UniRec: A Unimodular-Like Framework for Nested Recursions and Loops

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

Scheduling transformations reorder operations in a program to improve locality and/or parallelism. There are mature loop transformation frameworks such as the polyhedral model for composing and applying *instance-wise* scheduling transformations for loop nests. In recent years, there have been efforts to build frameworks for composing and applying scheduling transformations for nested *recursion* and loops, but these frameworks cannot employ the full power of transformations for loop nests since they have overly-restrictive representations. This paper describes a new framework, UniRec, that not only generalizes prior frameworks for reasoning about transformations on recursion, but also generalizes the unimodular framework, and hence unifies reasoning about perfectly-nested loops and recursion.

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

Additional Key Words and Phrases: Recursion, Scheduling Transformations, Skewing

ACM Reference Format:
Kirshanthan Sundararajah, Charitha Saumya, and Milind Kulkarni. 2022. UniRec: A Unimodular-Like Framework for Nested Recursions and Loops. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 170 (October 2022), 27 pages. https://doi.org/10.1145/3563333

1 INTRODUCTION

Over the past several decades, researchers have investigated a wide variety of strategies for transforming programs to improve their locality or parallelism. The basic approach adopted by these techniques is to *reschedule* when various computations happen so that different operations that touch the same data occur close together in time (improving locality) or so that independent operations are available to execute simultaneously (improving parallelism). As a result, these types of transformations are called *scheduling* transformations.

Most work on scheduling transformations has focused on *regular programs*—programs consisting of nests of loops operating over arrays and matrices. These programs contain predictable control flow and data access patterns, making them attractive targets for compile-time optimization. Researchers have proposed a whole catalog of scheduling transformations for regular programs, including loop interchange, loop reversal, loop tiling, loop fusion, among others [Kennedy and Allen 2002]. These transformations have been unified into frameworks, such as the unimodular framework [Banerjee 1991] and the polyhedral framework [Bondhugula et al. 2008; Feautrier 1992a,b], that allow individual transformations to be composed into more complex rescheduling schemes. Unfortunately, all of these frameworks only apply to programs that deal with dense loops and dense arrays and matrices.

Authors' addresses: Kirshanthan Sundararajah, Purdue University, USA, ksundar@purdue.edu; Charitha Saumya, Purdue University, USA, cgusthin@purdue.edu; Milind Kulkarni, Purdue University, USA, milind@purdue.edu.

2475-1421/2022/10-ART170
https://doi.org/10.1145/3563333

In recent years, there has been interest in developing transformations for *irregular programs*—programs that use a mix of recursion and loops to operate on lists, trees and graphs—-to enhance locality and expose parallelism by *restructuring* their computation schedules [Jo and Kulkarni 2011, 2012; Rajbhandari et al. 2016a,b; Sakka et al. 2017; Sundararajah et al. 2017]. The advent of transformations for programs that mix recursion and loops has spurred interest in inventing approaches and general frameworks for composing and applying these transformations [Kobeissi et al. 2020; Sundararajah and Kulkarni 2019], just as the unimodular and polyhedral frameworks compose and apply transformations on regular programs.

The key challenge of composing scheduling transformations is that these transformations may not always be safe. If there are *dependent* operations—where one operation writes to a memory location and another operation reads from or writes to the same memory location—and if the transformation does not maintain the order of execution of these dependent operations as in the original program then the transformation is unsound. Moreover individual transformations of a program may violate the correctness of the program but *composing* those transformations may yield a correct program. Hence, performing safe scheduling transformations necessitates a framework for reasoning about their composition.

In the world of loops, the polyhedral [Feautrier 1992a,b] and unimodular frameworks [Banerjee 1991] reason about composition of transformations by having a unified representation for the schedule of the computations and dependences between those computations. But these frameworks reason only about schedules that arise from loop nests and dependences that arise from affine indexing into arrays; they cannot reason about the irregular control flow that arises from recursion, or the complex dependences that arise from pointer-based data structures like trees and lists.

