Reasoning About Recursive Tree Traversals

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

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

Dalin Zhang
Beijing Jiaotong University
Beijing, China
dalin@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 cycletrees, 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:
Yanjun Wang, Jinwei Liu, Dalin Zhang, and Xiaokang Qiu. 2021. Reasoning About Recursive Tree Traversals. In *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21), February 27-March 3, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3437801.3441617

PPoPP '21, February 27-March 3, 2021, Virtual Event, Republic of Korea

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) [1] 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).

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

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 Cycletree 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 *Block*s 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 *AExpr*'s 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 *AExpr*'s 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 $g(n, \bar{v})$ should not call, directly or indirectly through inlining, itself, i.e., $g(n, \dots)$ 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.l 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*

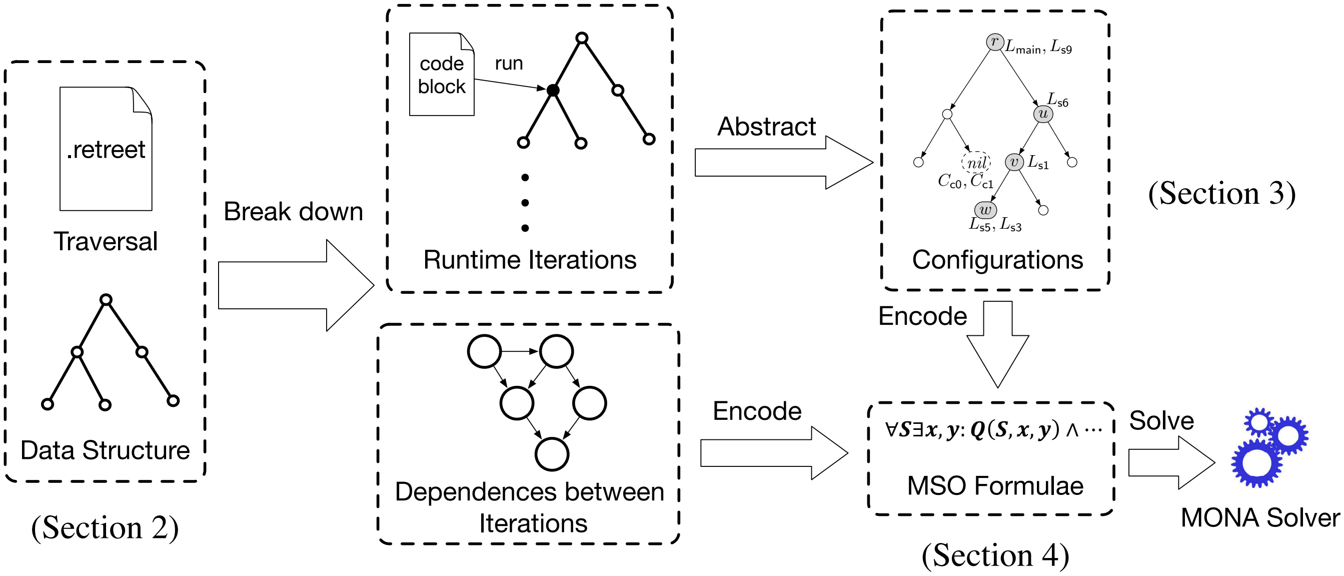


Figure 1. RETREET reasoning framework

$$
\begin{array}{ll}
dir \in \text{Loc Fields} \quad v \in \text{Int Vars} & n \in \text{Loc Vars} \\
f \in \text{Int Fields} & g : \text{Function IDs}
\end{array}
$$

$$
\begin{array}{rcl}
LExpr & ::= & n \mid LExpr.dir \\
AExpr & ::= & 0 \mid 1 \mid n.f \mid v \mid AExpr + AExpr \mid AExpr - AExpr \\
BExpr & ::= & LExpr == \text{nil} \mid \text{true} \mid AExpr > 0 \mid !\ BExpr \\
 & & \mid BExpr \ \&\&\ BExpr \\
Assgn & ::= & n.f = AExpr \mid v = AExpr \\
Block & ::= & \bar{v} = g(LExpr, \overline{AExpr}) \mid Assgn^{+} \\
Stmt & ::= & Block \mid \text{if } (BExpr)\ Stmt \text{ else } Stmt \mid Stmt\ ;\ Stmt \\
 & & \mid \{Stmt \parallel Stmt\} \\
Func & ::= & g(n, \bar{v})\{\ Stmt\ \} \\
Prog & ::= & Func^{+}
\end{array}
$$

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 $g(n, \bar{v})$ 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) + ... 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 restriction 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

```
Odd(n)
  if (n == nil) // c0
    n.Odd = 0 // s0
  else
    Even(n.l)  // s1
    Even(n.r)  // s2
    n.Odd = n.l.Even +
            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)
  { Odd(n) || // s8
    Even(n) }  // s9
  n.Main = (n.Odd, n.Even)  // s10
```

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

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. [3]

For the simplification of presentation, in the rest of the paper, we also assume: all trees are binary with two pointer fields l and r, every function only calls itself or other functions on n.l or n.r, and returns only a single Int value (which is rewritten to a special local field), and every boolean expression is atomic, i.e., of the form *LExpr* == nil or *AExpr* > 0. Calls to other functions on n are always inlined to make all operations on fields of n explicit. In addition, we assume the program is free of null dereference, i.e., every term *le.dir* is preceded by a guard *le* != nil. Note that relaxing these assumptions will not affect any result of this paper, because any Retreet program violating these assumptions can be easily rewritten to a version satisfying the assumptions.

