4-15-2019

Composable, Sound Transformations of Nested Recursion and Loops

Kirshanthan Sundararajah
Purdue University, ksundar@purdue.edu

Milind Kulkarni
Purdue University - Main Campus, milind@purdue.edu

Composable, Sound Transformations of Nested Recursion and Loops*

Extended version

Kirshanthan Sundararajah
Electrical and Computer Engineering
Purdue University
ksundar@purdue.edu

Milind Kulkarni
Electrical and Computer Engineering
Purdue University
milind@purdue.edu

Abstract

Scheduling transformations reorder a program's operations to improve locality and/or parallelism. The polyhedral model is a general framework for composing and applying *instance-wise* scheduling transformations for loop-based programs, but there is no analogous framework for recursive programs. This paper presents an approach for composing and applying scheduling transformations—like inlining, interchange, and code motion—to *nested recursive programs*. This paper describes the phases of the approach—representing dynamic instances, composing and applying transformations, reasoning about correctness—and shows that these techniques can verify the soundness of composed transformations.

Keywords Dependence Testing, Scheduling Transformations, Locality, Recursion

1 Introduction

There is a large catalog of scheduling transformations for *regular* programs—loop-based programs that operate over arrays and matrices—such as loop tiling, loop interchange, loop fusion, and unrolling [18]. In recent years, many analogous transformations have been developed for *irregular* programs that use recursion to manipulate lists, trees and graphs [14, 15, 25, 26, 29, 33]. As in the regular world, these transformations *restructure* and *reschedule* the operations of a program to enhance locality by moving computations that touch the same pieces of data closer together.

Transformations that reschedule the operations of a program are not necessarily safe. If, for example, operation y reads from a location x writes to, then these operations must be performed in the same order to produce the correct result, and transformations that make y execute before x are unsound. Hence, a transformation like tiling that is safe for one program may not be safe for another. Moreover, while applying one transformation, like tiling, may be unsound for a program, *composing* that transformation with another, like loop reversal, may render the combination of the two transformations safe. Hence, finding effective, safe scheduling transformations requires a framework for reasoning about their composition.

In the world of loops and matrices, frameworks such as the *polyhedral model* [5, 6] tackle this problem through a unified representation of the schedule of computations in a program, the dependences in the program, and transformations of those schedules, allowing compilers to soundly compose and apply loop transformations to programs [3]. However, *no such unifying framework exists* for analogous transformations in the irregular world—different optimizations each use different, *ad hoc* dependence analysis frameworks to drive the transformations [25, 29, 36], when dependence analyses are performed at all, and these disparate frameworks do not allow transformations to be composed.

Contributions

This paper presents the first approach for reasoning about, composing, and checking the soundness of transformations on *nested recursive programs*[1]—the first steps towards an analog of the polyhedral framework for irregular programs. Our approach, which we call PolyRec, incorporates several novel components:

- **A representation of the iteration space** A scheduling framework needs an *instance-wise* representation of a program's operations, representing the dynamic instances of each operation rather than just the static code[2]. PolyRec uses *multitape finite state automata* to represent the instances of statements of a recursive program—each instance is a tuple generated by the automaton—with lexicographic order representing the schedule of computation[3]. (Section 4)
- **A representation of transformations** PolyRec represents scheduling transformations as *multitape finite state transducers*, mapping each instance to another instance, with the new lexicographic order representing

*Extended version of "Composable, Sound Transformations of Nested Recursion and Loops," Sundararajah and Kulkarni, PLDI 2019.

[1] Nested recursion generalizes nested loops, including recursive functions nested within loops [14] or vice versa, or even recursion nested within other recursion [33]. See Section 2.2

[2] Adopting the terminology of Amiranoff et al. [2], we refer to these dynamic instances simply as "instances."

[3] Loops are transformed to tail-recursion to allow PolyRec to treat them uniformly with recursion.

the new schedule. Crucially, this representation is *compositional*, allowing complex transformations to be broken down into a sequence of simple transformations. To demonstrate the utility of PolyRec, we show how several specific transformations from the literature—inlining, interchange [14, 33], code motion [29], and strip-mining—can be represented using PolyRec's multitape transducers. This representation allows these transformations to be arbitrarily combined and composed, generalizing their prior use. (Section 5)

A representation of dependences A dependence analysis framework must represent any dependences in a program in a form that enables a *dependence test*: checking whether a particular transformation violates any dependences. Importantly, this check should apply to composed transformations—an individual transformation may actually *break* dependences, relying on a later transformation to "fix" them; the dependences may only be preserved because of the combination of all transformations. In PolyRec, dependences are represented by *witness tuples* that capture sets of dependent instances. PolyRec's dependence test applies the (composed) transformation transducer to these witness tuples and applies a decision procedure to determine if any dependences are violated—if none are, the transformation is sound. (Section 6)

We build a prototype implementation of PolyRec (Section 7) that can compose transformation transducers for nested recursive programs. When presented with witness tuples that capture the dependences of a source program, the prototype can then check these composed transformations for soundness by applying the dependence test. We also describe a *completion procedure*: starting with a *partial* transformation—an initial transformation that may not be sound—the prototype can be used to search through the (finite) space of transformations consistent with that partial transformation to identify a complete transformation that is sound. Our prototype ultimately verifies that a sequence of transformations is sound with respect to a set of dependences. This sequence of transformations can be applied (e.g., by a traditional compiler) to generate the transformed code.

To evaluate PolyRec, and our prototype implementation, we show that we can automatically create and check fairly sophisticated composed transformations for nested recursive programs, including transformations that are equivalent to combinations of point blocking [14] and traversal splicing [15]. (Section 8)

2 Background

This section provides a brief background on the premise of schedule transformations for iteration constructs (loops and recursion), describes the space of recursion and iteration that PolyRec handles, and gives an overview of recent work on analysis and transformations for recursive programs as well as a sketch of other related work.

```
1  A[N] = /* initialize array */;
2  node T = /* initialize tree */;

4  outer(int i, node n)
5    if (i >= N) return;
6    inner(i, n) //t1
7    outer(i + 1, n) //r1

9  inner(int i, node n)
10   if (n == null) return;
11   inner(i, n.left) //r2^l
12   inner(i, n.right) //r2^r
13   n.x += A[i] //s1

15 main()
16   outer(0, T)
```

(a) Repeatedly traversing a tree.

```
1  outer1(int i, node n, int j)
2    if (i < N)
3      inner(i, n, j)
4      outer1(i + 4, n, j)

6  inner(int i, node n, int j)
7    if (n == null) return;
8    outer2(i, n, j)
9    if (n.left != null)
10     inner(i, n.left.left, j)
11     inner(i, n.left.right, j)
12   if (n.right != null)
13     inner(i, n.right.left, j)
14     inner(i, n.right.right, j)

16 outer2(int i, node n, int j)
17   if (j < 4 && (i + j) < N)
18     if (n != null)
19       n.x += A[i + j]
20       if (n.left != null)
21         n.left.x += A[i+j]
22       if (n.right != null)
23         n.right.x += A[i+j]
24     outer2(i, n, j + 1)

26 main()
27   outer1(0, T, 0)
```

(b) After transforming.

Figure 1. Running example.

2.1 Schedule Transformations

Performing scheduling transformations on code is one of the fundamental ways of improving its performance: changing when an instruction executes can have deep impacts on locality (changing when a memory location is touched can transform a cache miss into a cache hit) and parallelism (moving operations around can increase the number of independent instructions that can be executed simultaneously). Crucially, not all schedules of computation are legal. If statement s_1 accesses a memory location l and statement s_2 accesses that same memory location, with one of those accesses being a write, this *dependence* constrains the possible legal schedules. In *all* legal schedules of computation, s_1 and s_2 must appear in the same order to ensure that they produce the correct result. The dependence must be *preserved*.

Consider the code in Figure 1a, which reflects the structure that arises in many tree-based applications [10]. The function outer represents a loop that iterates from 0 to N that has been transformed into tail recursion (allowing a uniform treatment of loops and recursion), while outer recurses over

some tree structure. The code s_1 (line 13) executes once for each combination of i and n (where n represents a node in the tree). This code has poor locality: while a given element of A is accessed repeatedly while traversing the tree rooted at T, the tree is fully traversed once for each element.

Analyses that look for dependences in statements that access recursive structures (e.g., [8, 12, 20, 28]) will correctly say that there is a dependence from that statement to itself. However, an *instance-wise* analysis of this code reveals more structure in that dependence: s_1 executing at (i, n) specifically has a dependence with $(i + 1, n)$ (the second instance has a different value for the loop induction variable, but executes at the same node in the tree). A program with this dependence structure can be safely transformed using technique akin to point blocking [14] and traversal splicing [15], to give the code in Figure 1b. Here, a 4-element *block* of A traverses the tree simultaneously. At each node of the tree, this block visits the node as well as its immediate children before continuing traversal. In this way, both chunks of A and the tree rooted at T stay in cache, providing better locality.

2.2 Perfect Nesting

This paper focuses on *perfectly-nested* programs, where the control structures are (non-mutual) recursive functions (loops can be treated as recursive functions through the well-known correspondence between loops and tail recursion). Such a recursive function can perform a base-case check to terminate recursion (equivalent to the bounds check in a loop), call itself one or more times, or call a different recursive function (we call this a *transfer* call), "descending" to an inner recursion. The "innermost" recursive function, which only calls itself, can, in addition, perform work that reads and writes different data structures. Figure 1a is perfectly nested: outer simply iterates from 0 to N, while inner performs the actual tree traversal.

Following the lead of Sakka et al. [29], we treat the body of the innermost recursion as a sequence of *compound statements* and recursive calls. These compound statements can contain control flow, but for the purposes of analysis and transformation they are treated as indivisible units[4].

Nested recursion generalizes the case of perfectly nested loops, a frequent target of loop optimization. Moreover, perfectly nested recursion (or perfectly-nested combinations of recursion and loops) arises in a number of domains, as considered by previous transformation frameworks [14, 15, 33, 36]. Note that this space of programs includes any algorithm that performs a *single* recursion, as such a program is perfectly-nested by default.

[4]From an AST perspective, we essentially consider only the top level list of statements in the method body

2.3 Instance-wise Analysis for Recursive Programs

Instance-wise analysis is common for regular programs that deal with nested loop structures that operate over dense arrays [3, 5, 6]. However, when it comes to *irregular* data structures like trees and non-loop control structures like recursion, there has been far less work. Perhaps the most comprehensive treatment of instance-wise analysis for recursive programs comes from Amiranoff et al. [2] (building on prior work by Cohen and Collard [4] and Feautrier [7]).

Amiranoff et al. [2] generate a context-free language representation of a recursive program that uniquely labels each dynamic instance of a statement using a trace string called a *control word*. Using these control words, they can define a dependence analysis that determines the set of dependent (dynamic) instances in a program, using that information to parallelize the program. This work is general in some ways, but has several drawbacks when considered for use in a transformation framework. Their work does not consider how to represent scheduling transformations beyond parallelization (including simple transformations such as code motion or inlining), nor do they consider transforming nested structures.

In recent years, there has been increasing interest in developing *transformation* frameworks for recursive programs. These have ranged from frameworks to support interchange and blocking of nested loops and recursion [14, 15, 36] to frameworks that target *fusing* multiple recursive traversals together [21, 25, 26, 29] to those that transform multiple recursive functions nested inside one another [33]. While some of these only provide informal arguments for correctness, others provide dependence tests that can be used to automatically determine when these transformations break dependences [25, 26, 29, 36]. However, these frameworks are *ad hoc*: they do not provide general, composable ways for reasoning about transformation correctness.

2.4 Other Related Work

There are numerous frameworks that reason about nested loops with affine loop bounds and affine array subscripts [1, 3, 5, 6, 17, 19, 22, 37, 38]. As mentioned above, these approaches focus on dense loops over dense arrays, so are not applicable to our domain. There has been work done in the past to generalize the loop-based model to handle non-affine loop bounds and subscripts using symbolic expressions [23, 34], and to handle sparse matrices and arrays [30–32, 35], but these approaches still only target loops, and hence do not generalize to the recursive constructs we consider.

