11-1-2018

Scheduling Transformation and Dependence Tests for Recursive Programs

Kirshanthan Sundararajah
Purdue University, ksundar@purdue.edu

Milind Kulkarni
Purdue University, milind@purdue.edu

Scheduling Transformations and Dependence Tests for Recursive Programs

KIRSHANTHAN SUNDARARAJAH, Purdue University, USA
MILIND KULKARNI, Purdue University, USA

Scheduling transformations reorder the execution of operations in a program to improve locality and/or parallelism. The polyhedral model provides a general framework for performing *instance-wise* scheduling transformations for regular programs, reordering the iterations of loops that operate over dense arrays through transformations like tiling. There is no analogous framework for recursive programs—despite recent interest in optimizations like tiling and fusion for recursive applications. This paper presents PolyRec, the first general framework for applying scheduling transformations—like inlining, interchange, and code motion—to *nested recursive programs*, and reasoning about their correctness. We describe the phases of PolyRec—representing dynamic instances, applying transformations, reasoning about correctness—and show that PolyRec is able to apply sophisticated, composed transformations to complex, nested recursive programs and improve performance through enhanced locality.

Additional Key Words and Phrases: Dependence Analysis, Scheduling Transformations, Locality, Recursion

1 INTRODUCTION

Over the past few decades, researchers have developed a large catalog of transformations for *regular* programs—loop-based programs that operate over arrays and matrices—such as loop tiling, loop interchange, loop fusion, and unrolling [Kennedy and Allen 2002]. In recent years, many analogous transformations have been developed for *irregular* programs that use recursion to manipulate lists, trees and graphs [Jo and Kulkarni 2011, 2012; Rajbhandari et al. 2016a,b; Sakka et al. 2017; Sundararajah et al. 2017]. 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 and exploit parallelism by locating independent computations.

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. In the world of loops and matrices, frameworks such as the *polyhedral model* [Feautrier 1992a,b] 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 apply loop transformations to programs [Bondhugula et al. 2008]. 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 [Rajbhandari et al. 2016a; Sakka et al. 2017; Weijiang et al. 2015], when dependence analyses are performed at all. This paper presents PolyRec, the first framework for reasoning about, and soundly applying, a large class of transformations on irregular, recursive programs—the first steps towards an analog of the polyhedral framework for irregular programs.

Authors' addresses: Kirshanthan Sundararajah, School of Electrical and Computer Engineering, Purdue University, 465 Northwestern Avenue, West Lafayette, IN, 47906, USA, ksundar@purdue.edu; Milind Kulkarni, School of Electrical and Computer Engineering, Purdue University, 465 Northwestern Avenue, West Lafayette, IN, 47906, USA, milind@purdue.edu.

2018. 2475-1421/2018/11-ART1 $15.00
https://doi.org/

1.1 Dependence analysis frameworks

Following the structure of polyhedral compilers, we can think of a dependence analysis framework as consisting of several elements. This paper presents novel instantiations of these elements that work for recursive programs. These elements are:

A representation of the iteration space Because both loop-based and recursive programs execute the same operation(s) repeatedly, an analysis framework needs an *instance-wise* representation of the operations of a program, representing the dynamic instances of each operation rather than just the static code. In the polyhedral framework, this representation for a loop nest is a system of linear inequalities forming a polyhedron capturing the loop bounds combined with a scheduling function enumerating the instances (integer points) within that polyhedron. PolyRec targets "perfect" nesting of general iteration constructs (recursive functions nested within loops [Jo and Kulkarni 2011] or vice versa, or even recursion nested within other recursion [Sundararajah et al. 2017]). It 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.

A representation of scheduling transformations In the polyhedral framework, transformations such as loop interchange can be represented as linear transformations of the instance polyhedron, leading to a different schedule of computations when the polyhedron's points are enumerated. PolyRec represents scheduling transformations as *multitape finite state transducers*, mapping each instance to another instance, with the new lexicographic order representing the new schedule.

A representation of dependences and a dependence test To ensure the soundness of transformations, 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. In the polyhedral framework, dependences are represented with polyhedra. The dependence test applies the transformation to these polyhedra to determine if any dependent iterations get reordered. In PolyRec, dependences are represented by *witness tuples* that capture sets of dependent instances. PolyRec's dependence test applies the 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.

Code generation Finally, the transformed schedule must be synthesized back into code that can be efficiently executed. PolyRec performs code generation by incrementally applying transformations to the original code—in essence, mimicking the transformations applied to the iteration space automaton.

PolyRec is a first step towards a general framework for reasoning about and transforming arbitrary combinations of loops and recursion. To demonstrate the utility of PolyRec, we show how several specific transformations from the literature—inlining, interchange [Jo and Kulkarni 2011; Sundararajah et al. 2017], code motion [Sakka et al. 2017], and strip-mining—can be represented using PolyRec's multitape transducers. This allows these transformations to be arbitrarily combined and composed, generalizing their prior use.

In particular, we show that PolyRec can automatically apply, check, and synthesize fairly sophisticated transformations of nested recursive programs, including transformations that are equivalent to combinations of point blocking [Jo and Kulkarni 2011] and traversal splicing [Jo and Kulkarni 2012].

1.2 Contributions

In summary, the contributions of this paper are:

(1) We develop a new framework, PolyRec, that can (a) lift perfectly nested recursive programs into a high level, *instance-wise* representation; (b) represent and apply scheduling transformations on that representation; (c) represent dependences between instances of computations; and (d) test whether these dependences are violated by transformations. PolyRec is the first general scheduling transformation framework for recursive programs.
(2) We present instantiations of PolyRec to apply the composition of several basic transformations of recursive programs (inlining, interchange, strip mining, and code motion) and to generate witness tuples—and hence check the soundness of transformations—for certain types of programs.
(3) We show that we can use PolyRec to automatically analyze and soundly transform nested recursive programs to improve locality and hence performance.

1.3 Outline

The rest of the paper is organized as follows. Section 2 provides background on instance-wise analysis and transformation of recursive programs (including related work). Section 3 gives a high level overview of PolyRec's operations. Section 4 lays the groundwork for explaining PolyRec in more detail by defining a core language of nested recursion that we target. Section 5 explains PolyRec's multitape automaton representation of iteration spaces for nested recursion. Section 6 presents the use of transducers to capture scheduling transformations, and shows how several well-known transformations can be expressed as transducers. Section 7 describes the witness tuple representation of dependences and shows how this representation can be used to check the soundness of transformations; this section also describes an analysis for generating witness tuples for certain kinds of programs. Section 8 presents PolyRec's code generation strategy. Section 9 evaluates PolyRec by analyzing and transforming complex, nested-recursion programs and showing that these transformations yield performance benefits. Finally, Section 10 concludes.

2 BACKGROUND

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

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*.