2.2 Code Blocks

With assumptions made above, Retreet programs can be decomposed to code blocks, which are a key to our framework. Each code block is a function call or a sequence of straight-line, non-call assignments derived from the *Block* symbol of the grammar (see Figure 2). We use some necessary notations for blocks, of which the meaning is determined by the syntactic structure of the program. Figure 5 lists common sets of functions, blocks, parameters and nodes that will be frequently used in this paper.

We also define the possible relations between blocks, as shown in Figure 6. Every function's body can be represented as a syntax tree whose leaves are statement blocks and non-leaf nodes are sequentials, conditionals or parallels. Then the relation between two statement blocks is determined by

[3] Composed expressions, such as n.l.Even, are used for code readability. Accesses to the special local field are allowed when $n == nil$, since the transformed program is not executable and used for analysis only. In our MSO encoding (described in Section 4), $isNil$ is a special MSO predicate.

AllFuncs	the set of all functions
AllParams	the set of all Int function parameters
AllBlocks	the set of all blocks
AllCalls	the set of all blocks for function calls
AllNonCalls	the set of all blocks for straight-line non-call assignments
Blocks(f)	the set of all blocks belonging to a function f
Calls(f)	Blocks(f) ∩ AllCalls
Params(f)	the set of Int parameters for f
Nodes(T)	the set of all nodes in the tree T
Paths(t)	the set of all possible paths (through statement-level interleaving) to t from the entry point of the function that t belongs to

Figure 5. Commonly used notations

LCA(s, t)	The least common ancestor (LCA) of blocks s and t in the syntax tree.
s ◁ t	s is a function call to f and t ∈ Blocks(f).
s ∼ t	s and t are from the same function definition, i.e., s, t ∈ Blocks(f) for some function f.
s ≺ t	LCA(s, t) is a sequential, i.e., s precedes t.
s ↑ t	LCA(s, t) is a conditional, i.e., there is a conditional if (...) then A else B such that s and t belong to A and B, respectively.
s ∥ t	LCA(s, t) is a parallel, i.e., s and t can be executed in arbitrary order.

Figure 6. Relations between blocks

their positions in the syntax tree. In particular, when two blocks s ∼ 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 s0 through s10, as shown in the comment following each block. There are six call blocks: AllCalls = {s1, s2, s5, s6, s8, s9}; and five non-call blocks: AllNonCalls = {s0, s3, s4, s7, s10}. Take s6 for example, Path(s6) is just the path from the beginning of function Even (which s6 belongs to) to s6, i.e., from ¬ c1 to s5 then s6. The ∼ relation holds between any two blocks from the same group: s0 through s3, s4 through s7, or s8 through s10. s2 ◁ s7 because s2 calls Even and s7 ∈ Blocks(Even); s5 ≺ s7 because s5 precedes s7; s0 ↑ s1 because s0 belongs to the if-branch and s1 belongs to the else-branch; s8 ∥ s9 because they are running in parallel.

Lemma 2.3. *For any two statement blocks* s *and* t, s ∼ t *if and only if exactly one of the following relations holds:* s ≺ t, s ↑ t, t ≺ s, t ↑ s *or* s ∥ 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.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 Retreet programs and are a key to our framework. In our running example (Figure 4), there are 11 blocks. Then the execution of a 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 = nil), one possible execution is a sequence of iterations (also called instances in the literature):

$$(\mathsf{s0}, u.l), (\mathsf{s0}, u.r), (\mathsf{s7}, u), (\mathsf{s4}, u.l), (\mathsf{s4}, u.r), (\mathsf{s3}, u)$$

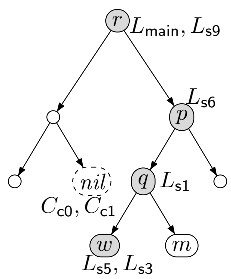
Note that every iteration is unique and appears at most once in a traversal, as per the obviously terminating restriction of Retreet. However, this representation is not sufficient to reason about the dependences between steps. For example, if the middle steps $(\mathsf{s7}, u), (\mathsf{s4}, u.l)$ were 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: $(\mathsf{s7}, u)$ is executed in the call to Even(u) (block s9); $(\mathsf{s4}, u.l)$ is executed in the call to Even(u.l) (block s1), which is further in Odd(u) (block s8). 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* 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 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 Retreet program. The function is of the form $f_c(\mathsf{n}, \mathsf{v}_1, \mathsf{v}_2)$: n is a Loc parameter and $\mathsf{v}_1, \mathsf{v}_2$ are Int parameters. It treats $\mathsf{v}_1, \mathsf{v}_2$ as the current values of the two counters, updates the two counter values to $\mathsf{u}_1, \mathsf{u}_2$ by simulating the execution of c, then recursively calls $f_{c'}(\mathsf{n.l})$ if c' the next instruction. for the halt instruction, a special function f_{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, *root*) occurs. ◻

R#	Content
0	(main, r, s8 = 5, . . .)
1	(s9, r, s5 = 3, . . .)
2	(s6, p, . . .)
3	(s1, q, . . .)
4	(s5, w, s1 = 0, s2 = 0)
5	(s3, w)

(a) A configuration



(b) Represented as labels on the tree

Figure 7. Example of configuration encoding

3.1 Configuration

As precise reasoning about Retreet is undecidable, we propose an iteration representation called *configuration*, which is a right level of abstraction for which automated reasoning is possible. Intuitively, a configuration looks like a snapshot of the call stack, which consists of multiple records. The last record describes the current running block as we discussed above. Each other record describes a call context which includes: the callee block, the single Loc parameter, and other Int variables' values. These Int values include: first, for each Int parameter, the context records its *initial* value received when the call begins; second, for each function call within the current call, the context uses *ghost variables* to predict the return values.