3 Overview of PolyRec

This section gives a quick overview of PolyRec's representations and mechanisms. Sections 4 through 5 elaborate upon and formalize these facets of PolyRec.

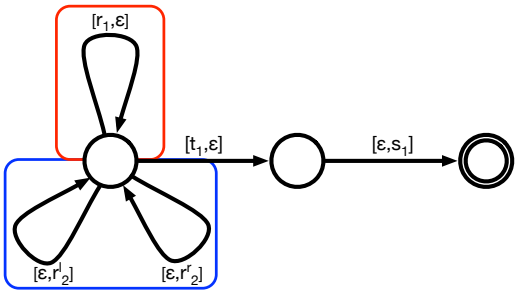


Figure 2. Multitape automaton for Figure 1a.

Running Example Let us return to our example from Figures 1a and 1b. To break down the transformations a little more concretely, first, the method outer was *strip mined* (Section 5.3.4) to break it into two loops, outer1 (at line 1) and outer2 (at line 16); outer2 performs groups of 4 iterations from outer1. Second, the method inner was changed from post-order (as it was in Figure 1a) to pre-order using *code motion* (Section 5.3.1). Then the calls inner(i, n.left, j) and inner(i, n.right, j) were *inlined* to operate on the left and right child before continuing recursion (resulting in three statements in inner and four new recursive calls) (Section 5.3.3) and code motion was applied again. Finally, inner and outer2 were *interchanged* (Section 5.3.2) so that inner iterates over the nodes in the tree before outer2 iterates over its 4-element block of A. PolyRec *can reason about this complex series of transformations, and check it for soundness, in a single representation*, as we explain next.

3.1 Iteration Space Representation

The first task in PolyRec is to capture the *iteration space* of a piece of code. This means finding a way to *name* each dynamic instance of a statement (be it a bounds check or a statement accessing an array or a tree), and capture the ordering relationship between them.

PolyRec uses a *regular relation* representation (i.e., a tuple of strings generated by a multitape finite automaton) for its iteration space. Each statement in a k-deep nest of recursion is named using a an *instance tuple*: a k-string (a k-tuple of strings), with each element in the k-string defining a location in the iteration space for that dimension (read: level of recursion).

Figure 2 shows the multitape automaton that generates the instances for the (pre-transformation) running example. The loop boxed in red represents the "iterations" of outer: each call to outer appends a new r_1 to the first dimension. The loop boxed in blue represents the iterations of inner, two calls, which append either r_2^l or r_2^r to the *second* element. Finally, we represent the two non-recursive statements of the two recursions: a transition that adds t_1 to the first dimension, representing switching to the inner recursion, and a transition that adds s_1 to the second element, representing executing the compound statement in inner. Because this execution is a complete instance, this transition moves to an accept state (i.e., generates an instance tuple).

The instance tuple can be flattened (by concatenating its elements), and alphabetical order then provides the iteration order of the dynamic statement instances. While we could carefully select the alphabets so that alphabetical order would correspond to the correct order, for convenience, PolyRec instead defines a lexicographic order on the alphabet. In our running example, the ordering is $(t_1, r_1, r_2^l, r_2^r, s_1)$. Note that the order of the first two symbols corresponds to the order of the statements in the outer recursion, and the order of the next three symbols corresponds to the order of the inner recursion. Hence, we see that instance $[r_1 r_1 t_1, r_2^l r_2^r s_1]$, which corresponds to the iteration space position i = 2 in the outer dimension and node = root.left.right in the inner dimension, occurs before the instance $[r_1 r_1 t_1, s_1]$ (which executes at the root node of the tree) because the inner recursion is postorder. Note that PolyRec does not attempt to *bound* the iteration space, but merely to order it. This is because most "bounds" in recursive applications are input-dependent and hence not amenable to analysis.

3.2 Transformations

A scheduling transformation preserves *which* instances execute but restructures the iteration space so that they execute in a different order. PolyRec represents a transformation as a *multitape finite state transducer* that rewrite instance tuples (with k elements for k-dimension nests) to other instance tuples (that may have a different number of dimensions). This transducer allows us to translate any instance in the original space to a new instance in the transformed space, and the ordering in this new space (determined by lexicographic order) represents the new schedule of computation.

For example, a *code motion* transformation that changes the order in the three calls and statements in inner execute, can be represented as a rewrite that changes the symbols r_2^l, r_2^r, and s_1 to sort in a different order. Note that this reordering, while seemingly simple, can change a post-order traversal to a pre-order traversal. The new order of $(t_1, r_1, s_1, r_2^l, r_2^r)$ means that $[r_1 r_1 t_1, r_2^l r_2^r s_1]$ now occurs *after* $[r_1 r_1 t_1, s_1]$. More complicatedly, we can implement *interchange*, swapping the inner recursion and the outer recursion by building a transducer that rewrites the call symbols from one dimension of the instance tuple to another.

Because PolyRec's transformations are represented as finite state transducers, they can naturally be composed (multitape FSTs are closed under composition [11]) to produce compound transformations.

3.3 Dependences

When transformation transducers are applied to a PolyRec iteration space, the schedule of instances change. Not all

schedules are valid however: if two dependent instances change their order, the new schedule will produce incorrect results. Thus, PolyRec provides a test for whether dependences are violated by a transformation.

PolyRec represents pairs of dependent instances as a *witness tuple*. This is a 3-tuple of regular relations that captures pairs of dependent instances in three parts: (i) the common prefix of a pair of dependent instances; (ii) the suffix(es) of the first of each pair; and (iii) the suffix(es) of the second of each pair. This tuple, which we write $\langle R_\alpha, (R_\beta, R_\gamma)\rangle$, functions as a generator for instance pairs: each pair can be formed by choosing an element from R_α, then appending an element from R_β to form the first instance and an element from R_γ to form the second instance.

In our running example, there are dependences from any instance that executes at a particular node n of the tree to any *later* instance that executes at the same node n of the tree (but with a different value of i). Hence, the witness tuple for this program is:

$$\langle [(r_1)^*,\ (r_2^l|r_2^r)^*],\ ([t_1,\ s_1],\ [(r_1)^+ t_1,\ s_1])\rangle.$$

So, for example, $[r_1 r_1 t_1, r_2^l r_2^r s_1]$ is dependent on $[r_1 t_1, r_2^l r_2^r s_1]$.

PolyRec provides a decision procedure for checking if a witness tuple is *preserved* by a transformation (i.e., whether all pairs generated by the tuple preserve their order under the transformation), allowing us to check the validity of arbitrary composed transformations. In our running example, the witness tuple is preserved by the proposed transformation. Note, however, that a different transformation that turns outer into a post-order traversal—the analog of loop reversal—by swapping t_1 and r_1 in the symbol ordering would not be sound. It would reverse dependences generated by the witness tuple (including the one given above).

4 Representing Recursive Iteration Spaces

The first step in any scheduling framework is designing a representation to capture the iteration space and schedule. While polyhedral frameworks use representations like systems of linear inequalities to capture the iteration space and a lexicographic ordering on integer points in that space to capture a schedule [3, 5, 6], the story is complicated for recursive programs because the iteration spaces are not affine. Amiranoff et al. [2] use *control words*, derived from a context-free language representation of a program, to name instances of recursive programs. By carefully choosing the alphabet, ordering on the control words is also lexicographic ordering.

Unfortunately, Amiranoff et al. [2]'s work is not quite suitable for our setting. In particular, the control word abstraction for instances creates a single string for each instance, without concern for the distinctions between dimensions. This limitation means that their representation cannot support transformations, such as interchange and strip mining, that focus on particular dimensions or the interaction between dimensions. To overcome this problem, PolyRec uses a novel representation, based on regular relations that are generated by *non-deterministic multitape finite automata* [24].

4.1 Preliminaries

Intuitively, a non-deterministic, multitape finite automaton is akin to a regular NFA that reads over multiple input tapes, rather than one. The transition function, rather than providing transitions between states when observing a single symbol from Σ^* (i.e., transitioning when seeing a symbol, or non-deterministically transitioning on ε) instead transitions based on a *tuple* of symbols drawn from $(\Sigma \cup \varepsilon)^k$ that matches symbols from k tapes.

More formally, let Σ be an alphabet of symbols, with Σ^* representing the set of words and ε denoting the empty word. For two words $w_1, w_2 \in \Sigma^*$, $w_1 \cdot w_2$ is their concatenation. A k-word is a k-tuple from the set $(\Sigma^* \cup \varepsilon)^k$. For two k-words, $v = [v_1, v_2, \ldots, v_k]$ and $w = [w_1, w_2, \ldots, w_k]$, their elementwise concatenation, $v \odot w$ is $[v_1 \cdot w_1, v_2 \cdot w_2, \ldots, v_k \cdot w_k]$.

We can thus define:

Definition 4.1. A *non-deterministic, k-tape finite automaton* is a 6-tuple $A = \langle k, \Sigma, Q, q_0, F, E\rangle$ where:

- k is the number of tapes
- Σ is the finite alphabet
- Q is a finite set of states
- $q_0 \in Q$ is the *start* state
- $F \subseteq Q$ is a set of *accept* states
- $E \subseteq Q \times (\Sigma \cup \varepsilon)^k \times Q$ is a finite set of labeled transitions (each labeled with a k-tuple of symbols and/or ε)

A recognizes the k-word $v \in (\Sigma^* \cup \varepsilon)^k$ iff there exists a path $q_0 a_1 q_1 a_2 q_2 \ldots a_n q_n$ where q_0 is the initial state, $q_n \in F$, for each $0 < i \leq n$, $\langle q_{i-1}, a_i, q_i\rangle \in E$, and $v = a_1 \odot a_2 \odot \cdots \odot a_n$

The relation $R(A) \subseteq (\Sigma^* \cup \varepsilon)^k$ is the set of k-words recognized by A, and is a *regular relation*; all regular relations have multitape automata that recognize them [16].

The class of regular relations is closed under concatenation, union, and Cartesian product. Regular relations are also closed under *projection*. Suppose R is a k-dimension relation, and $0 < i \leq k$:

$$R \ominus i := \{[w_1, \ldots, w_{i-1}, w_{i+1}, \ldots, w_k] \ \mid$$

$$\exists w : [w_1, \ldots, w_{i-1}, w, w_{i+1}, \ldots, w_k] \in R\}$$

In other words, we can "remove" a dimension from a regular relation. This can extended to projecting out multiple dimensions, $R \ominus I$, in the obvious way. Another important closure property for regular relations is *composition* [11]:

$$R_1 \circ R_2 := \{[w_1, \ldots, w_{k+l-2}] \ \mid$$

$$\exists w : [w_1, \ldots, w_{k-1}, w] \in R_1, [w, w_k, \ldots, w_{k+l-2}] \in R_2\}$$

which allows us to "match up" words from two relations to form a new relation. Note that composition itself composes: we can repeat composition to match up arbitrary dimensions from two regular relations to create a new regular relation.

4.2 Capturing Instances with a Multitape Automaton

PolyRec uniquely names and orders instances of statements in recursive programs using k-tuples of symbols generated by a k-tape automaton, where k is the number of recursion dimensions. Each statement that executes does so at a unique combination of call stack and static statement location (since we represent loops with recursion). This information is sufficient to uniquely name each dynamic instance. We can readily construct a multi-tape finite automaton $\mathcal{A}$ that enumerates a k-tuple of strings representing every possible call stack and static statement for a program. Let $|S|$ be the number of compound statements in the innermost recursion. Let k be the number of recursive methods in the program. Let $|C_i|$ be the number of recursive calls in recursive function r_i.