A recently introduced framework, PolyRec [Sundararajah and Kulkarni 2019], uses a different unified representation of schedules and dependences to reason about transformations of programs that use a mix of recursion and loops (e.g., n-body tree codes that use loops over recursive tree traversals). But while PolyRec's unified representation can capture many classic transformations of perfectly nested loops—loop interchange, reversal, and unrolling—and the general forms of these transformations that apply to perfectly-nested recursive structures—code motion, recursion interchange, and inlining—it is not powerful enough to handle other classical loop transformations such as *loop skewing* that *also* apply to perfectly nested loops (Crucially, loop skewing is a necessary transformation for a *complete* loop transformation framework that can handle arbitrary transformations of perfectly-nested loops). Moreover, whether there even *are* recursive analogs of loop skewing that can be integrated into a common framework is left unexplored, though some recent work has shown promise in defining a recursive variant of skewing [Iannetta et al. 2021].

No existing framework can handle the generality of transformations on perfectly-nested iteration structures: the unimodular and polyhedral frameworks cannot handle irregular, recursive programs, while PolyRec cannot handle loop skewing. For instance, consider the case of a loop that repeatedly traverses a tree, and when each tree node is visited, a second loop executes. In other words, a program that consists of a loop nested within a recursive function that is, in turn, nested within a loop. Loop transformation frameworks will not be able to reason about any transformation on this nest because of the presence of the recursion. PolyRec can reason about *some* transformations on this nest, such as an interchange of the outer loop and the tree traversal. But after the interchange if there is an opportunity to exploit parallelism through skewing the resulting doubly nested loops then, PolyRec will fail as it is not capable of handling such loop transformations.

Contributions In this paper, we introduce UniRec, which generalizes PolyRec to the entire known catalog of transformations on perfectly nested iteration structures, whether they be loops or recursive functions.[1] In addition to the transformations PolyRec already handles, UniRec supports loop skewing, including introducing a new, generalized form of skewing that also applies to recursive functions. The specific contributions we make are:

- **A new iteration space representation** Reasoning about scheduling transformations requires an *instance-wise* representation of a program—capturing the dynamic instances of each operation. The collection of all these instances is known as the *iteration space*. A scheduling transformation is a one-to-one mapping between iteration spaces. PolyRec represents iteration spaces using *multi-tape finite state automata* (as explained in Section 2). However, this representation does not support scheduling transformations that involve multiple dimensions of a loop nest at a time, such as skewing, which requires considering an instance's position along multiple dimensions in the iteration space. UniRec introduces a new, *extended multi-tape finite state automata* representation of iteration spaces that supports transformations that consider the arithmetic combination of multiple dimensions. (Section 4.1.)
- **A new definition of loop skewing** Loop skewing is a classical loop transformation that shifts when iterations of an inner loop occur based on which iteration of the outer loop is currently executing (or vice versa) [Wolfe 1986]. We show how this transformation—which makes the unimodular framework complete—can be captured in the automata-based representation of UniRec through the use of *multi-tape transducers*. We further define a *general skewing transformation* that can apply, in some cases, to tree-shaped iteration spaces, rather than just affine spaces; classical loop skewing is thus just a special case of general skewing. (Section 4.2.)
- **A dependence test for composed transformations** One of PolyRec's key contributions is a test for when composed transformations violate dependences, and hence the composition is unsound. The decidability of this dependence test relies on the very carefully constructed set of restrictions that PolyRec imposes on iteration spaces and transformations, making it unclear whether the dependence test can apply to more complex scenarios. We show how this dependence test can be generalized to UniRec's more complex iteration space representation and more general set of transformations while still remaining decidable. Thus, UniRec can generate composed sequences of transformations, including skewing, and verify that the composed sequence of transformations is still sound. (Section 5.1.)

We build a prototype implementation of UniRec that can compose transformations with our new extended representation of instances and verify their soundness.