Example 3.2. Figure 7a gives an example of a configuration, which consists of 6 records. The last record (record #5) indicates that the current step is running block s3 on tree node w, and the current values of local variables. In other records, we only show the callee stack, the Loc parameter, and other relevant Int variables. For example, the value s8 = 5 means that the call in s8 is predicted to finish and return value 5, which might be relevant to the next call context, s9.

Obviously, not all stacks of records are valid configurations. In particular, the beginning record should run main and the last record should run a non-call block. More importantly, for any non-beginning record, one of the *path conditions* [4] of the block should be satisfied, i.e., this block of code can be reached from the beginning of the function it belongs to. While a precise characterization of these constraints 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 $C : [k] \to \mathsf{AllBlocks} \times \mathsf{Nodes}(T) \times (\mathsf{AllParams} \cup \mathsf{AllCalls} \rightharpoonup \mathbb{Z})$ such that:

- For any $0 \le i < k$, $C(i)$ is of the form (s, u, M) where $\mathsf{s} \in \mathsf{AllCalls}$ is a call to a function f, and M is only defined on $\mathsf{Params}(f) \cup \mathsf{Blocks}(f)$.
- The last record $C(k)$ is of the form $(\mathsf{s}, u, \emptyset)$, where $\mathsf{s} \in \mathsf{AllNonCalls}$.
- The first record $C(0)$ is of the form $(\mathsf{main}, root_T, ...)$.
- For any two adjacent records $C(i-1) = (\mathsf{s}, u, M)$, $C(i) = (\mathsf{t}, v, N)$, s is a call to the function that t belongs to (denoted as $\mathsf{s} \triangleleft \mathsf{t}$, 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 reaches 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 $M|_{\mathsf{Params}(f)}$. Whenever a function call within the body of f is encountered, one just skips the call and returns the speculative output from $M|_{\mathsf{Calls}(f)}$. 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 $I : \mathsf{Params}(f) \to \mathbb{Z}$ and a group of speculative outputs $O : \mathsf{Calls}(f) \to \mathbb{Z}$, 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 $v = g(le, \bar{ie})$, then update the special field $le.g$'s value with $O(\mathsf{c})$.

With the formal definition above, we can formulate the speculative reachability using logical formulae. Note that

[4] We consider all the finitely many possible statement-level interleavings.

$$
\begin{array}{lcl}
\mathsf{wp}(n.f = AExpr,\ \varphi,\ M) & = & \varphi[AExpr/n.f] \\
\mathsf{wp}(v = AExpr,\ \varphi,\ M) & = & \varphi[AExpr/v] \\
\mathsf{wp}(\bar{v} = t(\dots),\ \varphi,\ M) & = & \varphi[M(\mathsf{s})/v] \\
\multicolumn{3}{c}{\text{where s is the id of the current statement}} \\
\mathsf{wp}(\mathsf{l}\ ;\ \mathsf{l}',\ \varphi,\ M) & = & \mathsf{wp}(\mathsf{l}, \mathsf{wp}(l', \varphi))
\end{array}
$$

Figure 8. Weakest precondition

the speculative execution 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 φ that should be satisfied after running l, we can compute the weakest precondition $\mathsf{wp}(\mathsf{l}, \varphi, M)$ 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

$$\mathsf{l}_1; \mathsf{assume}(\mathsf{c}_1); \dots; \mathsf{assume}(\mathsf{c}_{n-1}); \mathsf{l}_n; \text{Block t}$$

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

$$WP(\mathsf{c}_i, M) \equiv \mathsf{wp}(\mathsf{l}_1; \dots; \mathsf{l}_i,\ \mathsf{c}_i,\ M)[M(\bar{p})/\bar{p}]$$

where $\bar{p}$ 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 $Match_{\mathsf{s},\mathsf{t}}(u, v, M, N)$.

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

$$Match_{s,t}(u, v, M, N) \wedge \bigvee_{P \in Paths(t)} \Big(\bigwedge_{c \in P} WP(c, M) \Big)$$

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 foo(n, p, r0) { n.f = p + 1 ; r1 = r0; if (n.f < r1) {...} else { foo(n.l, p, r0) // 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; assume (n.f ≥ r1); Block t . In addition, since code blocks s and t invoke function foo on nodes n and n.l, respectively, $Match(u, v, M, N)$ should ensure that v is the left child of u, i.e. in this case, $Match(u, v, M, N) \equiv u.\mathsf{l} = v$. Therefore the path condition can be represented as

$$PathCond_{\mathsf{s},\mathsf{t}}(u, v, M, N) \equiv M(\mathsf{p}) + 1 \geq M(\mathsf{r0}) \wedge u.\mathsf{l} = v$$

Example 3.7. This example illustrates how paths are determined in the presence of concurrency. Consider a function foo(n) { v = 0; if (v == 1) { foo(n.l) // Block t; } || v = 1; } in which the recursive call to 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; assume v==1; foo(n.l); 2) v=0; assume v==1; v=1; foo(n.l); 3) v=0; assume v==1; foo(n.l); v=1;. The recursive call 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 foo(n) { n.f = 0; bar(n); if (n.f == 1) { foo(n.l) // Block t } } bar(n) { n.f = 1; } where function bar is indeed a single assignment manipulating the local field f of n. As we mentioned in Section 2.1, during preprocessing of function foo, calls to other functions on n are always inlined to make all operations on fields of n explicit. Function call to bar(n) in foo(n) will be inlined to n.f = 1, thus the recursive call to 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 *left* and *right*. There is a binary predicate *reach* as the transitive closure of *left* and *right*, and a special *isNil* predicate with constraint $\forall v. \big(isNil(v) \rightarrow isNil(left(v)) \wedge isNil(right(v))\big)$.