$\mathcal{A}$ is a 6-tuple, $\langle k, \Sigma, Q, q_0, F, E\rangle$, defined as follows:

- k is simply the number of dimensions of the loop nest.
- Σ is the union of the following set of symbols:
  - $\{s_i | 0 < i \leq |S|\}$. One symbol per compound statement in the program, with the ith compound statement getting the symbol s_i
  - $\{t_i | 0 < i < k\}$. One symbol per transfer call in the program (note that there are $k-1$ total such calls).
  - $\{r_i^j | 0 < i \leq k \wedge 0 < j \leq |C_i|\}$. One symbol per recursive call in the program, with the jth recursive call made by the ith recursive function labeled r_i^j.
- Q has $k + |s|$ states: $\{q_0, q_1, \ldots q_{k-1}\} \cup \{q_{s_i} | 0 < i \leq |S|\}$ The first set of states are associated with the recursion levels, while the second set of states are associated with the compound statements.
- q_0, the start state, is, simply, q_0.
- F is $\{q_{s_i} | 0 < i \leq |S|\}$. In other words, every state associated with a compound statement is an accept state.
- E includes the following edges. For notational convenience, we will let the transition label $\ell_i[x]$ be a k-tuple with ε in every dimension *except* dimension i, which has the value x.[5]
  - $\{\langle q_0, \ell_i[r_i^j], q_0\rangle | 0 < i \leq k \wedge 0 < j \leq |C_i|\}$. In other words, a self loop with the label $\ell_i[r_i^j]$ for the jth recursive call from the ith level of recursion. Note that this label is only non-ε in dimension i.
  - $\{\langle q_{i-1}, \ell_i[t_i], q_i\rangle | 0 < i < k\}$. In other words, a sequence of edges from $q_0 \ldots q_{k-1}$ labeled with the transfer calls. Each call only adds a symbol to the dimension of its recursion level.
  - $\{\langle q_{k-1}, \ell_k[s_i], q_{s_i}\rangle | 0 < i \leq |S|\}$. In other words, a transition from q_{k-1} to the state associated with each compound statement, labeled with that compound statement's symbol.

[5]For example, in a two-dimensional nest, $\ell_1[r_1^1] = [r_1^1, \varepsilon]$, while $\ell_2[s_1] = [\varepsilon, s_1]$.

This construction procedure produces an automaton that produces the regular relation $R_{\mathcal{A}}$. Each dimension of the relation corresponds to one recursion level. The set of strings generated by each dimension is a sequence of recursive calls (that stay at that recursion level) followed by a transfer call that signals the end of that recursion level. The set of strings generated by the innermost recursion, at the last dimension, is a sequence of recursive calls followed by a single compound statement. Note that the "flattened" language of $R_{\mathcal{A}}$, which we will call $\mathcal{L}_{\mathcal{A}}$ corresponds to strings that represent all possible dynamic statement instances as a call stack (with the "transfer" recursive call that begins the execution of a level of recursion distinguished from recursive calls that stay at that level of recursion) plus the static statement in the innermost recursion. If the symbols of Σ are ordered by the order the calls and statements appear in each recursive function, it is also clear that the lexicographic ordering of the strings in $\mathcal{L}_{\mathcal{A}}$ corresponds to the order in which their respective instances would execute.

This automaton does not consider bounds checks in any way: it generates an infinite relation. Nevertheless, any real execution of the program, which requires that all recursive functions terminate, will generate a finite subset of $R_{\mathcal{A}}$, whose flattened, ordered set of strings can be embedded in the (infinite) ordered sequence of the strings of $\mathcal{L}_{\mathcal{A}}$. Hence, $R_{\mathcal{A}}$ is a sound overapproximation of the set of dynamic instances of a program (and their order).

Remark. *The language $\mathcal{L}_{\mathcal{A}}$ we can derive from flattening $R_{\mathcal{A}}$ is not particularly different from the language of control words defined by Amiranoff et al. [2]. They note that while they use context-free languages to generate their control words, for their programs, the languages are actually regular.*

The automata PolyRec considers share a common structure: a series of "loops" capturing the recursive calls at state q_0, then a sequence of transfer transitions representing the end of each recursion dimension, followed by a set of final states representing the compound statements in the innermost recursion.

5 Representing Scheduling Transformations

Armed with a representation for the iteration space, described in Section 4, the next step for PolyRec is to provide a representation for *scheduling transformations* of the iteration space. A scheduling transformation provides a new order of execution for instances, and can be used to, for example, improve locality.

PolyRec uses *multitape finite state transducers* [16, 24] to represent its transformations (Section 5.1). Transducers are naturally composable, meaning that PolyRec can synthesize compound transformations that apply multiple rewrites to a recursive loop nest (Section 5.2). Finally, we describe techniques for generating transducers that apply specific

transformations to nested recursion: code motion, inlining, strip mining, and interchange (Section 5.3).

5.1 Transformations as Multitape Transducers

A scheduling transformation is a bijective function that maps instances in one iteration space to instances in a different, transformed iteration space. Of course, not all such functions are useful to consider as transformations. We would like scheduling functions to meet the following criteria:

1. The co-domain of the scheduling function should also be a regular relation of strings. This means that PolyRec can keep iteration spaces in the world of regular relations and reason about schedules using flattening and lexicographic ordering —the universe of PolyRec iteration spaces will be closed under scheduling transformations.
2. Scheduling transformations should be easily *composable* —it should be possible to combine multiple transformation functions to produce a composite function that transforms an input schedule to an output schedule.
3. Scheduling transformations should preserve perfect nesting: if an iteration space is perfectly nested, applying the transformation should result in a new perfectly nested space.[6] Note that this means that the transformation should not alter the general structure of the iteration space automaton—a series of self loops at q_0 representing all the recursive calls in different dimensions, a sequence of transfer transitions, and a set of final states representing the computations of the innermost recursion.

To satisfy the first two properties, PolyRec uses *multitape, non-deterministic, finite-state transducers* to represent scheduling transformations. These multi-tape automata act as string rewriters, and can rewrite the (multi-dimensional) strings representing one iteration space into (multi-dimensional) strings from another iteration space.

Consider an input iteration space with k dimensions over symbols (calls and statements) Σ, $R_{\mathcal{L}} \in (\Sigma \cup \varepsilon)^k$. A transformation that reschedules it to k dimensions with new symbols, Σ' would be a transducer $T \in (\Sigma \cup \varepsilon)^k \times (\Sigma' \cup \varepsilon)^k$. We can use T as a function $T : (\Sigma \cup \varepsilon)^k \rightarrow (\Sigma' \cup \varepsilon)^k$ defined in the obvious way, provided that T's first k dimensions are a superset of $R_{\mathcal{L}}$ (making T total) and that each tuple in $R_{\mathcal{L}}$ only "matches" a single tuple in T's second k dimensions (making T a function).

Remark. *The properties required for T to be used as a function are, in general, undecidable for multitape finite automata [9]. However, we build our transformations in a constructive manner that makes it straightforward to see that T acts as expected.*

[6] Not all scheduling functions that meet the first two requirements meet this third one (indeed, well-understood transformations like fission break this requirement). But PolyRec currently only handles perfect nesting.

We can think of an edge label in a transformation transducer T as having the following form (we show a 2-dimension to 2-dimension transformation, but this generalizes in the obvious way to multiple dimensions):

$$[i_1, i_2] \rightarrow [j_1, j_2]$$

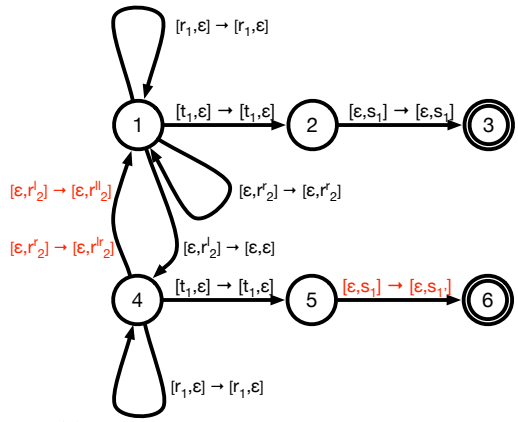
This transition rewrites the symbols $[i_1, i_2]$ from the input tape into $[j_1, j_2]$ on the output tape.

Remark. *Note that, as with many polyhedral frameworks,* PolyRec *targets rescheduling transformations. In particular, because T is a bijection, computations are only reordered, not created or removed. This is in contrast to transformations that eliminate redundant or unnecessary operations.* PolyRec *does not handle such transformations.*

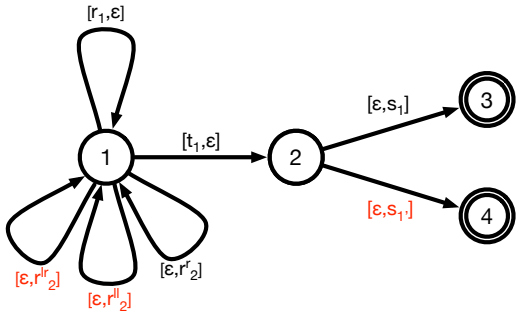
To help satisfy the third property of transformation transducers, and to admit a decidable dependence test (Section 6), we place an additional structural restriction on transformations: they should be *order-free* rewrites. In short, there are two restrictions: (i) any state in the transducer that has an input transition accepting a recursive symbol (on any tape) must also have transitions that accept *all other* recursive symbols[7]; and (ii) any such state must have a transition to a *tail*: a sequence of states that accepts the transition call symbols one after another, followed by states that accept any of the innermost compound statements. These conditions combined essentially mean that the transformation should be able to rewrite the recursive calls in any order it wants; consuming a recursive call off of one input tape cannot preclude either consuming recursive calls on other tapes or ending the rewrite by entering the tail. Figure 3a gives an example of what an order-free rewrite looks like. Note that both "looping" states have rewrites for $[r_1, \varepsilon]$.

Applying a transformation transducer is straightforward: we simply project out the output dimensions of the transducer, and apply ε elimination to simplify the resulting multitape automaton. The result is an automaton that produces the *transformed* iteration space. Note, though, that it is important that the transformation be implemented as a transducer, rather than simply giving the transformed iteration space. We are not just interested in the final schedule of computation, but *how we got there*: which specific instances in the original schedule got mapped to which specific instances in the transformed schedule. It is this information that allows us to check the soundness of schedules (Section 6). This also means that any transformation transducer must not only rewrite one iteration space to another, but do so *faithfully*—it should correspond to the way instances in the original program are mapped to instances in the transformed program. The transformations we present in Section 5.3 all do this translation faithfully, although in general there is no way for us to automatically verify this for an arbitrary transformation.

[7] Either directly or through an ε transition.



(a) Transducer to inline of inner(i, n.left).



(b) Projected the output tapes of Figure 3a.

Figure 3. Inlining using transducers.

Example Figure 3a shows a transformation transducer that inlines the call inner(i, n.left) from Figure 1a. We can see how, e.g., $[t_1, r_2^l r_2^r s_1]$ is rewritten to $[t_1, r_2^{lr} s_1]$, as captured by the transformed iteration space (Figure 3b). (Section 5.3.3 explains how this transducer would get constructed)

5.2 Composing Transformations

Our process for composing transformations is straightforward. To apply one transformation, we take the input automaton and construct the necessary transducer that applies the transformation (see Section 5.3 for specific examples of these constructions). We project out the output tapes of the transducer to generate a new iteration space automaton, as described in the previous section.

Because the transformation transducer preserves perfect nesting, this new iteration space automaton looks, from the perspective of PolyRec, no different than a valid iteration space automaton generated from a piece of input source code. So we can simply repeat this process of generating a new transducer and applying it to compose transformations.

Note that while the transformed iteration space automaton captures the new schedule of computation, by itself, it loses the mapping of the original iteration space to the transformed space. Without this mapping, it is not possible to tell if dependences are violated: we cannot tell if any dependences in the original space are "flipped" in the transformed space. Thus, while the transformations are being applied, the original sequence of transducers that represent the transformations are saved, and composed using multitape automaton composition (see Section 4.1). Hence, after a sequence of transformations are applied PolyRec has a transducer that maps the input automaton (the original code) to the output automaton (the fully transformed code). As we will see in Section 6, this is sufficient information for PolyRec to test whether dependences are violated.