UniRec fully subsumes both PolyRec and the unimodular framework, and shows, for the first time, that the complete set of scheduling transformations on perfectly-nested programs can be represented, reasoned about, composed, and checked for soundness in a single framework.

2 BACKGROUND

This section provides a brief background of PolyRec and discusses the inability of PolyRec's representation to capture transformations such as skewing.

2.1 Scheduling Transformations and Instance-wise Analysis

One of the fundamental ways to improve performance is performing *scheduling transformations*: changing the order of the execution of computations can improve locality (bringing memory locations accessed multiple times temporally closer together) or enhance parallelism (increasing the number of independent computations that can be executed simultaneously). The order in which computations execute is generally known as a *schedule* and not all schedules are legal. If

[1]With the exception of loop scaling, which is not a unimodular transformation, but also does not represent a true rescheduling of a computation.

```
1  node L = /* initialize data structure */;

3  void compute(int n, int i)
4    if (.. /*guard*/ ..)
5      n.data[i] = n.next.data[i−1] + 1

7  void loop(int i, node n)
8    if (i >= N) return;
9    traverse(i, n); //t1
10   loop(i+1, n); //r1

12 void traverse(int i, node n)
13   if (n == NULL) return;
14   compute(n, i); //s1
15   traverse(i, n.next); //r2

17 main()
18   loop(0, L);
```

(a) Repeatedly traversing a list.

```
1  void traverse(node n, int j, int i)
2    if (j>=len+N−1) return;
3    /*calls that can be executed in parallel*/
4    loop(n, j, max(0, j−len+1));
5    /* compute next node */
6    node nn = ~(n.next) ? n : n.next;
7    int jj = j+1;
8    traverse(nn, jj, i);

10 void loop(node n, int j, int i)
11   if (i > min(j, N−1)) return;
12   compute(n, i)
13   loop(n.prev, j, i+1);

15 main()
16   traverse(L, 0, 0);
```

(b) After transforming.

Fig. 1. Running example.

computations s_1 and s_2 access the same memory location and one of the accesses is a write, then there is a *dependence* between these computations. The set of legal schedules for a program is constrained by dependences, since the order of execution of dependent computations must be identical in all legal schedules to ensure the program computes the same result.

Consider the code in Figure 1a, which is a doubly-linked list traversal where each node has an array. The function loop represents the loop that repeatedly traverses a list N times. For uniform treatment of loops and recursion, it has been converted to tail recursion. The statement s_1 in line 14 gets executed for every combination of n—the list node on which computation is being performed—and i—the i^{th} traversal on the list. Each dynamic execution of s_1 is known as an *instance*. Classical instruction scheduling considers only the static instruction when building a schedule, but here each instance of s_1 represents a different dynamic invocation of the *same* static instruction. Thus, it is necessary for scheduling transformations for loops and recursion to reason about *instances*, not *instructions*.

Reasoning about dynamic instances rather than static instructions is called *instance-wise* analysis. An instance-wise analysis framework consists of three components: (i) a representation of an *iteration space* that uniquely names each dynamic instance places it in a schedule; (ii) a representation of transformations that maps one schedule of instances to another; and (iii) a representation of dependences that places constraints on what schedules are legal along with a decision procedure for determining whether a given transformation violates those dependences.

Instance-wise analysis is common for *regular programs* with nested loop structure that operate over dense arrays [Bondhugula et al. 2008; Feautrier 1992a,b]. But when it comes to *irregular programs*, with data structures such as lists and trees, and control structures such as recursive functions, there has been far less work. Amiranoff et al. [Amiranoff et al. 2006] has done a comprehensive study of instance-wise analysis for recursive programs. Their work introduces a context-free language representation for recursive programs that uniquely labels each dynamic instance for a statement using a trace string called a *control word*. Control words have been used to identify