Traditional analyses, such as reaching definitions, consider the *static* behavior of statements, without considering that statements that execute within a loop or recursive code have different

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        A[i+1][j] = A[i][j] //s1
```

(a) Doubly-nested loop

```
for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
        A[i+1][j] = A[i][j] //s1
```

(b) After loop interchange

```
for (i = 0; i < N; i++)
    rec(i, root)

void rec(int i, Node n)
    if (n == null) return
    n.x += i //s1
    rec(i, n.right)
    rec(i, n.left)
```

(c) Recursion nested in loop

```
void rec(Node n)
    if (n == null) return
    for (i = 0; i < N; i++)
        n.x += i //s1
    rec(n.right)
    rec(n.left)
```

(d) After recursion/loop interchange

Fig. 1. Examples of schedule transformations

behavior for each iteration. Consider the loop in Figure 1a, with a single statement s_1. Reaching definitions analysis would (correctly) determine that s_1 depends *on itself*, and hence would (incorrectly) determine that a transformation such as loop interchange was not legal.

Looking deeper, we see that each time s_1 executes, it takes on a different value for i and j from the loops, and hence the dependences are more structured than "the statement depends on itself." An *instance-wise* analysis of this loop would parametrize the statement: $s_1(i, j)$, and would find that the statement executing at *instance* (i, j) writes to the same location that a statement executing at instance $(i + 1, j)$ reads from. Armed with this information, a transformation framework like the polyhedral model [Bondhugula et al. 2008; Feautrier 1992a,b] would correctly determine that loop interchange was legal (Figure 1b).

The same considerations apply when considering code that mixes loops and recursion, as in Figure 1c. Here, the function rec recurses over some tree structure, and the code s_1 executes once for each combination of i and n (where n essentially represents a node in the tree). Analyses that look for dependences in statements that access recursive structures (e.g., [Ghiya et al. 1998; Hummel et al. 1994; Larus and Hilfinger 1988; Rugina and Rinard 2005]) will correctly say that there is a dependence from that statement to itself. But an *instance-wise* analysis of this code shows that s_1 executing at (i, n) has a dependence with $(i+1, n)$ (the second statement has a different value for the loop induction variable, but executes at the same node in the tree), and hence can be *interchanged* to produce the code in Figure 1d [Jo and Kulkarni 2011], yielding better performance due to increased locality in the tree.

2.2 Instance-wise Analysis for Recursive Programs

As mentioned above, instance-wise analysis is common for regular programs that deal with nested loop structures that operate over dense arrays. 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. [2006] (building on prior work by Cohen and Collard [1998] and Feautrier [1998]).

The key challenge in instance-wise analysis is providing a scheme that uniquely names the individual *dynamic* instances of a static statement. (For example, in the world of loops, an instance can be named by providing the values of the induction variables at that dynamic instance). Amiranoff et al. [2006] generate a context-free language representation of a recursive program that uniquely labels each dynamic instance of a statement using a trace string. Using these strings they can define a dependence analysis that determines the set of dependent (dynamic) instances in a program, using that information to parallelize the program.

Amiranoff et al. [2006]'s 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). While they handle complex control structures such as loops nested inside recursion, they do not consider nested recursive methods, nor transformations over that nested structure such as interchange [Jo and Kulkarni 2011]. As they do not consider representing these transformations, they also do not provide a theory for reasoning about their correctness.

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 (as in the example in Figures 1c and 1d) [Jo and Kulkarni 2011, 2012; Weijiang et al. 2015] to frameworks that target *fusing* multiple recursive traversals together [Petrashko et al. 2017; Rajbhandari et al. 2016a,b; Sakka et al. 2017] to those that transform multiple recursive functions nested inside one another [Sundararajah et al. 2017]. 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 [Rajbhandari et al. 2016a,b; Sakka et al. 2017; Weijiang et al. 2015]. However, these frameworks are *ad hoc*: they do not provide general ways of reasoning about combinations of recursion and loops, nor general, composable ways for reasoning about transformation correctness, instead focusing on specific techniques to prove the correctness of specific transformations.

This paper combines the best of both worlds. As in Amiranoff et al. [2006], we provide a means of labeling every dynamic instance in a nest of recursive methods. (By transforming loops into tail-recursion, we can also uniformly handle combinations of loops and recursion.) We go further in providing machinery for representing and composing transformations on recursive codes (including code motion and interchange). Finally, we define dependence representation that can be used to reason about transformation soundness. We thus partially generalize the prior work on transformations for recursive programs.[1]

2.3 Other Related Work

There are numerous frameworks that reason about nested loops with affine loop bounds and affine array subscripts [Allen and Kennedy 1984; Bondhugula et al. 2008; Feautrier 1992a,b; Lam et al. 1991; Pugh 1991; Wolf and Lam 1991; Wolfe 1989]. 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 [Pugh and Wonnacott 1994; van Engelen et al. 2004], and to handle sparse matrices and arrays [Strout et al. 2003, 2016, 2014; Venkat et al. 2015], 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 6 elaborate upon and formalize these facets of PolyRec.

[1]Because we focus on perfect nesting, we do not handle fusion, though we do handle Sakka et al. [2017]'s code motion. We also do not handle the cases of Weijiang et al. [2015]'s Tree Dependence Analysis that require SMT reasoning. However, our compositional framework means that we do handle situations like combining code motion, inlining, and interchange, which no prior work does.

3.1 Running Example

Figure 2a shows a simple program using nested recursion. The "outer" recursion (the function defined in line 4) is actually a loop from 0 to N that has been transformed into a tail-recursion (this is how PolyRec handles loops). The "inner" recursion (defined in line 9) is a recursive function that traverses a binary tree. Within the inner recursion, an element of an array is accessed via the induction variable of the outer recursion and is used to update fields of the current node the inner recursion is visiting. Note that this example illustrates several aspects of the programs PolyRec targets: (i) nested recursion; (ii) *perfect* nesting—only the inner recursion performs any work, while the outer recursion merely enumerates an induction variable; and (iii) accesses to both regular and irregular data structures through the induction variables and global heap variables. Note that this example 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.

Figure 2c shows the result of applying a series of transformations using PolyRec (including some cleanup to produce more readable code). These transformations, combined, give similar behavior to applying point blocking [Jo and Kulkarni 2011] and traversal splicing [Jo and Kulkarni 2012]. 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.

To break down the transformations a little more concretely, first, the method outer was *strip mined* (Section 6.3.4) to break it into two loops, outer1 (at line 4) and outer2 (at line 18); outer2 performs groups of 4 iterations from outer1. Second, the method inner was changed from post-order (as it was in Figure 2a) to pre-order using *code motion* 6.3.1. Then the call inner(i, n.left, j) was *inlined* to operate on the left child before continuing recursion (resulting in two statements in inner and two new recursive calls) (Section 6.3.3) and code motion was applied again. Finally, inner and outer2 were *interchanged* (Section 6.3.2) so that inner iterates over the nodes in the tree before outer2 iterates over its 4-element block of A. PolyRec *can synthesize this complex series of transformations, and check it for soundness, in a single representation.*

3.2 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. This is analogous to the polytope representation of loop iteration spaces in the polyhedral model.

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 2b 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

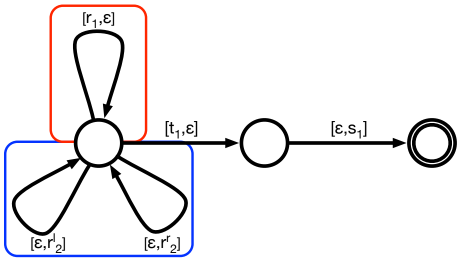
```
A[N] = /* initialize global array */;
node T = /* initialize tree */;