5.3 Representing Specific Transformations

We now describe techniques to generate transformation transducers for four well-defined transformations: code motion, recursion interchange, inlining, and strip mining. These component transformations are simple, but their composition is powerful enough to construct transformations such as *point blocking* [14] (a combination of strip mining and interchange) and *traversal splicing* [15] (a combination of all four transformations) when combined in the right way. For each transformation, we describe the basic change the transformation makes to the iteration space, and provide a procedure for constructing a transducer in PolyRec that implements the transformation.

5.3.1 Code motion

As its name would suggest, *code motion* simply reorders the statements in the code around. In non-innermost recursions, this changes the order of the recursive calls and the transfer calls, while in the innermost recursion, this changes the order of the recursive calls and leaf compound statements. Note that because code motion applies within each dimension, it does not change the recursive structure of the iteration space. Nevertheless, code motion reorders the execution schedule, so can break dependences.

The transducer representing code motion looks exactly the same as the multitape automaton representing the iteration space with output tuples same as the input ones at every edge of transition, with one-to-one replacements of symbols in the input alphabet with symbols in the output alphabet; the output alphabet merely has a different lexicographical order than the input one. Section 3.2 gave an example of how code motion might be used to change a post-order recursion to a pre-order recursion.

5.3.2 Interchange

In our example, transition $[r_1, \varepsilon] \rightarrow [\varepsilon, r_2]$ switches r_1 from dimension one to two, and transitions $[\varepsilon, r_2^l] \rightarrow [r_1^l, \varepsilon]$ and $[\varepsilon, r_2^r] \rightarrow [r_1^r, \varepsilon]$ switch r_2^r and r_2^l from dimension two to one.

Interchange is a seemingly complex transformation — changing the nesting order of recursion— with a simple transducer. As a code transformation, interchange is well-understood for loops [1], while interchange of loops and

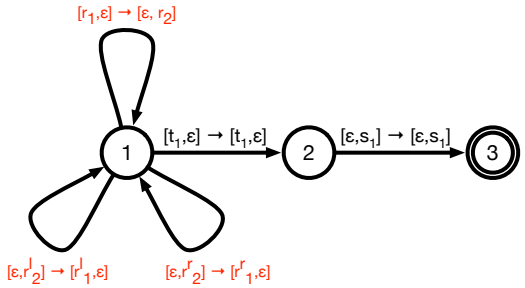


Figure 4. Transducer implementing interchange of first and second dimensions.

more general recursion [14] and general recursive methods [33] has been studied in the literature. Figure 4 shows the transducer for interchanging first and second dimensions of the code in Figure 1a. To interchange dimensions i and j, the transducer begins with a single state with a set of self-transitions that (a) rewrite every recursive call in dimension i to dimension j, and vice versa; and (b) leave recursive calls in other dimensions in their original dimension. We then add a tail of transitions that leave all transfer and compound statements alone. Like code motion, interchange changes the schedule of computation, so can break dependences.

5.3.3 Inlining

Inlining is a standard compiler transformation that nevertheless forms an integral part of more sophisticated transformations that we want to apply to recursive programs, such as traversal splicing [15]. While applying the transformation directly to code is straightforward, it is slightly more tricky to construct the transducer that captures this transformation (an example of which is in Figure 3a).

Our construction for inlining only applies to the innermost dimension, due to the perfect nesting requirement (though note that other dimensions can be essentially inlined by composing interchange, then inlining, then interchange again). For the innermost dimension, we specify which recursive call c should be inlined. We then duplicate *all* the states in the iteration space automaton, creating a new recursive state and a new set of tail states. For the tail states, we leave identity rewrites for all the transfer statements, but for transitions for compound statements s_4 produce rewrites that map them to new copies of those compound statements s_i' (representing the code that was inlined). For the recursive state, we keep self-loops with identity rewrites for non-innermost recursive calls. We then add the following transitions between the original recursive state q_r and the new recursive state $q_{r'}$:

1. A transition from q_r to $q_{r'}$ that rewrites the inlined call c to ε: $[\varepsilon, r_2^l] \longrightarrow [\varepsilon, \varepsilon]$ in our example.
2. For each innermost recursive call in the original dimension, add a transition from $q_{r'}$ *back to* q_r that rewrites that call into a *new* recursive call (representing the inlined versions of those calls). In our example, that creates new recursive calls r_2^{ll} and r_2^{lr}.

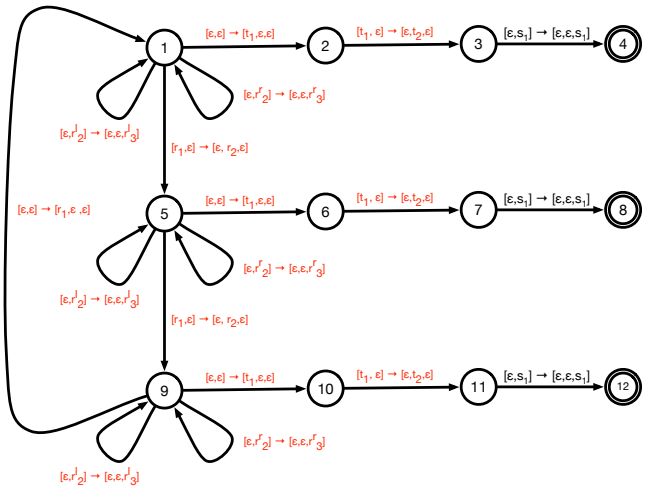


Figure 5. Transducer implementing strip size of two on first dimension.

Note that, as expected, inlining increases the number of calls and compound statements in the innermost recursive call. Unlike code motion and interchange, inlining on its own does not change the schedule of execution—but instances that used to be labeled with strings of the form $(r_l|r_r)^*s_i$ may now be labeled with strings of the form $(r_{ll}|r_{rl}|r_r)^*(s_i|s_i')$.

5.3.4 Strip Mining

While inlining as implemented in PolyRec only applies to innermost recursions, we present another inlining-like transformation that can apply to any recursion that is *loop-like* (i.e., it only has a single recursive call—though that call may be pre-order or post-order): *strip mining*. Strip mining is a well-known transformation for loops that is a precursor to loop tiling [38]; it also appears in transformations for recursive codes [14, 15]. We show how PolyRec can represent the transformation as a transducer.

Conceptually, strip mining looks like inlining a single recursive call multiple times, then collecting those calls into a *new* recursive function in a new dimension of recursion. A general procedure for constructing strip mining transducer is parameterized over the *strip size*—the number of times the original recursion gets inlined. The following steps are taken to construct the transducer in Figure 5:

Replicate states and transitions We replicate all the states and transitions as many times as strip size except the dimension that gets split.

Shift dimensions We shift all transitions corresponding to the dimensions occurring after the dimension that gets split to one lower In this example, we shift dimensions using transitions like $[\varepsilon, r_2^r] \longrightarrow [\varepsilon, \varepsilon, r_3^r]$, $[t_1, \varepsilon] \longrightarrow [\varepsilon, t_2, \varepsilon]$ and $[\varepsilon, s_1] \longrightarrow [\varepsilon, \varepsilon, s_1]$.

Add transfer statements We need a new transfer statement for the newly constructed dimension; we add the transition $[\varepsilon, \varepsilon] \rightarrow [t_1, \varepsilon, \varepsilon]$ to every replica.

Add entering and exiting transitions We add transitions that switch between the original set of states and the replicas. These transitions rewrite the recursive call from the strip-mined dimension to one dimension deeper: $[r_1, \varepsilon] \rightarrow [\varepsilon, r_2, \varepsilon]$. Finally, we add the transition $[\varepsilon, \varepsilon] \rightarrow [r_1, \varepsilon, \varepsilon]$ from the last replica to the first.

Note that the transducer does not directly capture the size of the strip in the "inner" part of the strip-mined loop—PolyRec does not consider loop bounds, leaving that to the transformed code itself. Like inlining, strip mining does not change the relative ordering of instances; but instances that used to be labeled with k dimensions will now be labeled with $k + 1$ dimensions.

6 Representing Dependences and Checking Soundness

Transformations applied by PolyRec in the manner of the previous section are not necessarily *sound*: there is no guarantee that the final transformed program produces the same result as the original program. As an example, consider the double recursive example from Figure 1a. Changing the outer recursion from a pre-order traversal to a post-order traversal can be easily implemented by a code motion transformation that swaps lines 6 and 7. However, doing so means that the updates to each tree node's n.x field in line 13 will occur in the opposite order of the original program, potentially changing the result of the program (e.g., if the addition in line 13 were floating point). Just because PolyRec can synthesize a transformation does not mean that it is legal.

Any transformation *must respect all dependences* in the program. Two instances, i and j, where i executes before j in a schedule have a dependence if j is data dependent on i—they both access the same memory location m, and at least one of i or j writes to m. Give a set of instances I, a schedule of those instances S_I that totally orders I, and a set of dependences $D \subseteq I \times I$, a transformation that produces the schedule S'_I is sound if and only if all pairs in D appear in the same order in S'_I as in S_I. So how can we tell whether a given transformation breaks dependences?

6.1 Witness Tuples to Represent Dependences

The first step in reasoning about dependences is representing the set of dependences in a program. Recall that we want to think about dependences as a set of pairs of instances. Because PolyRec does not reason about the bounds of programs, the set of instances it considers is infinite, as is, potentially, the set of dependence pairs. This representation problem arises in other instance-wise analyses. In the world of loops and matrices, dependences are represented with abstractions ranging from distance and direction vectors [18] to dependence polytopes [3]. In the world of irregular programs, Amiranoff et al. [2] uses a series of *dependence transducers* to represent and reason about dependences in irregular programs. PolyRec representation of dependences is called *witness tuples*, a rough analog of distance vectors.

Definition 6.1. A *witness tuple* is a 3-tuple of regular relations over the alphabet $\Sigma \cup \varepsilon$ of symbols in a given iteration space $R_{\mathcal{A}}$, written $\langle R_\alpha,\ (R_\beta,\ R_\gamma)\rangle$ such that:

1. $R_\alpha \odot R_\beta \subseteq R_{\mathcal{A}}$ and $R_\alpha \odot R_\gamma \subseteq R_{\mathcal{A}}$ (i.e., R_α generates prefixes of instances that, when suffixed with members of R_β or R_γ, are instances of the iteration space).
2. R_α's individual elements are either ε or of the form $(r_i^1 | r_i^2 | \dots)^*$—a sequence of recursive calls from a given dimension. (Note that the prefix contains no transfer statements t_i or leaf compound statements s_i.)
3. If $a \in R_\alpha$, then $\forall b \in (a \odot R_\beta), c \in (a \odot R_\gamma).b \prec c$. In other words, for all instances generated by concatenating a prefix a from R_α with suffixes from R_β and R_γ, the instances generated from $a \odot R_\beta$ lexicographically precede those generated from $a \odot R_\gamma$—they occur earlier in the schedule

Essentially, a witness tuple acts as a generator for pairs of instances with a common prefix that arise in a specific lexicographic order.

Definition 6.2. A witness tuple $\langle R_\alpha,\ (R_\beta,\ R_\gamma)\rangle$ *captures* a set of dependences $D \subseteq R_{\mathcal{A}} \times R_{\mathcal{A}}$ (pairs of instances from $R_{\mathcal{A}}$) if: $\forall (x, y) \in D. \exists a \in R_\alpha, b \in R_\beta, c \in R_\gamma . x = a \odot b \wedge y = a \odot c$. In other words, if the witness tuple can generate all dependence pairs in D.

We can generalize this notion of captures to *sets* of witness tuples if, for any dependence pair in D, at least one witness tuple generates the pair.