4.1 Configurations, Schedules and Dependences

First of all, we need to encode configurations we presented in Section 3. Given a Retreet 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 $L_\mathsf{s}(u)$ denotes that there exists a record $(\mathsf{s}, u, \dots)$ in the configuration;
- for each branch condition c, introduce a label C_c such that $C_\mathsf{c}(u)$ denotes that $WP(\mathsf{c}, M)$ is satisfied by a record of the form (s, u, M);
- for each pair of blocks s and t such that $\mathsf{s} \triangleleft \mathsf{t}$, introduce a label $K_{\mathsf{s},\mathsf{t}}$ such that $K_{\mathsf{s},\mathsf{t}}(u, v)$ denotes that $Match_{\mathsf{s},\mathsf{t}}(u, v, M, N)$ is satisfied by records (s, u, M) and (t, v, N).

Note that these labels allow us to build an MSO predicate $\overline{PathCond_{s,t}}$ as an abstracted version of the path condition $PathCond_{s,t}$ defined in Lemma 3.5:

$$\overline{PathCond_{s,t}}(u,v) \equiv K_{s,t}(u,v) \wedge \bigvee_{P \in \mathsf{Paths}(t)} \Big(\bigwedge_{c \in P} C_c(u) \Big)$$

Example 4.1. The configuration in Figure 7a can be encoded to labels on the tree in Figure 7b. Note that the labels C_{c0} and C_{c1} 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_{s6} 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 $\mathcal{L} \subseteq \mathsf{AllCalls} \cup \mathsf{AllNonCalls} \times \mathsf{Nodes}(T)$ and $C \subseteq \mathsf{AllConds} \times \mathsf{Nodes}(T)$ such that $\mathcal{L}(s,u)$ if and only if $L_s(u)$, $C(c,u)$ if and only if $C_c(u)$. In other words, $\mathcal{L}(s,u)$ is the syntactic sugar for $L_s(u)$ and $C(c,u)$ is the syntactic sugar for $C_c(u)$.

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

$$\begin{aligned}
&Configuration(\mathcal{L}, C, q, v) \equiv \mathcal{L}(\mathsf{main}, root)\\
&\quad \wedge Current(\mathcal{L}, q, v) \wedge \forall u. \big(u \neq v \rightarrow \bigwedge_{s \in \mathsf{AllNonCalls}} \neg \mathcal{L}(s,u)\big)\\
&\quad \wedge \forall u. \bigwedge_{s \in \mathsf{AllCalls}} \Big(\mathcal{L}(s,u) \rightarrow \bigvee_{s \lessdot t} \big(Next(\mathcal{L}, C, u, s, t)\\
&\qquad\qquad \wedge \bigwedge_{t \sim t', t \neq t'} \neg Next(\mathcal{L}, C, u, s, t')\big)\Big)\\
&\quad \wedge \forall u. \bigwedge_{t \in \mathsf{AllCalls} \cup \mathsf{AllNonCalls}} \Big(\mathcal{L}(t,u) \rightarrow Prev(\mathcal{L}, C, u, t)\Big)\\
&\quad \wedge \forall u. \bigvee_{\mathsf{C} \in \mathsf{ConsistentCondSet}} \Big(\bigwedge_{c \in \mathsf{C}} C(c,u) \wedge \bigwedge_{c \notin \mathsf{C}} \neg C(c,u) \Big)
\end{aligned}$$

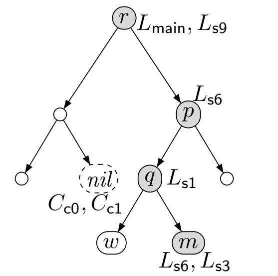
The first three lines claim that main is marked on the *root*, and q is the only non-call block marked on the tree, where $Current(\mathcal{L}, q, v)$ is a subformula indicating that for the current node v, a record $(q, v, \dots)$ is in the stack for exactly one non-call block q:

$$Current(\mathcal{L}, q, v) \equiv \mathcal{L}(q, v) \wedge \bigwedge_{q' \in \mathsf{AllNonCalls}, q' \neq q} \neg \mathcal{L}(q', v)$$

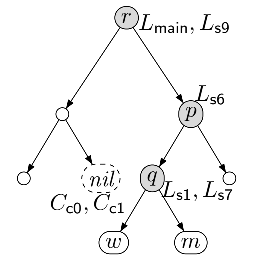
The next two lines, intuitively, say that every record has a unique successor (and predecessor) that can reach to (and from). Predicates *Next* and *Prev* are defined as below:

$$Next(\mathcal{L}, C, u, s, t) \equiv \exists v. \Big(\mathcal{L}(t, v) \wedge \overline{PathCond_{s,t}}(u, v) \Big)$$

$$\begin{aligned}
Prev(\mathcal{L}, C, u, t) \equiv \exists v. \Bigg(&\bigvee_{s \lessdot t} \Big(\mathcal{L}(s, v) \wedge \overline{PathCond_{s,t}}(v, u)\\
&\wedge \bigwedge_{s' \lessdot t, s' \neq s} \neg \big(\mathcal{L}(s', v) \wedge \overline{PathCond_{s,t}}(v, u) \big) \Big) \Bigg)
\end{aligned}$$



(a) A configuration running s3 on m



(b) A configuration running s7 on q

Figure 9. Examples of configuration

$Next(\mathcal{L}, C, u, s, t)$ indicates that a record $(t, v, \dots)$ exists and is reachable from record $(s, u, \dots)$. $Prev(\mathcal{L}, C, u, t)$ constrains that, for a record $(t, u, \dots)$, there should exist one and only one record $(s, v, \dots)$ that can reach $(t, u, \dots)$.