outer(int i, node n)
  if (i < N)
    inner(i, n) //$t_1$
    outer(i + 1, n) //$r_1$

inner(int i, node n)
  if (n != null)
    inner(i, n.left) //$r_2^l$
    inner(i, n.right) //$r_2^r$
    n.x += A[i] //$s_1$

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

(a) Repeatedly traversing a tree



(b) Multitape automaton for the above code.

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

outer1(int i, node n, int j)
  if (i < N)
    inner(i, n, j)
    outer1(i + 4, n, j)

inner(int i, node n, int j)
  outer2(i, n, j)
  if (n != null && n.left != null)
    inner(i, n.left.left, j)
  if (n != null && n.left != null)
    inner(i, n.left.right, j)
  if (n != null)
    inner(i, n.right, j)

outer2(int i, node n, int j)
  if (j < 4 && (i + j) < N)
    if (n != null)
      n.x += A[i + j]
    if (n != null && n.left != null)
      n.left.x += A[i+j]
    outer2(i, n, j + 1)

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

(c) After applying transformations.

Fig. 2. Running example

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.

3.3 Transformations

A scheduling transformation preserves *which* instances execute but merely restructures the iteration space so that the instances execute in a different order. In the polyhedral model, these transformations are represented as linear transforms of the iteration space polytope. In PolyRec, we represent a transformation as a *multitape finite state transducers* 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, which changes the order in which, say, 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 so that they sort in a different order. Alternately, we can (as we do in PolyRec) simply change the ordering. 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) to produce compound transformations.

3.4 Dependences

When transformation transducers are applied to the regular relations representing a PolyRec iteration space, the order (and hence schedule) of instances change (indeed, that is the point of the transducer!). Not all schedules are valid however: if two instances where one is dependent on the other change their order, the new schedule will produce incorrect results. Thus, to determine whether transformations are sound, PolyRec (conservatively) determines whether any pair of dependent instances can change their order when pushed through a transformation transducer.

PolyRec's representation for pairs of dependent instances is 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 $< R_\alpha, (R_\beta, R_\gamma) >$ 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:

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

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 determining whether a witness tuple is *preserved* by a transformation (intuitively, whether all pairs generated by the tuple preserve their order when transformed by a transducer), 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).

3.5 Code Generation

The final step of PolyRec is to generate code from a transformed program. This is straightforward: because each transformation on the iteration space automaton is analogous to a specific transformation on recursive code, and each transformation preserves the nested recursive structure of the code, we can simply apply the transformations one by one to the original code—each of these transformations has well-specified rewrites at the level of source code, as they have been presented in the literature before. Note that any individual transformation may be unsound (meaning that the code generator may temporarily produce incorrect code); PolyRec's dependence test guarantees

$$
\begin{aligned}
v \in \mathit{InductionVars} &::= i_1 \mid i_2 \mid \ldots \mid i_k \\
r \in \mathit{RecursiveMethods} &::= r_1 \mid r_2 \mid \ldots \mid r_k \\
h \in \mathit{HeapVars} &::= h_1 \mid h_2 \mid \ldots \\
\text{“Outer” recursions} &: r_j(i_1, i_2, \ldots, i_k)\{bd\} : p \\
\text{“Innermost” recursion} &: r_k(i_1, i_2, \ldots, i_k)\{ibd\} : p \\
p \in \mathit{BoundsCheck} &::= f_b(i_1, i_2, \ldots, i_k) \mid \textbf{true} \\
c \in \mathit{Calls} &::= r_j(i_1, i_2, \ldots, f_{invup}(i_j), \ldots, i_k) : p \\
t \in \mathit{TransferCall} &::= r_{j+1}(i_1, i_2, \ldots, i_k) \\
s \in \mathit{Stmts} &::= s_{comp}(i_1, i_2, \ldots, i_k, h_1, h_2, \ldots) : p \\
bd \in \mathit{Body} &::= c^*\ t\ c^* \\
ibd \in \mathit{InnerBody} &::= (s|c)^*
\end{aligned}
$$

(a) Language for defining tree traversal

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

r1(i, n) : (i < N)
  r2(i, n)
  r1(i + 1, n)