We note two things. First, a given set of static statements that have dynamic instances that are dependent on one another may require multiple witness tuples to capture the dependences, since a given witness tuple requires that all generated pairs have the same lexicographic order. Second, any set of dependences can be (conservatively) captured by one or more witness tuples. This is because we can always generate degenerate witness tuples that capture all pairs of instances in an iteration space.

Note that generating witness tuples is not the focus of this paper, as PolyRec is agnostic to *where* the witness tuples come from. Section 7.3 sketches how *tree dependence analysis* [36] could be adapted to build witness tuples, but many prior works present other analyses that could be modified to produce witness tuples [2, 26, 29].

6.2 Checking Soundness

The witness tuples for a program can be used to test transformation soundness. The basic approach is simple: we generate

a dependence pair, then push each instance tuple through the transducer representing the composed program ($\mathcal{A}_T$ producing regular relation R_T). If the transformed instances are still in the same lexicographic order, the dependence is preserved. The primary question is to determine how to test a possibly infinite set of dependences.

Our dependence test process proceeds as follows. For a given witness tuple $\langle R_\alpha,\ (R_\beta,\ R_\gamma)\rangle$, we generate a *single* k-string w from R_α. Recall that by the definition of the witness tuple, $\forall b \in w \odot R_\beta, c \in w \odot R_\gamma . b \prec c$. If we can determine whether $\forall b' \in (w \odot R_\beta) \circ R_T, c' \in (w \odot R_\gamma) \circ R_T . b' \prec c'$, then we will know that for the prefix w from the witness tuple, all dependences are preserved by the transformation. This is decidable as follows.

Dependence test for a given prefix w***:*** We run $\mathcal{A}_T =< k, \Sigma, Q, q_0, F, E >$ with w as its input (i.e., we trace all paths through $\mathcal{A}_T$ from the start state q_0 that accept w), arriving in a set of states $Q_w \subseteq Q$. With Q_w, we construct a derived automaton $\mathcal{A}_T^w =< k, \Sigma, Q', q_0', F', E' >$ as follows:

1. k and Σ are the same as in $\mathcal{A}_T$
2. $Q' = Q \cup q_0'$ (i.e., Q' is the set of states from $\mathcal{A}_T$ plus a fresh state q_0', which is the new start state.)
3. $F' = F$ (i.e., $\mathcal{A}_T^w$ has the same final states as $\mathcal{A}_T$)
4. $E' = E \cup \{\langle q_0', \varepsilon^k, q_i\rangle | q_i \in Q_w\}$. In other words, we add a null transition from the new start state to all the states of $\mathcal{A}_T$ we arrived at after reading in w.

Intuitively, the automaton $\mathcal{A}_T^w$ captures the effect of restarting $\mathcal{A}_T$ after executing w through it. In particular, note that $\mathcal{A}_T^w$ accepts some k-string x iff $\mathcal{A}_T$ accepts $w \odot x$. Moreover, because the transformations all act as functions from instances to instances, the *output* of running x through $\mathcal{A}_T^w$ is what $\mathcal{A}_T$ generates by running x after running w.

Remark. *Note that these two properties are not generally true for non-deterministic multitape finite automata. However, because our transformation transducers have the* order-free *structural property (Section 5) and the prefixes w are only combinations of recursive calls, these properties hold.*

We use $\mathcal{A}_T^w$ as follows. We derive two new regular relations, $R_{w,\beta}$ and $R_{w,\gamma}$ by composing R_β and R_γ respectively with the relation from $\mathcal{A}_T^w$. Note that these relations are the equivalent of running $w \odot R_\beta$ and $w \odot R_\gamma$ through the original transformation transducer $\mathcal{A}_T$.

For $R_{w,\beta}$, we create the lexicographically *latest* k-string by tracing along the first tape. At each state, we trace (keep) all paths with ε transitions on the first tape. From this set of states, we then trace (keep) the transitions with the lexicographically latest symbol and remove the remaining transitions. We continue this process until we have visited or disconnected all the states. This new reduced machine either reaches a final state, in which case it matches some finite string α on the first tape, or it loops, in which case it consumes some infinite string $\alpha \cdot \beta^\infty$ on the first tape. In this reduced machine, there may be multiple paths that match the same string on the first tape. We can repeat this procedure on the reduced machine using the *second* tape—this has the effect of breaking "ties" by considering the behavior of the second dimension in generating the latest possible k-string. Repeating this process for all tapes leads to a machine that generates the latest possible k-string.

For $R_{w,\gamma}$, we create the lexicographically *earliest* k-string in a similar fashion, looking for the earliest string on the first tape, with the added condition that we disconnect all outgoing transitions from final states (since extending the string past a final state can only create a lexicographically later string). Again, this machine will either match some finite string α on the first tape or loop, consuming some infinite string $\alpha \cdot \beta^\infty$. Repeating the process for all the tapes to break ties produces the earliest possible k string.

It is straightforward, then, to compare the k-strings generated in this manner to determine whether the latest k-string from $R_{w,\beta}$ precedes the earliest k-string from $R_{w,\gamma}$, preserving the dependences.

Extending to all possible prefixes: Thus far, we have only shown that the dependence is preserved for *one* prefix w from the witness tuple. How can we show that this holds for all possible w?

First, note that trivially, there are only a finite number of possible sets of states, Q_w, that can be used to construct $\mathcal{A}_T^w$ (since $\mathcal{A}_T$ has a finite number of states). But, more interestingly, we can *find* all possible Q_ws by enumerating the possible ws.

After finding Q_w and running the single-prefix dependence test, we can *extend* w by adding a single symbol to one of its dimensions (assume, without loss of generality, that it is the first tape) and fixing the values of the other dimensions, producing w_1. We can then repeat the single-prefix dependence test for w_1, generating Q_{w_1}, $\mathcal{A}_T^{w_1}$, and so forth. If w_1 is sound, we can then extend w_1 by a single symbol along the same tape, generating w_2, and so on. Note three things: (i) eventually, by the pigeonhole principle, some w_k will generate a Q_{w_k} that is the same as some previously-generated Q_{w_j}; (ii) because Q_{w_k} and Q_{w_j} are the same, the dependence test for w_k passes if and only if the dependence test for w_j passes, hence we do not need to test w_k; and (ii) once this happens, any further extension of w_k along the same dimension can only repeat the behavior of previously-tested prefixes. Thus, we only need to test a finite number of ws to determine that *all* ws with the given values in dimensions $2 \ldots d$ are sound.

From here, a straightforward diagonalization argument shows that we can incrementally extend w along *all* dimensions to eventually discover all the possible Q_w state sets and test them all for soundness. Hence, testing the soundness of all prefixes w is decidable.

Proof Sketch of Soundness The soundness of this dependence test is straightforward. First, because each individual

transducer that goes into the composition correctly maps the input iteration space to the output iteration space for that transformation, the composition correctly changes the schedule of the input program to the schedule of the output program. Hence, given any pair of instances i_1 and i_2, with $i_1 \prec i_2$, running them through the composed transducer will reveal if their order is preserved in the transformed schedule.

Second, the decision procedure we provide for witness tuples determines that for all w_α that arrive at Q_w (the set of states in the composed transducer), $\forall w_\beta \in \beta, w_\gamma \in \gamma . w_\alpha \odot w_\beta \prec w_\alpha \odot w_\gamma$. Because there is only a finite set of Q_w configurations, we can check all these configurations to determine that $\forall w_\alpha \in \alpha, w_\beta \in \beta, w_\gamma \in \gamma . w_\alpha \odot w_\beta \prec w_\alpha \odot w_\gamma$

As long as the set of witness tuples $\mathcal{D}$ covers the set of dependences D (i.e., there is not a dependence $(i_1, i_2) \in D$ that cannot be generated by at least one witness tuple in $\mathcal{D}$), we have that if each witness tuple is preserved, all dependences must be preserved.

Remark. *Note that if $\mathcal{D}$ covers all pairs of instances in the iteration space, our dependence test reduces to checking that* no *pairs of instances are reordered. This is, in fact, true for transformations like inlining or strip mining.*

Examples Recall from Section 3 that the witness tuple for our running example is:

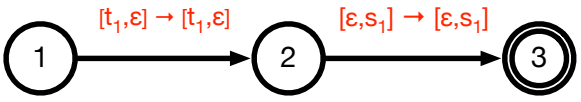
$$\langle [(r_1)^*,\ (r_2^l|r_2^r)^*],\ ([t_1,\ s_1],\ [(r_1)^+ t_1,\ s_1]) \rangle.$$

We will use this witness tuple to check whether the interchange transformation from Figure 4 is sound. There is only one possible state in the transformation transducer that a prefix drawn from $[(r_1)^*, (r_2^l|r_2^r)^*]$ can end in, $\{1\}$, yielding a single possible partially-run automaton, $\mathcal{A}_T^w$, that has $\{1\}$ as its start state. Note that in other cases such as inlining or strip-mining, different prefixes can end in different states. We will later consider an inlining transformation, which has two possible $\mathcal{A}_T^w$s.

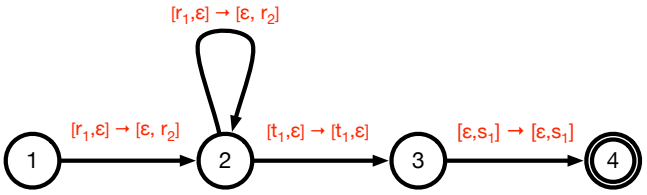
Next, we compose the two suffixes of the witness tuple with $\mathcal{A}_T^w$. Figure 6a and Figure 6b show the resulting transducers for composing the suffixes $[t_1,\ s_1]$ and $[(r_1)^+ t_1,\ s_1]$, respectively. Projecting these transducers onto their output tapes produce automata generating *transformed suffixes*.

As discussed in Section 6.1, for a given prefix all the strings generated by the suffix $[t_1,\ s_1]$ lexicographically precede all the strings generated by the suffix $[(r_1)^+ t_1,\ s_1]$. In other words, the latest string produced by the suffix $[t_1,\ s_1]$ is lexicographically earlier than the earliest string produced by the suffix $[(r_1)^+ t_1,\ s_1]$. In order to check the soundness of the transformation, we check whether the same property holds for the *transformed suffixes*.

The output program obtained by performing interchange transformation has a new order $(t_1, r_1^l, r_1^r, s_1, r_2)$. First we find the lexicographically latest string generated by the transformed suffix represented as transducer in Figure 6a, $[t_1,\ s_1]$.



(a) Suffix $[t_1, s_1]$ composed with interchange transducer.



(b) Suffix $[(r_1)^+ t_1, s_1]$ composed with interchange transducer.

Figure 6. Transformed suffixes of the witness tuple.

We refer to this string as *latest string*. Then we find the lexicographically earliest string generated by the transformed suffix represented as transducer in Figure 6b, $[t_1,\ r_2 s_1]$. We refer to this string as *earliest string*. It is evident that the latest string $[t_1,\ s_1]$ is lexicographically earlier than the earliest string $[t_1,\ r_2 s_1]$. Hence we conclude that the interchange transformation is sound for our running example.

We can also check whether the inlining transformation from Figure 3a is sound in the presence of the witness tuple. In this case, there are two possible states in the transformation transducer that a prefix drawn from $[(r_1)^*, (r_2^l|r_2^r)^*]$ can end up in, $\{1\}$ and $\{4\}$, so those are the two configurations we need to consider when testing soundness.

Configuration $\{1\}$ The latest string produced by the suffix $[t_1, s_1]$ is, simply, $[t_1, s_1]$, and the earliest string produced by the suffix $[(r_1)^+ t_1, s_1]$ is $[r_1 t_1, s_1]$, which preserves the order.

Configuration $\{4\}$ From here, the analysis is the same, and the order is preserved.

Because both prefix configurations preserve the dependence order, the transformation is sound.