The last line makes sure that for each node u, the set of satisfied conditions C is consistent, i.e., $\bigwedge_{c \in \mathsf{C}} WP(c, M)$ is satisfiable for every record (s, u, M). In other words, a consistent condition set for a node u represents a feasible conditional path from the root of the tree to reach node u. Notice that this is a linear integer arithmetic constraint and SMT-solvable. Hence we can assume the set of all possible consistent condition set, denoted by ConsistentCondSet, has been computed a priori.

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_{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 $\lessdot$ 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 pairwise compare the records in the two configurations from the beginning and find the place where they

$$\begin{aligned}
&Ordered(\mathcal{L}_1, \mathcal{L}_2, C_1, C_2) \equiv \\
&\bigvee_{\substack{\mathsf{s},\mathsf{t}_1,\mathsf{t}_2 \\ \mathsf{s} \lessdot \mathsf{t}_1, \mathsf{s} \lessdot \mathsf{t}_2, \mathsf{t}_1 \prec \mathsf{t}_2}} Consistent_{\mathsf{s},\mathsf{t}_1,\mathsf{t}_2}(\mathcal{L}_1, \mathcal{L}_2, C_1, C_2) \\
&Parallel(\mathcal{L}_1, \mathcal{L}_2, C_1, C_2) \equiv \\
&\bigvee_{\substack{\mathsf{s},\mathsf{t}_1,\mathsf{t}_2 \\ \mathsf{s} \lessdot \mathsf{t}_1, \mathsf{s} \lessdot \mathsf{t}_2, \mathsf{t}_1 \| \mathsf{t}_2}} Consistent_{\mathsf{s},\mathsf{t}_1,\mathsf{t}_2}(\mathcal{L}_1, \mathcal{L}_2, C_1, C_2)
\end{aligned}$$

Figure 10. Relations between consistent configurations

diverge. We define the following predicate:

$$\begin{aligned}
&Consistent_{\mathsf{s},\mathsf{t}_1,\mathsf{t}_2}(\mathcal{L}_1, \mathcal{L}_2, C_1, C_2) \equiv \exists z.\Big[\\
&\quad \forall v.\Big(reach(v, z) \to \Big(\bigwedge_{\mathsf{s}} (\mathcal{L}_1(\mathsf{s}, v) \leftrightarrow \mathcal{L}_2(\mathsf{s}, v)) \\
&\quad\quad \wedge \bigwedge_{\mathsf{c}} (C_1(\mathsf{c}, v) \leftrightarrow C_2(\mathsf{c}, v))\Big)\Big) \\
&\quad \wedge \mathcal{L}_1(\mathsf{s}, z) \wedge \mathcal{L}_2(\mathsf{s}, z) \wedge Next(\mathcal{L}_1, C_1, z, \mathsf{s}, \mathsf{t}_1) \\
&\quad \wedge Next(\mathcal{L}_2, C_2, z, \mathsf{s}, \mathsf{t}_2)\Big]
\end{aligned}$$

The predicate assumes that there are two sequences of records represented as $(\mathcal{L}_1, C_1)$ and $(\mathcal{L}_2, C_2)$, respectively, and indicates that there is a diverging record $(\mathsf{s}, z, \dots)$ 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 $(\mathsf{t}_1, \dots)$ and $(\mathsf{t}_2, \dots)$, 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 $\mathsf{t}_1 \neq \mathsf{t}_2$, there are two possible relations between them: a) if t_1 precedes t_2 (or symmetrically, t_2 precedes t_1), then configuration $(\mathcal{L}_1, C_1)$ always precedes $(\mathcal{L}_2, C_2)$ (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 $(\mathcal{L}_3, C_3)$. Consider another configuration $(\mathcal{L}_4, C_4)$ shown in Figure 9a with execution path $r \to p \to q \to m$. Instead of labeling $L_{\mathsf{s}5}$ and $L_{\mathsf{s}3}$ on node w, $L_{\mathsf{s}6}$ and $L_{\mathsf{s}3}$ is labeled on node m. All the other labels on nodes r, p, q in $\mathcal{L}_4, C_4$ are the same with the ones in $\mathcal{L}_3, C_3$. In this case, $Consistent_{\mathsf{s}1,\mathsf{s}5,\mathsf{s}6}(\mathcal{L}_3, \mathcal{L}_4, C_3, C_4)$ and q is the node where two configurations diverge. Since $\mathsf{s}1 \lessdot \mathsf{s}5, \mathsf{s}1 \lessdot \mathsf{s}6$ and $\mathsf{s}5 \prec \mathsf{s}6$, $Ordered(\mathcal{L}_3, \mathcal{L}_4, C_3, C_4)$. In other words, configurations $(\mathcal{L}_3, C_3)$ and $(\mathcal{L}_4, C_4)$ are ordered.