r2(i, n) : (n != null)
  r2(i, n.left)
  r2(i, n.right)
  scomp(n, i, A) //scomp = n.x+ = A[i]
```

(b) Expressing Figure 2a in core language

Fig. 3. Target language and example.

that the *composition* of all the transformations is, eventually, sound. This process produces the code in Figure 2c.

4 CORE LANGUAGE

Our eventual goal is for PolyRec to be able to handle all programs that manipulate arrays and pointer-based structures using loops and recursion. However, as a start, PolyRec operates on a subset of these programs: *perfectly nested* recursion. Informally, PolyRec targets programs with the following structure:

(1) The control structures are a set of nested recursive functions. Each function takes as arguments a series of induction variables, one associated with each recursive function. A given recursive function can do one of three things: (i) call itself recursively, manipulating its induction variable during the recursive call; (ii) perform a (read only) test to determine whether the recursion should end (i.e., whether the function should return); (iii) make a single call to a different recursive function, moving to a deeper level of the “nest”. Each recursive function should appear in this nest only one time (i.e., there is no mutual recursion). Analogizing with loops, operations (i) and (ii) are the equivalent of the loop bounds, and operation (iii) is the equivalent of executing the loop body.
(2) The *last* recursive function in the nest has no other recursive functions that it can call (due to the requirement of non-mutuality of recursion). This “innermost” function can execute other statements that allow the overall recursive structure to perform computations. Unlike “bounds check” statements, these computations can write to the heap. Note that these computations can depend on all the induction variables from all the recursive functions.

These restrictions mean that all the “real work” is performed in the deepest-nested recursive function, while all the other recursive functions work to provide different values for the induction variables (which may, for example, effectively traverse a tree or a list) that the statements in the inner function can use during computation. This is a direct analog for perfectly nested loops, where the outer loops enumerate the values of induction variables while the innermost loop performs the actual computation. *Indeed, the way that* PolyRec *handles loops is through a simple transformation to a tail-recursive function.* A perfectly-nested loop nest that is transformed in this way will form a recursion nest that matches the two restrictions from above.

To formalize these restrictions, Figure 3a provides the syntax of a simple core language that captures the nesting structure of k recursive methods. Figure 3b shows our running example expressed in this core language. We note a few quirks of our language:

(1) Functions, calls, statements, etc. are *predicated* by *bounds checks.*[2] Recursive methods are predicated with bounds that affect all statements and calls within the method. They are checked when the method is called and the method returns immediately if the check fails. (These are the "loop bounds," such as n != null.) Statements and calls can be further predicated with bounds checks (we do not write out predicates that are **true**). The bounds checks are pure functions of the induction variables in the program and do not read from memory that can ever be written to by statements in the recursion nest. (This is analogous to allowing loop bounds like i < N but not A[i] < B[j]).

(2) Our language does not have explicit return statements. Predication on the recursive method, or predication of individual statements and calls, can substitute for returns. This restriction simplifies our handling of some code transformations, as described in Section 6. It is not fundamental to PolyRec.

(3) Each "outer" recursive function, r_j with $j < k$ consists of a series of recursive calls (to itself), as well as exactly one *transfer* call that invokes the next recursive function, r_{j+1}.

(4) Each recursive call updates the recursive method's induction variable using simple uninterpreted pure functions called f_{invup} (for example, $\lambda i.\ i + 1$ or $\lambda n.\ n.right$). All other arguments to the recursion remain constant.
At this stage, we do not constrain what these functions do, because PolyRec's iteration space representation (Section 5 and transformations (Section 6) are independent of these updates. The specific form of these induction variable updates only affects the behavior of any dependence analysis, which PolyRec can be parameterized on.

(5) The innermost recursion, r_k can also perform computational statements. Computational statements are *compound* statements. They can, in fact, be a series of statements themselves, including with control flow. They can, unlike bounds checks, access writeable memory, allowing computation by storing intermediate results in the heap. As a result, the innermost recursive method (the only one that contains computational statements) can be thought of as a sequence of compound statements and recursive calls. This treatment is similar to that of Sakka et al. [2017].
The exact structure of the computations performed by a computational statement only affects the behavior of any dependence analysis PolyRec is instantiated with (Section 7), so for now we leave them as uninterpreted operations, s_{comp}.

5 REPRESENTING RECURSIVE ITERATION SPACES

The unsurprising first step in any scheduling framework is designing a representation to capture the schedule. In traditional scheduling settings, like instruction scheduling, representations like Directed Acyclic Graphs of instructions suffice. However, in instance-wise settings, where we are concerned with the dynamic instances of static statements, more sophisticated representations are required: DAG representations rely on using the static instructions to "name" each operation, while in instance-wise analyses we must find a way to name each dynamic operation. In the case of the polyhedral model [Bondhugula et al. 2008; Feautrier 1992a,b], this representation takes the form of a *polytope*. The polytope is a system of linear inequalities across multiple dimensions, where each dimension represents one loop in the nest, and the inequalities capture the loop bounds. The

[2]Note that the code in Figures 2a and 2c do not use predication, but rather use explicit if statements—this is a straightforward desugaring that we apply for exposition purposes.

integer points within that polytope represent the individual instances, and they can be named with a multidimensional vector. The order of computations is represented by a simple lexicographic ordering of these vectors.

In the case of recursive programs, the story is not quite so simple. Amiranoff et al. [2006] use *control words* to name instances of recursive programs. They transform a recursive program into a context-free language (using the straightforward mechanism of replacing functions with non-terminals and statements with terminals, plus some special handling for loops) that generates strings. Each string naturally captures a trace of one possible execution of the program. Because each dynamic instance of an operation in a program can only be produced by one possible trace through the program, these strings uniquely name each instance in the program. By carefully choosing the alphabet, ordering on the control words is also lexicographic ordering.

While Amiranoff et al. [2006]'s work seems sufficient for handling our problem of naming and ordering the instances of nested recursive programs, it is not quite suitable for our setting, since, unlike them, we are interested in transformations of nested recursive spaces. 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 automata are not suitable for applying transformations that focus on particular dimensions or the interaction between dimensions (as we will see in Section 6). To overcome this problem, PolyRec uses a novel representation, based on regular relations that are generated by *non-deterministic multitape finite automata* [Rabin and Scott 1959].

5.1 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, $v = [v_1, v_2, \dots, v_k]$ and $w = [w_1, w_2, \dots, w_k]$, their elementwise concatenation, $v \odot w$ is $[v_1 \cdot w_1, v_2 \cdot w_2, \dots, v_k \cdot w_k]$.

We can thus define:

Definition 5.1. A *non-deterministic, k-tape finite automaton* is a 6-tuple $A =< k, \Sigma, Q, q_0, F, E >$ 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 $q_0 a_1 q_1 a_2 q_2 \dots a_n q_n$ where q_0 is the initial state, $q_n \in F$, for each $0 < i \leq n$, $< q_{i-1}, a_i, q_i > \in E$, 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 [Kaplan and Kay 1994].