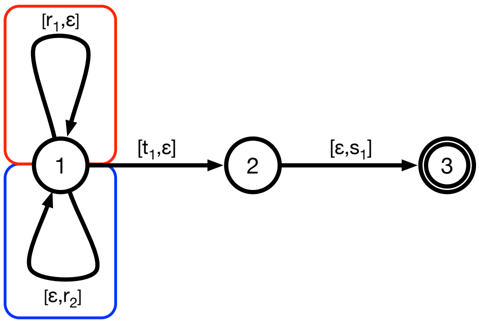


Fig. 2. PolyRec iteration space automaton

dependences among instances and parallelize them appropriately. But Amiranoff et al. only target *parallelizing* recursive programs given an initial schedule. They did not consider other scheduling transformations that could *change* the schedule of computation, and hence improve locality or expose additional parallelism. In other words, Amiranoff's work defines an iteration space, but does not provide the other two components of an instance-wise transformation framework.

2.2 PolyRec

In recent work, Sundararajah and Kulkarni proposed PolyRec, a framework for instance-wise reasoning about recursive programs that allows compilers to reason about classical loop transformations like interchange, reversal, and unrolling, as well as analogs that apply to recursive programs [Sundararajah and Kulkarni 2019]. To do so, PolyRec provides all three components of an instance-wise framework. We will use the code in Figure 1a to see how PolyRec builds these components.

Iteration space. The PolyRec framework represents the iteration space of a nested recursive code by naming each dynamic instance of the inner-most statements uniquely and capturing the ordering relation between those instances. PolyRec names each statement in a k-deep nest of recursion using 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. In our example, statement s_1 is in a two-deep nest of recursion and the instance would be named using a two-tuple of strings.

The very first instance is $[t_1, s_1]$: the statement that executes during the first invocation of loop and the first invocation of traverse. The next instance is $[t_1, r_2 s_1]$: the first invocation of loop, and the symbol r_2 representing one additional recursive invocation of traverse, and hence the second iteration of the inner loop. Analogously, the instance tuple naming the second iteration of the inner loop during the *second* iteration of the outer loop is $[r_1 t_1, r_2 s_1]$: one recursive invocation of loop followed by one recursive invocation of traverse. The strings for an instance of s_1 correspond to the labels of the functions (see the comments in Figure 1a) in the call stack when s_1 executes.

The iteration space is the collection of all instances. PolyRec uses *regular relations* to succinctly represent this (possibly statically unbounded) set of k-strings. This regular relation can be expressed as a *multitape automaton* [Rabin and Scott 1959] (since each instance is represented as a k-string). The set of all possible k-strings generated (accepted) by the automaton is the set of possible dynamic instances for the original program[2]. An important point is the k-string that represents a particular instance is *unique*. The automaton for a program is called its *iteration space automaton*. Figure 2 shows the automaton for the running example.

[2]More precisely, this set is a *superset* of the dynamic instances of the original program, since a terminating program will generate a finite set of instances, while most regular relations that PolyRec uses represent infinite sets of k-strings.

PolyRec must not only name the dynamic instances, but also capture the *schedule* of those instances—an ordering on the instances according to when they would dynamically execute in the program. PolyRec captures the order of dynamic instances by *flattening* the k-string—concatenating the individual strings in the tuple—and then sorting them by some lexicographic order. The order of the alphabet of the k-string matches the program order. For example, the recursive call r_1 in the function loop of Figure 1a appears after t_1, the call to traverse, so the symbol t_1 *precedes* the symbol r_1 in the alphabet. This ordering of labels guarantees that the lexicographic ordering of k-strings matches the execution schedule of their associated dynamic instances. Thus, the two main challenges of representing an iteration space—uniquely naming instances and expressing the order of execution of the instances (i.e. schedule)—are successfully overcome by the multitape automata representation with the order on its alphabet.

Transformations. A scheduling transformation is a one-to-one mapping between the instances of an original iteration space and the transformed one. Since instances are uniquely named, transformations are bijective maps (essentially, rewrites) from one set of k-strings to another.

