traversals over ASTs of CSS files. They specify tree traversals as attribute grammars and present a synthesizer that automatically fuses and parallelizes the attribute grammars. Their framework only supports traversals that can be written as attribute grammars, basically layout traversals. Rajbhandari et al. [21] provide a domain specific fusion compiler that fuses traversals of $k$-$d$ trees in computational simulations. Both frameworks are ad hoc, designed to serve specific applications. The tree traversals they can handle are less general than RETREET.

Most recently, TreeFuser presented by Sakka et al. [26] is an automatic framework that fuses tree traversals written in a general language. TreeFuser supports code motion and partial fusion, i.e., parts of a traversal (left subtree or right subtree) can be fused together when possible, even if the traversals cannot be fully fused. Their approach cannot handle transformations other than fusion. In other words, parallelization of traversals is beyond the scope of TreeFuser. Besides, TreeFuser also suffers from the restrictions that RETREET has, i.e. no tree mutation and single node traversal. PolyRec [27] is a framework that can handle schedule transformations for nested recursive programs only. PolyRec targets a limited class of tree traversals, called perfectly nested recursive programs, hence the framework is not able to handle arbitrary recursive tree traversals. Also PolyRec does not handle dependence analysis and suffers from the restriction that no tree mutation is allowed. The transformations that they handle are interchange, inlining and code motion rather than fusion and parallelization. Another deforestation transformation proposed by Sakka [25] combines fusion and tupling to optimize functional programming. Their framework focuses on runtime complexity and termination guarantees, hence they do not handle dependence analysis either. None of the dependence analysis in the frameworks above is expressive enough to handle mutual recursion.

7 Conclusion

We introduced RETREET, a general tree-traversal-describing language, and developed a stack-based, fine-grained representation of dynamic instances in a tree traversal. Based on the new language and new representation, we presented a MSO encoding that can check data-race-freeness and transformation correctness automatically. Our approach is more general than existing approaches and allows us to efficiently reason about traversals with sophisticated mutual recursion on real-world data structures such as CSS and cycle-trees, and synthesize provably-correct optimizations. We also show our approach can be integrated with other MSO-based analysis techniques.

Acknowledgments

We would like to thank Milind Kulkarni and Kirshanthan Sundararajah for the fruitful discussions we had when we started this project.

This research was supported in part by the National Science Foundation under Grant No. CCF-1919197.
References