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*:

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

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.

5.2 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 in the source code. The intuition behind PolyRec's labeling is straightforward: we are interested in naming the instances of statements that perform computation that could incur dependences (i.e., any statement that reads or writes from the heap)—statements that cannot incur dependences are irrelevant, since they can be executed in any order without affecting correctness.[3] Each statement that executes does so at a unique combination of call stack and static statement location (since our language replaces loops with recursion). Hence, 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, $< k, \Sigma, Q, q_0, F, E >$, 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.

[3]Note that we do not consider control dependences in this paper. Control dependences are either part of the predicates on statements or embedded in control structures that are part of compound statements in the innermost recursion. In the former case, our language does not allow predicates that depend on writeable heap data, so this control dependence is independent of the order of computation, while in the latter case, the scope of control dependence does not escape outside the compound statement, so can be ignored.

- 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.[4]
  - $\{< q_0, \ell_i[r_i^j], q_0 > | 0 < i \leq k \wedge 0 < j \leq |C_i|\}$. 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.
  - $\{< q_{i-1}, \ell_i[t_i], q_i | 0 < i < k\}$. In other words, a sequence of edges from $q_0 \ldots q_k$ labeled with the transfer calls. Each call only adds a symbol to the dimension of its recursion level.
  - $\{< q_k, \ell_k[s_i], q_{s_i} > | 0 < i \leq |S|\}$. In other words, a transition from q_k to the state associated with each compound statement, labeled with that compound statement's symbol.

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.

Note that 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)—and, indeed, is the only sound thing to do if we cannot reason about recursion termination.

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. [2006]. Amiranoff and Cohen note that while they use context-free languages to generate their control words, for their programs, the languages are actually regular.*

We note that all the automata we build 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. Indeed, because of the close correspondence of this structure to the call structure of nested recursion, *any* relation R that generates the iteration space for perfectly nested recursion can be captured with an automaton of this structure.

6 REPRESENTING SCHEDULING TRANSFORMATIONS

Armed with a representation for the iteration space, described in Section 5, 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. Scheduling transformations in the polyhedral framework are represented using transformations of the iteration space polytope [Bondhugula et al. 2008]; transforming one polytope into another creates new loop bounds (because the polytope boundaries change) and

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

a new schedule (because the lexicographic enumeration order of the points inside the polytope changes). Interestingly, these transformations can even change the number of dimensions of the polytope to implement optimizations such as strip mining and tiling.

PolyRec uses *multitape finite state transducers* [Kaplan and Kay 1994; Rabin and Scott 1959] to represent its transformations (Section 6.1). Transducers are naturally composable, meaning that PolyRec can synthesize compound transformations that apply multiple rewrites to a recursive loop nest, analogous to multiplying multiple transformation matrices to synthesize a single polytope transformation in the polyhedral model (Section 6.2). Finally, PolyRec provides support for generating transducers that apply specific transformations to nested recursion: code motion, inlining, strip mining, and interchange (Section 6.3).

6.1 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:

- 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.
- 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.
- Scheduling transformations should preserve perfect nesting: if an iteration space is perfectly nested, applying the transformation should result in a new perfectly nested space.[5] 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.
- Transformation transducers should be *order-free* rewrites. This is a structural restriction that we place to support a decidable dependence test (Section 7). 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[6]; 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. This general structure is easy to maintain for transformations that preserve perfect nesting. Figure 4a gives an example of what one of these order-free rewrites looks like. Note that both "looping" states have rewrites for $[r_1, \varepsilon]$.

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.

[5]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.
[6]Either directly or through an ε transition.

A multi-tape transducer is just a specific way of thinking about a multi-tape automaton where some subset of the tapes are consider "input" tapes—the multi-dimensional string inputs to the transformation—and another subset of the tapes are considered "output" tapes—the multi-dimensional string outputs to the transformation. As with all multi-tape finite automata, these transducers define regular relations.

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 $T : (\Sigma \cup \varepsilon)^k \longrightarrow (\Sigma' \cup \varepsilon)^k$ 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 [Griffiths 1968]. However, we build our transformations in a constructive manner that makes it straightforward to see that T acts as expected.*

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):

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

This transition rewrites the symbols $[i_1, i_2]$ from the input tape into $[j_1, j_2]$ on the output tape (alternatively, we could view this as a *four* tape transition with the label $[i_1, i_2, j_1, j_2]$).

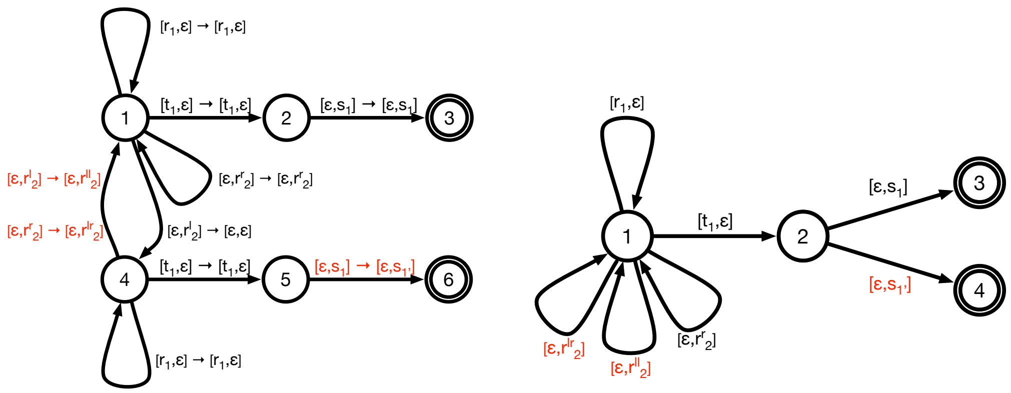
Applying a transformation transducer is straightforward: we simply project out the output dimensions of the transducer, and apply ε elimination to simplify the resulting multitape automaton. The result is an automaton that produces the *transformed* iteration space. Note, though, that it is important 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[7]. It is this information that allows us to check the soundness of schedules (Section 7). 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 6.3 all do this translation faithfully, although in general there is no way for us to automatically verify this for an arbitrary transformation.

Example. Figure 4a shows a transformation transducer that inlines the call inner(i, n.left) from Figure 2a, with Figure 4b representing the transformed iteration space. (Section 6.3.3 explains how this transducer would get constructed) Consider what the actual inlining transformation does: inner(i, n.left) got replaced with n.left.x += A[i] : n.left != null; inner(i, n.left.left) : n.left != null; inner(i, n.left.right) : n.left != null. In the iteration space, we see how that gets captured with, for example, $[t_1, r_2^l r_2^r s_1]$ being rewritten to $[t_1, r_2^{lr} s_1]$.

6.2 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 6.3 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.

[7]By way of analogy, in the polyhedral model, an interchanged iteration space polytope can easily be formulated; but the fact that interchange is produced by a specific linear transformation also captures how specific instances are mapped from the original iteration space to the interchanged space.



(a) Transducer implementing inlining of inner(i, n.left).

(b) Projecting the output tapes of Figure 4a, yielding a new iteration space automaton.

Fig. 4. Inlining using transducers.

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.

We note two important things about this process. First, every time a transformation transducer is applied to an iteration space automaton, we can reflect that transformation in the original input code (e.g., by applying the usual inlining transformation to a piece of code). In this way, the PolyRec transformation framework keeps the iteration space automaton and the underlying code in sync with each other: as we apply transducers to generate new automata, we apply regular code transformations to generate the corresponding code (Section 8 elaborates on this approach).