Another set of relations is necessary to describe the data dependences. 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: $Write_{\mathsf{s}}(u, v)$ if running s on u will write to v; $ReadWrite_{\mathsf{s}}(u, v)$ if running s on u will read or write to v. The following predicate describes two configurations $(\mathcal{L}_1, C_1, \mathsf{s}, u)$ and $(\mathcal{L}_2, C_2, \mathsf{t}, v)$ with data dependence if both last records $(\mathsf{s}, u, \dots)$ and $(\mathsf{t}, v, \dots)$ access the same node z and at least one of the accesses is a write:

$$\begin{aligned}
&Dependence_{\mathsf{s},\mathsf{t}}(u, v, \mathcal{L}_1, \mathcal{L}_2, C_1, C_2) \equiv \\
&\quad Configuration(\mathcal{L}_1, C_1, \mathsf{s}, u) \wedge Configuration(\mathcal{L}_2, C_2, \mathsf{t}, v) \\
&\quad \wedge \exists z.\Big(\big(ReadWrite_{\mathsf{s}}(u, z) \wedge Write_{\mathsf{t}}(v, z)\big) \\
&\quad\quad \vee \big(Write_{\mathsf{s}}(u, z) \wedge ReadWrite_{\mathsf{t}}(v, z)\big)\Big)
\end{aligned}$$

Example 4.4. Considering another configuration $(\mathcal{L}_5, C_5)$ with execution path $r \to p \to q$ shown in Figure 9b. The labels in configuration $(\mathcal{L}_5, C_5)$ on nodes r, p are the same with the ones in configuration $(\mathcal{L}_3, C_3)$. Labels $L_{\mathsf{s}1}$ and $L_{\mathsf{s}7}$ are on node q. Thus $Dependence_{\mathsf{s}3,\mathsf{s}5}(w, q, \mathcal{L}_3, \mathcal{L}_5, C_3, C_5)$ is true since $s3$ is writing n.Odd on node w while $s7$ 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:

$$\begin{aligned}
&DataRace[\![P]\!] \equiv \bigvee_{\mathsf{q}_1,\mathsf{q}_2 \in \mathsf{AllNonCalls}} \exists x_1, x_2, \mathcal{L}_1, \mathcal{L}_2, C_1, C_2.\Big(\\
&\quad Dependence_{\mathsf{q}_1,\mathsf{q}_2}(x_1, x_2, \mathcal{L}_1, \mathcal{L}_2, C_1, C_2) \\
&\quad \wedge Parallel(\mathcal{L}_1, \mathcal{L}_2, C_1, C_2)\Big)
\end{aligned}$$

Theorem 4.5. *A Retreet program P is data-race-free if $DataRace[\![P]\!]$ is invalid.*

Proof. If P is not data-race-free, there must exist two iterations, represented as $(\mathcal{L}_1, C_1)$ and $(\mathcal{L}_2, C_2)$ 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 $DataRace[\![P]\!]$, as *Dependence* encodes data dependences and *Parallel* encodes the absence of happens-before. □

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 Sim : $\mathsf{AllBlocks}(P) \to \mathsf{AllBlocks}(P')$ such that

- for any q $\in$ AllNonCalls, q and Sim(q) are identical (modulo variable renaming).
- Sim is a bijective mapping between AllNonCalls(P) and AllNonCalls(P').
- for any call s $\in$ AllCalls(P), s and Sim(s) are calling the same node.
- if s $\triangleleft$ t in P, then Sim(s) $\triangleleft$ Sim(t) in P'.
- if s′ $\triangleleft$ t′ in P' and Sim(t) $= t'$, then there is a unique s such that s $\triangleleft$ t and Sim(s) = s′.
- for any nodes u, v, s, speculative values M, N, and any blocks s′, t, t′ such that $\underline{Sim(\mathsf{s}) = \mathsf{s}'}$ and $Sim(\mathsf{t}) = t'$, the path conditions $\overline{PathCond_{\mathsf{s},\mathsf{t}}}(u, v, M, N)$ and $\overline{PathCond_{\mathsf{s}',\mathsf{t}'}}(u, v, M, N)$ are equivalent.

Intuitively, P and P' bisimulate if any configuration for P can be converted to a corresponding configuration for P', and vice versa. It is not hard to develop a naive bisimulation-checking algorithm to check if two Retreet programs P and P' bisimulate: just enumerate all possible relations between P blocks and P' blocks, by brute force.

The correspondence between configurations can be extended to executions, i.e., every execution of P corresponds to an execution of P' that runs exactly the same blocks of code on the same nodes, and vice versa. To guarantee the equivalence, it suffices to make sure that the correspondence does not swap any pair of ordered configurations with data dependences. [5] In the following formula, the predicates $Dependence^{P}_{\mathsf{q}_1,\mathsf{q}_2}$ and $Dependence^{P'}_{\mathsf{q}_1,\mathsf{q}_2}$ guarantee four configurations, two on P and two on P', and pair-wisely bisimulating (as they end with the same blocks).

$$
\begin{aligned}
&Conflict[\![P, P']\!] \equiv \\
&\bigvee_{\mathsf{q}_1,\mathsf{q}_2 \in \mathsf{AllNonCalls}} \exists x_1, x_2, \mathcal{L}_1, \mathcal{L}_2, C_1, C_2, \mathcal{L}'_1, \mathcal{L}'_2, C'_1, C'_2. \Big(\\
&\;Dependence^{P}_{\mathsf{q}_1,\mathsf{q}_2}(x_1, x_2, \mathcal{L}_1, \mathcal{L}_2, C_1, C_2) \\
&\;\wedge Dependence^{P'}_{\mathsf{q}_1,\mathsf{q}_2}(x_1, x_2, \mathcal{L}'_1, \mathcal{L}'_2, C'_1, C'_2) \\
&\;\wedge\ Ordered^{P}(\mathcal{L}_1, \mathcal{L}_2, C_1, C_2)\ \wedge\ Ordered^{P'}(\mathcal{L}'_2, \mathcal{L}'_1, C'_2, C'_1) \Big)
\end{aligned}
$$

Theorem 4.7. *For any two data-race-free* Retreet *programs P and P' that bisimulate, they are equivalent if $Conflict[\![P, P']\!]$ is invalid.*