6.3 Discussion

The various restrictions in PolyRec—the structure of the dependence representation, and the restrictions on the types of programs the framework can handle—are all driven by the demands of this dependence test, and, in particular, preserving its decidability.

First, the dependence representation, with all instances generated by one witness tuple suffix preceding all instances generated by the other witness tuple suffix is structured to yield a sound single-prefix dependence test. In particular, it lets us reduce the single-prefix dependence test to the problem of testing whether all the strings generated by one automaton (the output of the first suffix) precede all the strings generated by the second automaton (the output of the second suffix), which we can then, in turn, reduce to

the decidable problem of comparing the latest and earliest strings generated by those automata.

Second, the peculiar structural restriction on the transformation transducers—that they be order-free—is necessary for the overall dependence test to be decidable. Specifically, it is this order-free property that allows us to split the dependence test into two phases. The first to find the derived automaton given a prefix w, and the second to compose this derived automaton with the suffixes to perform the single-prefix dependence test. This splitting, then, lets us show that the overall process is decidable. Because we can uniquely specify the derived automaton $\mathcal{A}_T^w$ for a prefix w from the state set Q_w, we obtain the result that we can enumerate all possible Q_w by enumerating ws in a diagonalization manner. The restriction of PolyRec to perfectly-nested loops and recursion is also related to the restriction to order-free transformations, as it is easy to construct useful order-free transformations for perfectly-nested programs.

Notice that our iteration space representation can comfortably handle imperfect nesting, and our transformation representation can capture, and compose, non-order-free transformations over these imperfectly-nested programs. The witness tuple representation, likewise, can accommodate imperfect nesting. The restrictions in PolyRec seem necessary to support a sound dependence test. Generalizing PolyRec to work with a larger set of transformations and programs thus boils down to designing a more general dependence test.

7 Prototype Implementation

We built a prototype implementation of the PolyRec framework[8] that can combine a sequence of basic transformation transducers on a nested recursive code to generate a transducer that represents a complex composed transformation. When presented with a witness tuple (or tuples), our prototype applies the dependence test to verify that the dependences in that witness tuple are preserved by the composed transformation.

This section also describes three additional pieces that build towards a workflow for applying transformations to nested recursive codes: (i) how *code generation* can work alongside our prototype; (ii) a *completion procedure* that allows our prototype to search a space of transformations to find a sound transformation consistent with a partial transformation; and (iii) a sketch of a *dependence analysis* that could be used to generate witness tuples.

7.1 Code Generation

The key to code generation in our approach is that the concrete code can be generated independently of the process of composing transformations and checking that they preserve dependences. Hence, while the prototype itself does not generate code, it can easily sit alongside a code generation procedure. Because the transformation transducers are (a) based on well-known compiler transformations and (b) each preserve perfect nesting, once PolyRec verifies that the composed transformation is sound, it is possible to directly apply the (concrete) transformations, in order, to the code. Note that because PolyRec avers that the composed transformation is safe, intermediate states of the code (i.e., before all the transformations are applied) may become unsound but will be fixed by later transformations (For example, interchange can break dependences that are fixed by subsequent code motion that changes the recursion order of a function.)

7.2 Completion Procedure

Finding the right set of transformations to apply to a program is a challenging problem, even in the world of regular programs. Our prototype can be used as part of a *completion procedure* akin to those in polyhedral frameworks [3]: given a *partial* transformation (e.g., a programmer might know that changing the order of two statements, or interchanging two loops, will help locality), find a complete, sound transformation that respects that partial transformation (e.g., by performing additional code motion or interchange)[9].

This completion procedure is driven by the following observation: of the four basic transformations PolyRec supports (Section 5.3), the only two that can break (or fix) dependences are interchange and code motion, and the combination of possible dimension and statement orders is finite. Moreover, a single interchange transducer can produce any order of dimensions and a single code motion transducer can produce any order of statements. So, given a *partial* order of dimensions and statements[10], we can enumerate a (finite) set of pairs of transducers (one for interchange and one for code motion), each of which captures a different total order consistent with the provided partial order. PolyRec can then apply the dependence test to each (composed) pair to find a total order that is sound, if one exists. Any inlining or strip mining can be safely applied before (or after) this sequence is identified.

7.3 Dependence Analysis

Any dependence analysis that can generate witness tuples can be used with PolyRec. While our prototype does not generate witness tuples, here we sketch how *tree dependence analysis* [36] could be adapted to identify witness tuples. Note that tree dependence analysis only handles recursion nested inside loops, though PolyRec's witness tuple representation can capture dependences from more complex nesting structures.

[8] Available at https://bitbucket.org/plcl/polyrec

[9] We leave the additional challenge of heuristics for finding this first partial transformation to future work, though Section 9 sketches some ideas for heuristics to exploit parallelism.

[10] The order of statements in the innermost recursion encompasses the order of compound statements *and* recursive calls.

Weijiang et al. [36] target programs where recursive functions traverse tree structures; in other words the induction variables represent nodes in the tree and the compound statements in the innermost recursion access fields of the tree indexed by the induction variables. Weijiang et al. use a variant of Larus and Hilfinger [20]'s dependence analysis to find compound statements where there may be *some* instantiation of the induction variable (in two different instances) such that the accesses depend on each other. For example, statement s_1 might be n.x = ... while statement s_2 might be ... = n.left.x. These two statements depend on each other when s_2 executes at some node n, and s_1 executes at $n.left$—Weijiang et al. determine this "distance" information by looking at the common prefixes of node accesses in the two statements. The sequence of recursive calls that separate an instance at n from an instance at $n.left$ is straightforward to determine from the recursive calls in the method. This information can be directly used to construct a witness tuple: the dependence occurs between instances separated by *left* along the dimension with the induction variable n, and *any* value for the other dimensions (i.e., the witness tuple should consider all possible pairs of values in those other dimensions).

8 Evaluation

Our evaluation of PolyRec targets two key questions:

1. Can PolyRec correctly check the validity of complex, composed transformations of nested recursive spaces? Do these transformations generalize prior work? Can PolyRec check novel transformations? (And, as a subquestion: are those transformations useful?)
2. Can PolyRec find a valid transformation consistent with a partial transformation?

Benchmarks We use four test cases, all of which have at least one general recursion. The first three cases use a recursive traversal over a tree structure nested in a loop (which is rewritten into tail-recursion prior to transformation); this is the computation structure examined in several previous papers [14, 15, 39]. The last case uses two trees, with a recursion over tree B nested inside a recursion over tree A; this is the computation structure used in Sundararajah et al. [33]. The loops run for 1000 iterations, and the tree(s) have $1M$ nodes. In each of the cases, we vary the dependence structure through different instantiations of the leaf statements, such that different transformations will be legal; we provide witness tuples capturing these dependence structures to PolyRec:

Case 1 The recursion nest performs an update of the form n.x[i] += n.x[i+1], where n is the induction variable of the tree traversal and i is the induction variable of the loop. Thus, there are dependences across the loop, but in a given traversal, node updates are independent. Here, all basic transformations for the general recursion are legal.

Case 2 Here, the computation is n.x[i] = n.l.x[i] + n.r.x[i]. There are no dependences across the loop, but each instance depends on the computation at its parent in the tree. Some code motion is not legal because it may cause the tree nodes to be visited in the wrong order.

Case 3 In this case, the updates create dependences across both recursions: n.x[i] += n.l.x[i + 1] + n.r.x[i + 1]. The dependence structure prevents interchange from occurring if the inner recursion is post-order (which it is in the source), but code motion can change the recursion to pre-order to make interchange legal.

Case 4 In this case, there are *two* tree induction variables, n, traversed by the outer recursion, and m, in the inner. The update is n.x *= m.x, performing a cross product of the two trees. All transformations are legal.

For each of these cases, after our prototype verifies that a transformation is sound, we directly apply the (concrete) transformations to the initial code to produce transformed versions of the code. Appendix A includes pseudocode, detailed dependence information, and transformation examples for the various cases.

Experimental Platform Our nested recursive traversals are written in C with annotations to aid our tool. We used *ICC Compiler 16.0.3* to compile our traversals and transformed traversals. The execution platform for the various performance runs is a dual 12-core, Intel Xeon 2.7 GHz Core with 32 KB of L1 cache, 256 KB of L2 cache and 20 MB of L3 cache.

Can* PolyRec *analyze composed transformations? On our four case studies, we evaluated various combinations of PolyRec's four basic transformations, *Code Motion* (CM), *Inlining* (IL), *Interchange* (IC) and *Strip Mining* (SM). Table 1 shows, for the four case studies, which compositions of transformations PolyRec is able to verify as sound.

Applying strip mining and interchange to a recursion nested within a loop (our first three test cases) produces the same schedule transformation as point blocking [14] while adding inlining produces the same schedule transformation as traversal splicing [15], and interchanging two general recursions (our fourth test case) produces the same schedule transformation as recursion interchange [33], showing that PolyRec generalizes prior work [11] Case 3 requires applying code motion to make interchange legal—this represents a new transformation that requires a compositional framework, as no prior work could capture this transformation.

For completeness, Table 1 also shows the performance of the transformed code over the untransformed baselines. There are two key points here. First, these transformations are able to improve the performance of nested codes. Second, different combinations of transformations have different

[11] Prior work includes transformation-specific code generation optimizations, which our prototype code generator does not support, but the fundamental scheduling transformations are equivalent to ours.

Table 1. Performance Results of the Case Study

Case	Transform	Runtime(s)
Case 1	Baseline	79.95
	IC	0.96
	IC-SM-CM	0.59
	IC-SM-IL-CM	0.62
Case 2	Baseline	61.61
	IC	1.04
	IC-SM	2.92
	IC-SM-IL	0.77
Case 3	Baseline	61.48
	IC-CM	1.05
	IC-SM-CM	3.04
	IC-SM-IL-CM	0.56
Case 4	Baseline	5.89
	IC	2.92

performance, confirming the value proposition of having a framework that allows programmers to explore a space of possible transformations to determine which are sound.

Can* PolyRec *find valid transformations? We also test the ability of our completion procedure to find a complete transformation given a partial transformation using Case 3. Here, interchange is only legal if composed with code motion (and vice versa). We start with two partial transformations: in one, the inner recursion is changed from post-order to pre-order, requiring the transformation to be completed with interchange, and in the other, we begin with interchange, require the transformation to be completed with code motion. In both cases, given the finite set of possible completions, PolyRec correctly identifies a sound transformation consistent with the partial transformation.

9 Incorporating Parallelism

PolyRec currently targets transformations of sequential code. An interesting question is how to incorporate parallelism transformations into the framework. In some sense, parallelism is simpler than the types of transformations PolyRec targets: rather than worrying about the specific structure of dependences and whether that structure permits a transformation, parallelizing code requires "simply" showing the absence of a dependence. PolyRec can identify opportunities for parallelism by checking if there are no dependences carried along that dimension. A simple test for a lack of dependences across a dimension is whether all witness tuples have no recursive call labels in either suffix for a given dimension. In this case, all pairs of dependences have the same "value" in that dimension, and that dimension is parallel.

We can add heuristics to PolyRec to exploit parallelism in one of several ways. First, mirroring how polyhedral frameworks can be used to generate coarse-grained parallelism, parallel dimensions can be moved to the outside through loop interchange (which will always be safe). The recursive calls in the outer dimensions will thus produce coarse-grained tasks that can be parallelized using standard techniques (e.g., using Cilk-style spawn calls).

PolyRec can also be used to move parallel dimensions to the *inner* parts of a loop nest, creating fine-grained parallelism. This fine-grained parallelism can be exploited through standard vectorization techniques if the inner dimension is a simple loop. Indeed, interchanging coarse-grained parallel loops with (non-parallel) recursive methods to promote vectorization is essentially Jo and Kulkarni's approach to vectorizing tree applications [13]. If, instead, the parallel dimension is a general (non-loop) recursion, Ren et al.'s vectorization technique can be applied [27]. While Ren et al.'s specific code generation technique is outside the scope of PolyRec, the framework can soundly generate the high level code structure necessary to then apply Ren et al.'s technique.