Second, 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 5.1). Hence, after a sequence of transformations are applied, in addition to the transformed code, 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 7, this is sufficient information for PolyRec to test whether dependences are violated.

6.3 Representing Specific Transformations

In previous sections we have presented the machinery to represent and compose transformations. In this section we provide four well-defined transformations—code motion, recursion interchange, inlining, and strip mining—and methods to construct their respective transducers. These basic transformations are simple, but powerful enough to construct transformations such as *point blocking* [Jo and Kulkarni 2011] (a combination of strip mining and interchange) and *traversal splicing* [Jo and Kulkarni 2012] (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. Transformations are applied one at a time in PolyRec (so each transducer is built

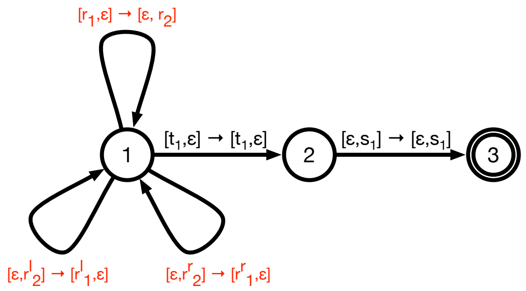


Fig. 5. Transducer implementing interchange of first and second dimensions.

starting from a perfectly-nested iteration space), but the composition of transducers is used to check transformation soundness (Section 7).

6.3.1 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. Note that 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. Because PolyRec represents the lexicographic order separately, it does not explicitly build a transducer for code motion; it just records the new lexicographic order. Section 3.3 gave an example of how code motion might be used to change a post-order recursion to a pre-order recursion.

6.3.2 Interchange. Interchange is a seemingly-complex transformation—changing the nesting order of recursion—with a simple corresponding transducer. As a code transformation, interchange is well-understood for loops [Allen and Kennedy 1984], and interchange of loops and more general recursion [Jo and Kulkarni 2011] and general recursive methods [Sundararajah et al. 2017] have been studied in the literature. The transducer for *interchange* is straightforward to construct. Figure 5 shows the transducer for interchanging first and second dimensions of the code in Figure 2a. 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. Note that this transformation is obviously order-free as defined above. Like code motion, interchange changes the schedule of computation, so can break dependences.

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

6.3.3 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 [Jo and Kulkarni 2012]. 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 4a).

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}.

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 $(r_{ll}|r_{rl}|r_r)^*(s_i|s'_i)$.

6.3.4 *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 are loop-like* (i.e., they only have 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 [Wolfe 1989]; it also appears in transformations for recursive codes [Jo and Kulkarni 2011, 2012]. 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. In other words, strip mining is essentially inlining a "linear recursion" (recursion expressible as an affine loop) multiple times and expresses the inlined piece of code as a new linear recursion which get called from the original one. In the below presentation, *strip size* is the number of times original recursion gets inlined. A general procedure for constructing strip mining transducer is parameterized over the strip size. Following steps are taken to construct the transducer in Figure 6:

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 new transfer statement for the newly constructed dimension due to strip mining; we add the transition $[\varepsilon, \varepsilon] \longrightarrow [t_1, \varepsilon, \varepsilon]$ to every replica.
Adding entering transition 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: $[r_1, \varepsilon] \longrightarrow [\varepsilon, r_2, \varepsilon]$.
Adding exiting transition We add the transition $[\varepsilon, \varepsilon] \longrightarrow [r_1, \varepsilon, \varepsilon]$ from the last replica to the first. Essentially this transition represents the newly constructed recursive dimension.

This transducer is fairly tricky. Intuitively, it creates a new dimension representing the "outer" part of the strip-mined loop which executes once every strip-size calls. Note that the transducer does not directly capture the size of the strip in the "inner" part of the strip-mined loop—PolyRec

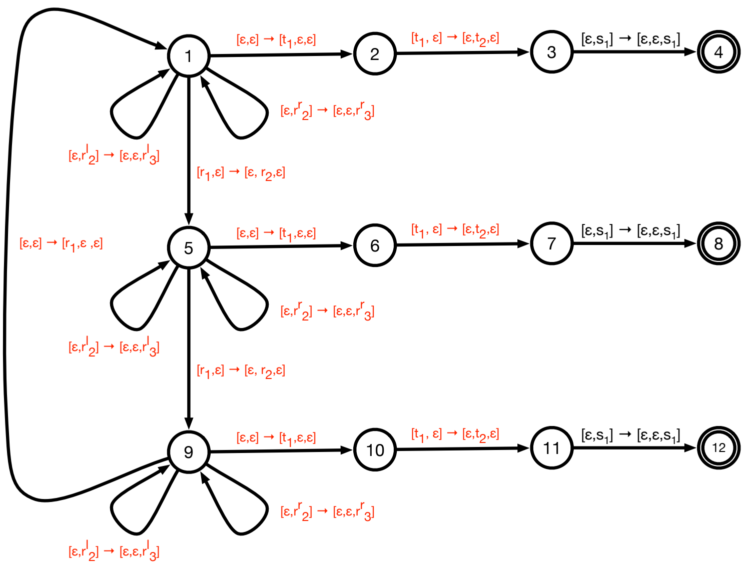


Fig. 6. Transducer implementing strip size of two on first dimension.

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.

7 REPRESENTING DEPENDENCES AND CHECKING SOUNDNESS

The previous section presented an approach for transforming perfectly nested recursive programs into other perfectly nested programs by chaining together a series of simple transformations. However, the transformation strategy we presented does not necessarily produce *sound* transformations: 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 3b. 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 5 and 11. However, doing so means that the updates to each tree node's n.x field in line 11 will occur in the opposite order of the original program, potentially changing the result of the program (e.g., if the addition in line 11 were floating point). Just because PolyRec can synthesize a transformation does not mean that it is legal.

The general principle behind any scheduling transformation (whether it be something as simple as instruction scheduling or as complicated as the polyhedral model) is that 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 (i) 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—or (ii) j is control dependent on i [Ferrante et al. 1987].[8] 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?

[8]Note that the restrictions we place on our core language means that we need not consider control dependences.

7.1 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 [Kennedy and Allen 2002] to dependence polytopes [Bondhugula et al. 2008]. In the world of irregular programs, Amiranoff et al. [2006] uses a series of *dependence transducers* to represent and reason about dependences in irregular programs.

The representation PolyRec uses to capture the potentially infinite set of dependences is *witness tuples*, a rough analog of distance vectors.

Definition 7.1. 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 $< R_\alpha, (R_\beta, R_\gamma) >$ such that:

(1) $R_\alpha \odot R_\beta \in R_{\mathcal{A}}$ and $R_\alpha \odot R_\gamma \in R_{\mathcal{A}}$ (essentially, 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 this means 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 7.2. A witness tuple $< R_\alpha, (R_\beta, R_\gamma) >$ *captures* a set of dependences $D \in R_{\mathcal{A}} \times R_{\mathcal{A}}$ (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. For example, the following set of witness tuples captures all pairs of instances from the iteration space in Figure 2b:

$$\begin{array}{rll} < [(r_1)^*, \varepsilon], & ([t_1, (r_2^l|r_2^r)^* s_1], & [(r_1)^+ t_1, (r_2^l|r_2^r)^* s_1]) > \\ < [(r_1)^*, (r_2^l|r_2^r)^*], & ([t_1, r_2^l(r_2^l|r_2^r)^* s_1)], & [t_1, s_1|(r_2^r(r_2^l|r_2^r)^* s_1))]) \\ < [(r_1)^*, (r_2^l|r_2^r)^*], & ([t_1, r_2^r(r_2^l|r_2^r)^* s_1)], & [t_1, s_1]) \end{array}$$

Note that this paper does not focus on procedures to generate witness tuples, as PolyRec is agnostic to *where* the witness tuples come from. Many prior works present dependence analyses that could be modified to produce witness tuples [Amiranoff et al. 2006; Rajbhandari et al. 2016b; Sakka et al. 2017; Weijiang et al. 2015]. Section 7.3 sketches a possible procedure for generating witness tuples for some recursive programs. The next section explains how witness tuples, regardless of how they are obtained, can be used to verify that a transformation is sound.

7.2 Checking Soundness

Once we have represented the dependences in the program with one or more witness pairs, we can use these pairs to test for transformation soundness. The basic premise of the approach is as follows: 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 using the following approach. For a given witness tuple $< R_\alpha, (R_\beta, R_\gamma) >$, 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:

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) $E' = E \cup \{< q_0', \varepsilon^k, q_i | q_i \in Q_w\}$. 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.

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 6) 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 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 α^* on the first tape. We repeat this process for the second tape, and so on. For $R_{w,\gamma}$, we create the lexicographically *earliest* k-string in a similar fashion, looking for the earliest string on each 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 a tape or loop, consuming some infinite string α^*. 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 $R_{w,\gamma}$.

Note that thus far, we have only shown that the order is preserved for one prefix w from the witness tuple. However, we can enumerate all possible prefixes w, discarding any prefixes that lead to the same set of states Q_w during the construction of $\mathcal{A}_T^w$ (since that prefix will give the same behavior as one already explored). Hence, we can for example extend the first dimension of the prefix until we repeat a set of Q_w states, then extend the second dimension of the prefix and restart the first from ε, and so forth. Because there is a finite set of states Q in $\mathcal{A}_T$, there is a finite set of differentiated prefixes that this enumeration will eventually find. Thus, the overall process of determining if dependences are preserved 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.

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

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

We will use this witness tuple to check whether the inlining transformation from Figure 4a is sound. 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.

7.3 Generating Witness Tuples

Any dependence analysis that can generate witness tuples can be used with PolyRec. This section sketches out how the dependence analysis from Weijiang et al. [2015] can be used to generate witness tuples for programs that operate over trees. (A similar procedure could be used to generate witness tuples from the analyses from Rajbhandari et al. [2016b] and Sakka et al. [2017].)

Weijiang et al. [2015] target programs where recursive functions are used to traverse tree structures; in other words the induction variables represent nodes in the tree (the f_{invup} function for a recursive call might look like $\lambda n.n.left$) 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 [1988]'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 then 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).

Note that in general, generating witness tuples is akin to identifying *distance vectors* between instances in the polyhedral approach. This means that if operations on induction variables are linear, with affine access functions (for example, n.x[i] where i is an induction variable), we can combine an analysis in the style of Weijiang et al. [2015] with an analysis to find distance vectors (using, e.g., the Omega calculator [Pugh 1991]) to build more discriminating witness pairs for additional dimensions. In this way, PolyRec can handle programs that use a combination of trees and arrays. We leave a full development of this style of dependence analysis to future work.

8 CODE GENERATION

PolyRec provides a systematic code generation strategy to build complex transformations that are composed of basic transformations. To do so, PolyRec maintains three representations: the code, C, the iteration space automaton $\mathcal{A}$, and the transducer representing the composition of all transformations thus far applied to generate $\mathcal{A}$, $\mathcal{A}_{T^*}$, which begins as the identity transducer.

To apply a transformation, PolyRec (a) builds the specific transducer $\mathcal{A}_T$ that transforms the iteration space according to the transformation (Section 6.3); (b) applies $\mathcal{A}_T$ to $\mathcal{A}$ and projects out the output tapes to derive the transformed iteration space, $\mathcal{A}'$; (c) updates $\mathcal{A}_{T^*}$ to $\mathcal{A}_{T^*} \circ \mathcal{A}_T$ (capturing the updated composed transformation); and (d) *directly applies the code transformation to* C. At the end of this process, PolyRec still has three representations: the new code C', the iteration space $\mathcal{A}'$ that represents C', and the transducer A_{T^*} representing *all the transformations needed* to transform the original iteration space $\mathcal{A}$ to $\mathcal{A}'$. This process is repeated for each transformation.

Note that throughout this process, the code C is kept in sync with the iteration space automaton $\mathcal{A}$—PolyRec does not attempt to synthesize code from the iteration space automaton (as would be analogous to polyhedral code generators [Bastoul 2004]). Note, too, that each transformation preserves the perfectly nested structure of the code and the iteration space. That means that each *subsequent* transformation can be applied without considering what transformations were performed earlier, simplifying the code generation process—the basic transformations PolyRec supports are all variants of known code transformations—and the process of constructing transformation transducers.

Once all the desired transformations have been applied—and only then—PolyRec performs the dependence test (Section 7) on $\mathcal{A}_{T^*}$ to see whether the overall composed transformation is sound. Because PolyRec waits until all transformations have been applied before checking soundness, it is possible that some intermediate states of the code actually represent *unsound* transformations that are later fixed by other transformations. (For example, interchange can break dependences that are fixed by code motion that changes the recursion order of a function, analogous to situations where loop interchange in regular codes breaks unless loop reversal is also applied.)

Note that while the transformations PolyRec supports exist in the literature, we place some syntactic restrictions on when they can be applied, to ensure that the transformed code can still be expressed using the core language of Section 4. We now briefly describe how code generation for the various transformations is performed.

Code motion Code motion is applied as expected. The guards on a statement move with the statement. We do not need to consider control dependences on the guards as they do not read from memory writeable by the recursion nest.

Interchange PolyRec uses a generalization of interchange that works for any perfectly-nested recursion [Sundararajah et al. 2017]. Note that this general transformation uses complex bounds check code in certain situations that breaks perfect nesting [Sundararajah et al. 2017, Section 4]. Hence, PolyRec only performs interchange for recursion when the guards on the statements for levels at *or above* the deepest recursion being interchanged all depend only on the current level they are at.

Inlining Inlining is the standard inlining procedure. PolyRec will only inline innermost recursive calls. Note that the functional form of induction variable updates makes it straightforward to perform inlining (by composing the update function for inlined code).

Strip mining Strip mining only applies to *linear* recursion, and hence is applied as it is in standard loop transformations.

9 EVALUATION

We evaluate PolyRec on nested traversals similar to the ones used in prior work [Jo and Kulkarni 2011, 2012; Sundararajah et al. 2017]. Our focus of this case study is to show that PolyRec is capable of performing the types of scheduling transformations for nested recursive code structures proposed as specific transformations in literature, but in a unified manner.

Prototype Implementation. Since our framework has well-defined rewrite rules to realize all basic transformations, we implemented it as a source-to-source translator in Clang. The prototype takes a program written in C with annotations indicating the nested recursive structure. The prototype also takes in a configuration file with an order of transformations to perform. In lieu of performing dependence analysis (as we do not contribute a new dependence analysis, as explained in Section 7.3), the witness tuples are provided in a configuration file.

The prototype (i) lifts the C code to the automaton abstraction; (ii) constructs and composes transducers for transformations as mentioned in Section 6.3; and (iii) generates code as mentioned in Section 8. At the end it performs the soundness check using the final transducer and the witness tuple(s) provided. If the provided witness tuples pass the dependence test, our prototype outputs the transformed code.