Proof. According to Definition 4.6, it can be proved by recursion 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 $Conflict[\![P, P']\!]$ 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. □

[5]We assume both programs are free of data races; otherwise the equivalence between them is undefined.

```
f(n)
  ...
  h = height(n.l)
  s = size(n.l)
  if (h == 5 && s == 3)
    f(n.l)
  ...
```

Figure 11. Example of incompleteness

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 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, cycletree 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 cycletree 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

```
Fused(n)
  if (n == nil) return (0, 0)
  else
    (ls, lv) = Fused(n.l)
    (rs, rv) = Fused(n.r)
    return (ls + rs + 1, lv + rv)
```

(a) A valid fusion

```
Fused(n)
  if (n == nil) return (0, 0)
  else
    (ret1, ret2) = (ls + rs + 1, lv + rv)
    (ls, lv) = Fused(n.l)
    (rs, rv) = Fused(n.r)
    return (ret1, ret2)
```

(b) An invalid fusion

Figure 12. Fusing two mutually recursive traversals

time spent on program construction and encoding is negligible. Remember our MSO encodings of data-race-freeness and equivalence are sound but not complete, the negative answers could be spurious. To this end, whenever Mona returned a counterexample, we manually investigated if it corresponds to a real evidence of violation.

Mutually Recursive Size-Counting. This is our running example presented in Figure 4. We synthesized a fused traversal shown in Figure 12a and verified that the mutually recursive traversals Odd and Even can be fused to the single traversal (solved by Mona in 0.14s). This simple synthesis and verification task, to our knowledge, is already beyond the capability of all existing approaches. We also designed an invalid fused traversal (shown in Figure 12b) and encode the fusibility to MSO. Mona returned a counterexample in 0.14s that illustrates how the data dependence is violated. Basically, the read-after-write dependence between a child and its parent in traversal Even is violated after the fusion. We manually verified that the counterexample is a true positive.

We also checked the data-race-freeness of the original program. The two parallel traversals Odd(n) and Even(n) in the main function are independent because in every layer of the tree there is exactly one Odd call and one Even call and they belong to different traversals on each layer of the tree. The data-race-freeness was checked in 0.02s.

CSS Minification. Cascading Style Sheets (CSS) are a widely-used style sheet language for web pages. In order to lessen the page loading time, many minification techniques are adapted to reduce the size of CSS document so that the time spent on delivering CSS documents can be reduced [2–4, 6, 18]. When minifying the CSS file, the Abstract Syntax Tree (AST) of the CSS code is traversed several times to perform different kinds of minifications, such as shortening 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

```
ConvertValues(n)
  if (n == nil) return 0
  else
    for each child p: ConvertValues(n.p)
    if (n.type == ''word'' || n.type == ''func'')
      n.value = TransValue(n.value)
MinifyFont(n)
  if (n == nil) return 0
  else
    for each child p: MinifyFont(n.p)
    if (n.prop == ''font−weight'')
      n.value = MinifyWeight(n.value)
ReduceInit(n)
  if (n == nil) return 0
  else
    for each child p: ReduceInit(n.p)
    if (length(n.value) < initialLength)
      n.value = ReduceInitial(n.value)
```

```
Main(n)
  ConvertValues(n)
  MinifyFont(n)
  ReduceInit(n)
```

Figure 13. CSS minification traversals

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

```
RootMode(n, number)
  if (n == nil) return
  else
    n.num = number
    number = number+1
    PreMode(n.l, number)
    PostMode(n.r, number)
PreMode(n, number)
  if (n == nil) return
  else
    n.num = number
    number = number+1
    PreMode(n.l, number)
    InMode(n.r, number)
InMode(n, number)
  if (n == nil) return
  else
    PostMode(n.l, number)
    n.num = number
    number = number+1
    PreMode(n.r, number)
PostMode(n, number)
  if (n == nil) return
  else
    InMode(n.l, number)
    PostMode(n.r, number)
    n.num = number
    number = number+1
ComputeRouting(n)
  if (n == nil) return
  else
    ComputeRouting(n.l)
    ComputeRouting(n.r)
    n.lmin = n.l.min
    n.rmin = n.r.min
    n.lmax = n.l.max
    n.rmax = n.r.max
    n.max = MAX(n.lmax, n.rmax, n.num)
    n.min = MIN(n.lmin, n.rmin, n.num)
Main(n)
  RootMode(n, 0)
  ComputeRouting(n)
```

Figure 14. Ordered cycletree construction and routing data computation

communication. Cycletrees are proven to be an efficient network topology in terms of degree and number of communication links [28–30].

We consider two traversals regarding cycletrees. 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 cycletree. Another traversal ComputeRouting computes the router data of each node which are essential for an efficient cycletree 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.

```
Sum(n)
  if (n == nil) return
  else
    Sum(n.next)
    nv = n.next ? 0 : n.next.v
    n.v = nv + n.v
Shift(n)
  if (n == nil) return
  else
    nv = n.next ? 0 : n.next.v
    n.v = nv
    Shift(n.next)
```

(a) Traversals on list

```
Fused(n)
  if (n == nil) return
  else
    nv = n.next ? 0 : n.next.v
    n.v = nv
    Fused(n.next)
    nv = n.next ? 0 : n.next.v
    n.v = nv + n.v
```

(b) Single fused traversal (swapped order)

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:

$$sum(x, y) \equiv x \leq y$$

$$shift(x, y) \equiv x.next = y$$

such that $sum(x, y)$ (resp. $shift(x, y)$) means x belongs to the set $N(y)$ for traversal Sum (resp. Shift). We can further encode similar predicates for "sum then shift" and "shift then sum", respectively:

$$sum\_shift(x, y) \equiv \exists z.shift(x, z) \wedge sum(z, y)$$

$$shift\_sum(x, y) \equiv \exists z.sum(x, z) \wedge shift(z, y)$$

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

$$shift\_sum(x, y) \leftrightarrow sum\_shift(x, y)$$

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], Ghiya et al. [8] detect function calls that access disjoint subtrees 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 cycletrees, 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

[1] Pierre Amiranoff, Albert Cohen, and Paul Feautrier. 2006. Beyond Iteration Vectors: Instancewise Relational Abstract Domains. In *Static Analysis*, Kwangkeun Yi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 161–180.