10 Conclusions

Despite the long history of dependence analysis and transformation frameworks for loop-based programs, there are no comparable frameworks for programs that use recursion. PolyRec is the first comprehensive, compositional framework for recursive programs that provides an end-to-end strategy for representing schedules, transformations of those schedules, and dependences, allowing nested recursive programs (including combinations of recursion and loops) to be soundly transformed. Our prototype implementation is able to analyze complex transformations that arise in prior work, as well as generate and reason about new transformations.

Acknowledgments

This work was supported in part by National Science Foundation award CCF-1725672, and by Department of Energy award DE-SC0010295. We would like to thank Sriram Krishnamoorthy, Ryan Newton, Laith Sakka, and Xiaokang Qiu for their feedback during discussions of this work. We would like to thank our shepherd, Ayal Zaks, for his help with the revisions of this paper, as well as the anonymous reviewers for their suggestions and comments.

References

[1] John R. Allen and Ken Kennedy. 1984. Automatic Loop Interchange. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction (SIGPLAN '84)*. ACM, New York, NY, USA, 233–246. https://doi.org/10.1145/502874.502897

[2] Pierre Amiranoff, Albert Cohen, and Paul Feautrier. 2006. Beyond Iteration Vectors: Instancewise Relational Abstract Domains. In *Proceedings of the 13th International Conference on Static Analysis (SAS'06)*. Springer-Verlag, Berlin, Heidelberg, 161–180. https://doi.org/10.1007/11823230_11

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

[4] A. Cohen and J. F. Collard. 1998. Instance-wise reaching definition analysis for recursive programs using context-free transductions. In

Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques. 332–339. https://doi.org/10.1109/PACT.1998.727269

[5] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem: I. One-dimensional Time. *Int. J. Parallel Program.* 21, 5 (Oct. 1992), 313–348. https://doi.org/10.1007/BF01407835

[6] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming* 21, 6 (01 Dec 1992), 389–420. https://doi.org/10.1007/BF01379404

[7] Paul Feautrier. 1998. A Parallelization Framework for Recursive Tree Programs. In *Proceedings of the 4th International Euro-Par Conference on Parallel Processing (Euro-Par '98)*. Springer-Verlag, Berlin, Heidelberg, 470–479. http://dl.acm.org/citation.cfm?id=646663.700133

[8] Rakesh Ghiya, Laurie J. Hendren, and Yingchun Zhu. 1998. Detecting Parallelism in C Programs with Recursive Data 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] T. V. Griffiths. 1968. The Unsolvability of the Equivalence Problem for Λ-Free Nondeterministic Generalized Machines. *J. ACM* 15, 3 (July 1968), 409–413. https://doi.org/10.1145/321466.321473

[10] N. Hegde, J. Liu, K. Sundararajah, and M. Kulkarni. 2017. Treelogy: A benchmark suite for tree traversals. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 227–238. https://doi.org/10.1109/ISPASS.2017.7975294

[11] Mans Hulden. 2017. Rewrite Rule Grammars with Multitape Automata. *J. Language Modelling* 5, 1 (2017), 107–130. https://doi.org/10.15398/jlm.v5i1.158

[12] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. 1994. A General Data Dependence Test for Dynamic, Pointer-based Data Structures. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 218–229. https://doi.org/10.1145/178243.178262

[13] Youngjoon Jo, Michael Goldfarb, and Milind Kulkarni. 2013. Automatic Vectorization of Tree Traversals. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. IEEE Press, Piscataway, NJ, USA, 363–374. http://dl.acm.org/citation.cfm?id=2523721.2523770

[14] 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 (OOPSLA '11)*. ACM, New York, NY, USA, 463–482. https://doi.org/10.1145/2048066.2048104

[15] 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 (OOPSLA '12)*. ACM, New York, NY, USA, 355–374. https://doi.org/10.1145/2384616.2384643

[16] Ronald M. Kaplan and Martin Kay. 1994. Regular Models of Phonological Rule Systems. *Comput. Linguist.* 20, 3 (Sept. 1994), 331–378. http://dl.acm.org/citation.cfm?id=204915.204917

[17] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. 1967. The Organization of Computations for Uniform Recurrence Equations. *J. ACM* 14, 3 (July 1967), 563–590. https://doi.org/10.1145/321406.321418

[18] Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[19] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/106972.106981

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

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

[22] William Pugh. 1991. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 4–13. https://doi.org/10.1145/125826.125848

[23] William Pugh and David Wonnacott. 1994. *Nonlinear Array Dependence Analysis*. Technical Report. College Park, MD, USA. http://hdl.handle.net/1903/674

[24] M. O. Rabin and D. Scott. 1959. Finite Automata and Their Decision Problems. *IBM J. Res. Dev.* 3, 2 (April 1959), 114–125. https://doi.org/10.1147/rd.32.0114

[25] Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noel Pouchet, Fabrice Rastello, Robert J. Harrison, and P. Sadayappan. 2016. A Domain-specific Compiler for a Parallel Multiresolution Adaptive Numerical Simulation Environment. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 40, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014958

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

[27] Bin Ren, Youngjoon Jo, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. 2015. Efficient Execution of Recursive Programs on Commodity Vector Hardware. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 509–520. https://doi.org/10.1145/2737924.2738004

[28] Radu Rugina and Martin C. Rinard. 2005. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. *ACM Trans. Program. Lang. Syst.* 27, 2 (March 2005), 185–235. https://doi.org/10.1145/1057387.1057388

[29] 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

[30] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. 2003. Compile-time Composition of Run-time Data and Iteration Reorderings. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 91–102. https://doi.org/10.1145/781131.781142

[31] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. 2016. An Approach for Code Generation in the Sparse Polyhedral Framework. *Parallel Comput.* 53, C (April 2016), 32–57. https://doi.org/10.1016/j.parco.2016.02.004

[32] Michelle Mills Strout, Fabio Luporini, Christopher D. Krieger, Carlo Bertolli, Gheorghe-Teodor Bercea, Catherine Olschanowsky, J. Ramanujam, and Paul H. J. Kelly. 2014. Generalizing Run-Time Tiling with the Loop Chain Abstraction. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS '14)*. IEEE Computer Society, Washington, DC, USA, 1136–1145. https://doi.org/10.1109/IPDPS.2014.118

[33] Kirshanthan Sundararajah, Laith Sakka, and Milind Kulkarni. 2017. Locality Transformations for Nested Recursive Iteration Spaces. *SIGPLAN Not.* 52, 4 (April 2017), 281–295. https://doi.org/10.1145/3093336.3037720

[34] Robert A. van Engelen, J. Birch, Y. Shou, B. Walsh, and Kyle A. Gallivan. 2004. A Unified Framework for Nonlinear Dependence Testing and Symbolic Analysis. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS '04)*. ACM, New York, NY, USA, 106–115. https://doi.org/10.1145/1006209.1006226

[35] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 521–532. https://doi.org/10.1145/2737924.2738003

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

[37] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*. ACM, New York, NY, USA, 30–44. https://doi.org/10.1145/113445.113449

[38] M. Wolfe. 1989. More Iteration Space Tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing (Supercomputing '89)*. ACM, New York, NY, USA, 655–664. https://doi.org/10.1145/76263.76337

[39] Xingbin Zhang and Andrew A. Chien. 1997. Dynamic Pointer Alignment: Tiling and Communication Optimizations for Parallel Pointer-based Computations. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '97)*. ACM, New York, NY, USA, 37–47. https://doi.org/10.1145/263764.263771

A Appendix

This appendix explains our case study in detail. Focus of our evaluation here is showing the capability of PolyRec in performing various transfomations to explore different schedules, rather than analyzing performance efficiency of a particular transformations. Every case presented in the Section 8, we show the baseline code using our familiar notation presented in Section 3. For instance, the recursive function representing first dimension is named as r1 and there is a single call to r2 in the body of r1. Every general recursion in our case study traverses a binary tree. Hence these recursions have calls to left and right sub-trees and the induction variable of this general recursion is n. Left and right sub-trees of tree node Node * n are accessed by n->l and n->r, respectively. In every case a single statement s_1 is perfectly nested by two recursive functions. In any piece of code N is the number of iterations of the initial "linear recursion" and M is the strip size in a strip mining transformation. Induction variable of any "linear recursion" is denoted by parameters i and j. In order to identify the basic transformations, we use the same acronyms used in Section 8. Additionally, we mention specific details of a basic transformation within parantheses. For instance, if code motion makes a recursion pre-order then that is denoted by CM(pre). Similarly, inlining the call traversing the left sub-tree and strip mining of strip size 100 are denoted by IL(l) and SM(100), respectively.

A.1 Case 1

Baseline code for Case 1 is shown in Figure 7. The statement s_1 of this case is n->x[i] = n->x[i] + n->x[i+1]. The way induction variable i is used to index n->x has created a dependency between successive iterations of the outer "linear recursion". But there is no dependency between the statement and the calls of inner general recursion. In loop-universe, this is known as "loop-carried dependency". Since this is the only type of dependency present in this program, the generated witness tuple for this piece of code is shown in Equation 1

$$< [r_1^*, (r_2^l | r_2^r)^*], ([t_1, s_1], [(r_1)^+ t_1, s_1]) > . \quad (1)$$

Following subsections explain legal transformations of this code as a composition of basic transformations:

```
1   void r1(int i, Node * n){
2       if(i >= N) return;
3       r2(i, n); //t1
4       r1(i+1, n); //r1
5   }

7   void r2(int i, Node * n){
8       if (n == NULL) return;
9       r2(i, n->l); //r2l
10      r2(i, n->r); //r2r
11      n->x[i] = n->x[i] + n->x[i+1]; //s1
12  }
```

Figure 7. Baseline Code

A.1.1 Transformation 1: IC

This is a single basic transformation. It interchanges the outer and inner recursions. By performing this transformation we have enhanced the reuse distance of the elements in array n->x of every tree node n compared to the baseline. This transformation is shown in Figure 8.

A.1.2 Transformation 2: IC-SM(100)-CM(pre)

Here we have a composition of three different basic transformations. First, inner recursion of the baseline is transformed from post-order to be pre-order by code motion. Then, the outer "linear recursion" is strip mined for a strip size of 100. Finally, the inner recursion is interchanged with strip mined outer recursion. Figure 9 shows the transformed code.

A.1.3 Transformation 3: IC-SM(100)-IL(l)-CM(pre)

This transformation is similar to the previous one except it adds an inlining of the left call, after code motion. The transformed code is shown in Figure 10.

```
void r1(int i, Node * n){
 if (n == NULL) return;
 r1(i, n->l); //r1l
 r1(i, n->r); //r1r
 r2(i, n); //t1

}

void r2(int i, Node * n){
 if(i >= N) return;
 n->x[i] = n->x[i] + n->x[i+1]; //s1
 r2(i+1, n); //r2
}
```

Figure 8. Transformed Code for IC

```
void r1(int i, int j, Node * n){
 if(i >= N) return;
 r2(i, i, n); //t1
 r1(i+M, i, n); //r1
}

void r2(int i, int j, Node * n){
 if (n == NULL) return;
 r3(i, j, n); //t2
 r2(i, j, n->l); //r2l
 r2(i, j, n->r); //r2r
}

void r3(int i, int j, Node * n){
 if(j >= M || i >= N) return;
 n->x[i+j] = n->x[i+j] + n->x[i+j+1]; //s1
 r3(i, j+1, n); //r3
}
```

Figure 9. Transformed Code for IC-SM(100)-CM(pre)

A.1.4 Transformation 4: IL(r)

It is a single basic transformation. We have inlined the right call of inner recursion. This is shown in Figure 11.