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.

Case Study. We have evaluated PolyRec on four cases, all of which have at least one general recursion. The first three cases use a recursive traversal over a tree structure nested within a loop (which is rewritten into tail-recursion prior to transformation); this is the computation structure examined in several previous papers [Jo and Kulkarni 2011, 2012; Zhang and Chien 1997]. 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. [2017]. In each of the cases, we vary the dependence structure through different instantiations of the leaf statements. We perform *Code Motion* (CM), *Inlining* (IL), *Interchange* (IC) and *Strip Mining* (SM) when possible and we report runtimes and cache behavior of the original and changed codes. For space reasons, we describe only the dependence structure of the cases here. Appendix A of the supplemental material includes pseudocode, detailed dependence information, and transformation examples for the various cases. Unless otherwise specified, the tree(s) traversed have $2^{20} - 1$ nodes, loops run for 1,000 iterations, and strip mining uses a strip size of 100.

Table 1. Performance Results of the Case Study

Case	Transform	Runtime(s)	Normalized L2 Cache Access	Normalized L2 Cache Miss
Case 1	Baseline	79.95	1.00	1.00
	IC	0.96	0.03	0.01
	IC-SM-CM	0.59	0.01	0.01
	IC-SM-IL-CM	0.62	0.01	0.01
	IL	93.38	0.99	0.99
Case 2	Baseline	61.61	1.00	1.00
	IC	1.04	0.02	0.01
	IC-SM	2.92	0.04	0.04
	IC-SM-IL	0.77	0.01	0.01
	IL	54.08	0.98	0.98
Case 3	Baseline	61.48	1.00	1.00
	IC-CM	1.05	0.02	0.01
	IC-SM-CM	3.04	0.04	0.03
	IC-SM-IL-CM	0.56	0.01	0.01
Case 4	Baseline	5.89	1.00	1.00
	IC	2.92	0.98	0.00

Case 1 In each instance, 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. 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 In each instance, the computation is of the form n.x[i] = n.l.x[i] + n.r.x[i], where *i* is the induction variable of the loop. Here, there are no dependences across the loop, but each instance depends on the computation at its parent in the tree. Here, some code motion is not legal because it may cause the tree nodes to be visited in the wrong order. The dependence structure in this case and the previous one can be analyzed using Tree Dependence Analysis, as discussed in Section 7.3.

Case 3 In this case, the updates create dependences across both recursions: n.x[i] += n.left.x[i + 1] + n.right.x[i + 1]. The dependence structure prevents interchange from occurring if the inner recursion is post-order, but code motion can change the recursion to pre-order, at which point interchange is legal. This type of code can be analyzed by the extended dependence analysis that handles trees and arrays described in Section 7.3

Case 4 In this case, we primarily show that PolyRec can transform general nested recursion. We know of no existing dependence analysis that can automatically analyze such codes. For this special case, the other general recursion traverses a tree of size $2^{10} - 1$.

Table 1 summarizes the results of applying various transformations to the case studies. We see that in cases 1–3, interchange improves performance dramatically. This is partly due to increased locality and partly due to the fact that once the loop is moved to the inner level of the nest, ICC is able to recognize it as a dense loop and perform additional optimization. In case 4, interchange causes a substantial improvement in locality, but because both levels are general recursion, it does not benefit from additional optimizations by ICC. We note that in all cases, composing additional transformations such as inlining or code motion either *enables* interchange (in case 3, without code motion interchange is illegal) or enhances interchange (in all three cases where we perform composed transformations, the best performance is derived from combining multiple transformations). Notably, IC+SM in case 1 is equivalent to point blocking [Jo and Kulkarni 2011], and IC+SM+IL in cases 2 and 3 is similar to traversal splicing [Jo and Kulkarni 2012]. Note that transformations do not always improve performance—e.g., adding strip mining to interchange in cases 2 and 3—emphasizing the utility of being able to explore a large space of transformations with PolyRec.

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 framework for recursive programs that provides an end-to-end strategy for representing programs, transformations of those programs, and dependences in those programs, allowing for nested recursive programs (including combinations of recursion and loops) to be soundly transformed. This paper showed that PolyRec, through the composition of simple transformations, is able to represent and check fairly complex transformations from the recent literature, and yield substantial performance improvements.

REFERENCES

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

Pierre Amiranoff, Albert Cohen, and Paul Feautrier. 2006. Beyond Iteration Vectors: Instancewise Relational Abstract Domains. In *Static Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 161–180.

Cedric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. IEEE Computer Society, Washington, DC, USA, 7–16. https://doi.org/10.1109/PACT.2004.11

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

Albert Cohen and Jean-François Collard. 1998. Instance-Wise Reaching Definition Analysis for Recursive Programs Using Context-Free Transductions. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*. IEEE Computer Society, Washington, DC, USA, 332–. http://dl.acm.org/citation.cfm?id=522344.825716

Paul Feautrier. 1992a. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International Journal of Parallel Programming* 21, 5 (01 Oct 1992), 313–347. https://doi.org/10.1007/BF01407835

Paul Feautrier. 1992b. 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

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, London, UK, UK, 470–479. http://dl.acm.org/citation.cfm?id=646663.700133

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

Rakesh Ghiya, Laurie J. Hendren, and Yingchun Zhu. 1998. *Detecting parallelism in C programs with recursive data structures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 159–173. https://doi.org/10.1007/BFb0026429

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

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

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

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

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

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

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

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

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

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

William Pugh and David Wonnacott. 1994. *Nonlinear Array Dependence Analysis.* Technical Report. College Park, MD, USA.

M. O. Rabin and D. Scott. 1959. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development* 3, 2 (April 1959), 114–125. https://doi.org/10.1147/rd.32.0114

Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noel Pouchet, Fabrice Rastello, Robert J. Harrison, and P. Sadayappan. 2016a. 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

Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, Robert J. Harrison, and P. Sadayappan. 2016b. 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

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

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

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

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

M. M. Strout, F. Luporini, C. D. Krieger, C. Bertolli, G. T. Bercea, C. Olschanowsky, J. Ramanujam, and P. H. J. Kelly. 2014. Generalizing Run-Time Tiling with the Loop Chain Abstraction. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 1136–1145. https://doi.org/10.1109/IPDPS.2014.118

Kirshanthan Sundararajah, Laith Sakka, and Milind Kulkarni. 2017. Locality Transformations for Nested Recursive Iteration Spaces. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 281–295. https://doi.org/10.1145/3093337.3037720

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

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

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

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

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

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 & Practice of Parallel Programming (PPOPP), Las Vegas, Nevada, USA, June 18-21, 1997*. 37–47. https://doi.org/10.1145/263764.263771