[2] Johan Bleuzen. 2015. cssmin. https://www.npmjs.com/package/cssmin

[3] Ben Briggs. 2015. cssnano. https://cssnano.co/

[4] Steve Clay. 2007. minify. https://github.com/mrclay/minify

[5] Loris D'Antoni, Margus Veanes, Benjamin Livshits, and David Molnar. 2014. Fast: A Transducer-based Language for Tree Manipulation. *SIGPLAN Not.* 49, 6 (jun 2014), 384–394.

[6] Roman Dvornov. 2011. csso. https://github.com/css/csso

[7] Jacob Elgaard, Nils Klarlund, and Anders Møller. 1998. MONA 1.x: New techniques for WS1S and WS2S. In *Computer Aided Verification*, Alan J. Hu and Moshe Y. Vardi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 516–520.

[8] Rakesh Ghiya, Laurie J. Hendren, and Yingchun Zhu. 1998. Detecting Parallelism in C Programs with Recursive Darta Structures. In *Proceedings of the 7th International Conference on Compiler Construction (CC '98)*. Springer-Verlag, London, UK, UK, 159–173. http://dl.acm.org/citation.cfm?id=647474.727598

[9] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark) *(FPCA '93)*. ACM, New York, NY, USA, 223–232. https://doi.org/10.1145/165180.165214

[10] Matthew Hague, Anthony W. Lin, and Chih-Duo Hong. 2019. CSS Minification via Constraint Solving. *ACM Trans. Program. Lang. Syst.* 41, 2, Article 12 (June 2019), 76 pages. https://doi.org/10.1145/3310337

[11] Youngjoon Jo and Milind Kulkarni. 2011. Enhancing locality for recursive traversals of recursive structures. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (Portland, Oregon, USA) *(OOPSLA '11)*. ACM, New York, NY, USA, 463–482. https://doi.org/10.1145/2048066.2048104

[12] Youngjoon Jo and Milind Kulkarni. 2012. Automatically Enhancing Locality for Tree Traversals with Traversal Splicing. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 355–374. https://doi.org/10.1145/2384616.2384643

[13] Neil D. Jones and Steven S. Muchnick. 1982. A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL '82)*. ACM, New York, NY, USA, 66–74. https://doi.org/10.1145/582153.582161

[14] MóNica MartíNez and Alberto Pardo. 2013. A Shortcut Fusion Approach to Accumulations. *Sci. Comput. Program.* 78, 8 (Aug. 2013), 1121–1136. https://doi.org/10.1016/j.scico.2012.09.002

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

[16] 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* (Shenzhen, China) *(PPoPP '13)*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/2442516.2442535

[17] Marvin L. Minsky. 1967. *Computation: Finite and Infinite Machines.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[18] Jakub Pawlowicz. 2011. clean-css. https://github.com/jakubpawlowicz/clean-css

[19] 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* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 201–216. https://doi.org/10.1145/3062341.3062346

[20] Xiaokang Qiu. 2020. Streaming Transformations of Infinite Ordered-Data Words. arXiv:2001.06952 [cs.FL] https://arxiv.org/abs/2001.06952

[21] S. Rajbhandari, J. Kim, S. Krishnamoorthy, L. Pouchet, F. Rastello, R. J. Harrison, and P. Sadayappan. 2016. A Domain-Specific Compiler for a Parallel Multiresolution Adaptive Numerical Simulation Environment. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Piscataway, NJ, 468–479. https://doi.org/10.1109/SC.2016.39

[22] 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* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/2892208.2892228

[23] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing Data Structures in High-level Programs: New Directions for Extensible Compilers Based on Staging. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. ACM, New York, NY, USA, 497–510. https://doi.org/10.1145/2429069.2429128

[24] Radu Rugina and Martin Rinard. 2000. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. ACM, New York, NY, USA, 182–195. https://doi.org/10.1145/349299.349325

[25] Laith Sakka. 2020. *Techniques for Automatic Fusion of General Tree Traversals.* Ph.D. Dissertation. Purdue University.

[26] 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

[27] Kirshanthan Sundararajah and Milind Kulkarni. 2019. Composable, Sound Transformations of Nested Recursion and Loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 902–917. https://doi.org/10.1145/3314221.3314592

[28] Margus Veanes and Jonas Barklund. 1996. Construction of Natural Cycletrees. *Inf. Process. Lett.* 60, 6 (1996), 313–318. https://doi.org/10.1016/S0020-0190(96)00179-2

[29] Margus Veanes and Jonas Barklund. 1996. Natural Cycletrees: Flexible Interconnection Graphs. *J. Parallel Distrib. Comput.* 33 (02 1996), 44–54. https://doi.org/10.1006/jpdc.1996.0023

[30] Margus Veanes and Jonas Barklund. 1996. On the Number of Edges in Cycletrees. *Inf. Process. Lett.* 57, 4 (1996), 225–229. https://doi.org/10.1016/0020-0190(95)00183-2

[31] 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

[32] 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 (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 314–325. https://doi.org/10.1145/2737924.2737972