A.2 Case 2

Baseline code for Case 2 is shown in Figure 12. The statement s_1 of this case is n->x[i] = n->l->x[i] + n->r->x[i]. Induction variable of the general recursion accesses both left and right subtrees in this statement. This results in dependencies between statement and both calls of inner general recursion. But the induction variable of outer "linear recursion" used to index does not create any new dependencies. We call this a "recursion-carried dependency". The witness tuple in

```
void r1(int i, int j, Node * n){
 if(i >= N) return;
 r2(i, i, n); //t1
 r1(i+M, i, n); //r1
}

void r2(int i, int j, Node * n){
 if (n == NULL) return;
 r3(i, j, n); //t2
 if (n->l != NULL) {
  r2(i, j, n->l->l); //r2ll
  r2(i, j, n->l->r); //r2lr
 }
 r2(i, j, n->r); //r2r
}

void r3(int i, int j, Node * n){
 if(j >= M || i >= N) return;
 n->x[i+j] = n->x[i+j] + n->x[i+j+1]; //s1
    if (n->l != NULL)
        n->l->x[i+j] = n->l->x[i+j] + n->l->x[i+j
            +1]; //s1l
 r3(i, j+1, n); //r3
}
```

Figure 10. Transformed Code for IC-SM(100)-IL(l)-CM(pre)

```
void r1(int i, Node * n){
    if(i >= N) return;
    r2(i, n); //t2
    r1(i+1, n); //r1
}

void r2(int i, Node * n){
    if (n == NULL) return;
    r2(i, n->l); //r2l
    if (n->r != NULL){
        r2(i, n->r->l); //r2rl
        r2(i, n->r->r); //r2rr
        n->r->x[i] = n->r->x[i] + n->r->x[i+1]; //
            s1r
    }
    n->x[i] = n->x[i] + n->x[i+1]; //s1
}
```

Figure 11. Transformed Code for IL(r)

Equation 2 captures this dependence.

$$< [r_1^*, (r_2^l | r_2^r)^*], ([t_1, (r_2^l | r_2^r) s_1], [t_1, s_1]) > . \quad (2)$$

Following subsections explain legal transformations for this code as a composition of basic transformations:

```
1   void r1(int i, Node * n){
2       if(i >= N) return;
3       r2(i, n); //t2
4       r1(i+1, n); //r1
5   }

7   void r2(int i, Node * n){
8       if (n == NULL) return;
9       r2(i, n->l); //r2l
10      r2(i, n->r); //r2r
11      if (n->l != NULL && n->r != NULL)
12          n->x[i] = n->l->x[i] + n->r->x[i]; //s1
13  }
```

Figure 12. Baseline Code

A.2.1 Transformation 1: IC

This transformation is exactly similar to the one in Section A.1.1. Figure 13 shows the transformed code.

```
1   void r1(int i, Node * n){
2       if (n == NULL) return;
3       r1(i, n->l); //r1l
4       r1(i, n->r); //r1r
5       r2(i, n); //t1

7   }

9   void r2(int i, Node * n){
10      if(i >= N) return;
11      if (n->l != NULL && n->r != NULL)
12          n->x[i] = n->l->x[i] + n->r->x[i]; //s1
13      r2(i+1, n); //r2
14  }
```

Figure 13. Transformed Code for IC

A.2.2 Transformation 2: IC-SM(100)

It is similar to the previous case except for strip mining. Instead of completely interchanging outer and inner recursions first the outer recursion is strip mined and then interchanged with inner recursion. This whole transformation becomes handy when the outer "linear recursion" long enough not to yield any locality benefits. Figure 14 shows the transformed code.

A.2.3 Transformation 3: IC-SM(100)-IL(r)

This transformation is similar to previous one in the sense of interchange and strip mine. But before performing those transformations, right call of the inner recursion has been

```
1   void r1(int i, int j, Node * n){
2       if(i >= N) return;
3       r2(i, j, n); //t1
4       r1(i+M, j, n); //r1
5   }

7   void r2(int i, int j, Node * n){
8       if (n == NULL) return;
9       r2(i, j, n->l); //r2l
10      r2(i, j, n->r); //r2r
11      r3(i, j, n); //t2
12  }

14  void r3(int i, int j, Node * n){
15      if(j >= M || i >= N) return;
16      if (n->l != NULL && n->r != NULL)
17          n->x[i+j] = n->l->x[i+j] + n->r->x[i+j]; //
                s1
18      r3(i, j+1, n); //r3
19  }
```

Figure 14. Transformed Code for IC-SM(100)

inlined. The notable difference is that inlining gives an additional statement. Since we preserve the order of these statements, for all our intents and purposes these two statements could be considered as one compound statement. Figure 15 shows the transformed code for this transformation.

A.2.4 Transformation 4: IL(r)

This transformation is exaclty same as the one in Section A.1.4 Figure 16 shows the code.

A.3 Case 3

Baseline code for Case 3 is shown in Figure 17. The statement s_1 of this case is n->x[i] = n->l->x[i+1] + n->r->x[i+1]. The way induction variables used to index in statement s_1, this program has both "loop-carried dependency" and "recursion-carried dependency". The witness tuple shown in Equation 3 captures both type of dependencies.

$$< [r_1^*, (r_2^l | r_2^r)^*], ([r_1 t_1, (r_2^l | r_2^r) s_1], [t_1, s_1]) > \quad (3)$$

The dependency structure in Case 3 is a mix of Case 1 and Case 2. Following subsections explain legal transformations for this code as a composition of basic transformations:

A.3.1 Transformation 1: IC-CM(pre)

This case is similar to any interchange transformation mentioned before. Our main purpose here is performing interchange. But without performing code motion, trying to perform interchange would be illegal because of the dependency structure for this particular case. In this case code motion is

```
void r1(int i, int j, Node * n){
    if(i >= N) return;
    r2(i, i, n); //t1
    r1(i+M, i, n); //r1
}

void r2(int i, int j, Node * n){
    if (n == NULL) return;
    r2(i, j, n->l); //r2l
    if (n->r != NULL) {
        r2(i, j, n->r->l); //r2rl
        r2(i, j, n->r->r); //r2rr
    }
    r3(i, j, n); //t2
}

void r3(int i, int j, Node * n){
    if(j >= M || i >= N) return;
    if(n->r != NULL)
        if (n->r->l != NULL && n->r->r != NULL)
            n->r->x[i+j] = n->r->l->x[i+j] + n->r
                ->r->x[i+j]; //s1r
    if (n->l != NULL && n->r != NULL)
        n->x[i+j] = n->l->x[i+j] + n->r->x[i+j]; //
            s1
    r3(i, j+1, n); //r3
}
```

Figure 15. Transformed Code for IC-SM(100)-IL(r)

```
void r1(int i, Node * n){
    if(i >= N) return;
    r2(i, n); //t1
    r1(i+1, n); //r1
}

void r2(int i, Node * n){
    if (n == NULL) return;
    r2(i, n->l); //r2l
    if (n->r != NULL){
        r2(i, n->r->l); //r2rl
        r2(i, n->r->r); //r2rr
        if (n->r->l != NULL && n->r->r != NULL)
            n->r->x[i] = n->r->l->x[i] + n->r->r
                ->x[i]; //s1r
    }
    if (n->l != NULL && n->r != NULL)
        n->x[i] = n->l->x[i] + n->r->x[i]; //s1
}
```

Figure 16. Transformed Code for IL(r)

```
void r1(int i, Node * n){
    if(i >= N) return;
    r2(i, n); //t1
    r1(i+1, n); //r1
}

void r2(int i, Node * n){
    if (n == NULL) return;
    r2(i, n->l); //r2l
    r2(i, n->r); //r2r
    if (n->l != NULL && n->r != NULL)
        n->x[i] = n->l->x[i+1] + n->r->x[i+1]; //s1
}
```

Figure 17. Baseline Code

crucial to perform interchange. This transformation is shown in Figure 18.

```
void r1(int i, Node * n){
    if (n == NULL) return;
    r2(i, n); //t1
    r1(i, n->l); //r1l
    r1(i, n->r); //r1r

}

void r2(int i, Node * n){
    if(i >= N) return;
    if (n->l != NULL && n->r != NULL)
        n->x[i] = n->l->x[i+1] + n->r->x[i+1]; //s1
    r2(i+1, n); //r2
}
```

Figure 18. Transformed Code for IC-CM(pre)

A.3.2 Transformation 2:IC-SM(100)-CM(pre)

Even though this is similar to any interchange composed with strip mining, the notable fact here is that without code motion any sort of interchange would be illegal. Figure 19 shows the code for this transformation.

A.3.3 Transformation 3: IC-SM(100)-IL(l)-CM(pre)

Except for inlining this transformation is similar to the previous one. But adding a small tweak with inlining changes the performance. This transformation is shown in Figure 20.

A.4 Case 4

Baseline code for Case 4 is shown in Figure 21. It is known that all the instances in this case are independent of each

```
1   void r1(int i, int j, Node * n){
2       if(i >= N) return;
3       r2(i, j, n); //t1
4       r1(i+M, j, n); //r1
5   }

7   void r2(int i, int j, Node * n){
8       if (n == NULL) return;
9       r3(i, j, n); //t2
10      r2(i, j, n->l); //r2l
11      r2(i, j, n->r); //r2r
12  }

14  void r3(int i, int j, Node * n){
15      if(j >= M || i >= N) return;
16      if (n->l != NULL && n->r != NULL)
17          n->x[i+j] = n->l->x[i+j+1] + n->r->x[i+j
                +1]; //s1
18      r3(i, j+1, n); //r3
19  }
```

Figure 19. Transformed Code for IC-SM(100)-CM(pre)

```
1   void ir1(int i, int j, Node * n){
2       if(i >= N) return;
3       r2(i, i, n); //t1
4       r1(i+M, i, n); //r1
5   }

7   void r2(int i, int j, Node * n){
8       if (n == NULL) return;
9       r3(i, j, n); //t2
10      if (n->l != NULL) {
11          r2(i, j, n->l->l); //r2ll
12          r2(i, j, n->l->r); //r2lr
13      }
14      r2(i, j, n->r); //r2r
15  }

17  void r3(int i, int j, Node * n){
18      if(j >= M || i >= N) return;
19      if (n->l != NULL && n->r != NULL)
20          n->x[i+j] = n->l->x[i+j+1] + n->r->x[i+j
                +1]; //s1
21      if(n->l != NULL)
22          if (n->l->l != NULL && n->l->r != NULL)
23              n->l->x[i+j] = n->l->l->x[i+j+1] + n->
                    l->r->x[i+j+1]; //s1l
24      r3(i, j+1, n); //r3
25  }
```

Figure 20. Transformed Code for IC-SM(100)-IL(l)-CM(pre)

other. Hence, no witness tuple is required. Even though there is no well-known dependence analysis in order to automatically produce witness tuples for this type of nested general recursion, we have included this case in our evaluation to show that PolyRec is capable of generating code for interchange in general.

```
1   void r1(Node * n, Node * m){
2       if (n == NULL) return;
3       r2(n, m); //t1
4       r1(n->l, m); //r1l
5       r1(n->r, m); //r1r
6   }

8   void r2(Node * n, Node * m){
9       if (m == NULL) return;
10      n->y = n->x + m->x; //s1
11      r2(n, m->l); //r2l
12      r2(n, m->r); //r2r
13  }
```

Figure 21. Baseline Code

A.4.1 Transformation 1: IC

In the baseline code, outer recursion traverses a smaller tree while the inner recursion traverses a huge one. This transformation interchanges these traversals. Once the code is transformed to make outer recursion traverses the huge tree, it enhances the locality. This transformation is shown in Figure 22.

```
1   void r1(Node * n, Node * m){
2       if (m == NULL) return;
3       r2(n, m); //t1
4       r1(n, m->l); //r1l
5       r1(n, m->l); //r1r
6   }

8   void r2(Node * n, Node * m){
9       if (n == NULL) return;
10      n->y = n->x + m->x; //s1
11      r2(n->l, m); //r2l
12      r2(n->r, m); //r2r
13  }
```

Figure 22. Transformed Code for IC