PolyRec represents transformations as *multitape transducers* that translate the k-strings of an input regular relation to an output regular relation. For instance, the transitions $[r_1, \epsilon] \longrightarrow [\epsilon, r_2]$ and $[\epsilon, r_2] \longrightarrow [r_1, \epsilon]$ takes symbols from the first dimension of an input k-string and moves them to the second dimension and vice versa. These transitions thus implement *interchange*, swapping the inner and outer dimensions of a nested loop or recursion. The order of the output alphabet determines the lexicographic ordering of the k-strings in the transformed iteration space (i.e. transformed schedule).

Because transducers are composable [Rabin and Scott 1959], PolyRec gains two advantages from this representation: (i) composing an input iteration space automaton with a transformation transducer yields exactly the output iteration space automaton; (ii) *multiple* transformations can be composed into a single transducer that captures the sequence of transformations. This second point is crucial: some transformations yield schedules that are unsound, but a subsequent transformation can restore a valid schedule; without the ability to reason about sequences of transformations, PolyRec would be limited in the types of scheduling transformations it could validate.

Dependences and Soundness Checks. When transformations are applied to an iteration space, its schedule changes and not all schedules are valid: if two dependent instances change their order, the new schedule will produce incorrect results. PolyRec represents sets of dependences as a *witness tuple*. This is a 3-tuple of regular relations that captures the *set of pairs of dependent instances*. A witness tuple is written as $< R_\alpha, < R_\beta, R_\gamma >>$. R_α is the *common prefix* of the pair of instances and this can be treated as the prefix of the k-strings of dependent pairs. R_β is the *source suffix* and R_γ is the *sink suffix*. Each pair of k-strings representing a dependent pair of instances can be constructed by choosing a k-string from common prefix, then appending an element from source suffix to form the first k-string and an element from the sink suffix to form the second k-string.

In certain aspects, this representation is similar to the *distance vectors* used in affine iteration spaces to capture dependences [Kennedy and Allen 2001], but witness tuples can capture dependences of non-affine iteration spaces such as the ones considered in PolyRec and UniRec. In Figure 1b, $[t_1, r_2 s_1]$ and $[r_1 t_1, s_1]$ are a dependent pair of instances. Similarly, $[r_1 t_1, r_2 s_1]$ and $[r_1 r_1 t_1, s_1]$ are a dependent pair of instances. All dependent pairs of instances can be summarized as

$$\langle [r_1^*, r_2^*],\ ([t_1, r_2 s_1],\ [r_1 t_1, s_1]) \rangle$$

The affine equivalent of this dependence would be the distance vector $[1, -1]$. We can think of the source and sink suffixes as defining the start and end of the distance vector and the common prefix as the points where the distance vector can be placed.

The key to PolyRec's dependence representation is it allows the construction of a *decidable soundess check* that can determine whether a given (composed) transformation transducer violates any of the dependences. PolyRec's decision procedure checks all of the dependences captured by a witness tuple $\langle R_\alpha,\ (R_\beta,\ R_\gamma)\rangle$ as follows. First, it picks a single prefix from R_α, a. Note that *any* k-string that results from concatenating a with a suffix from R_β must lexicographically precede *any* k-string that results from concatenating a with a suffix from R_γ. PolyRec makes sure that after transformation this fact is still true (and hence the relative ordering of dependent instances is preserved) as follows:

- Run a through the compound transformation transducer to find all possible states it may end up in. Mark those states as the initial states of a new *prefix transducer*. This prefix transducer captures a partially-executed rewrite of a control word.
- Take the source suffix relation, R_β, and compose it with the prefix transducer. Project out the output tapes to find the rewrite of the source instances in the transformed iteration space (R'_β). Perform the same actions to the sink suffix R_γ to get the sink instances in the transformed iteration space (R'_γ).
- Find the *latest string* generated by R'_β and the *earliest string* generated by R'_γ.
- Checking whether the *latest string* lexicographically precedes *earliest string*. If it does, it means that *for this prefix*, all source instances continue to precede all sink instances.

PolyRec repeats this process for every possible prefix transducer (since the automata are finite state, there are only a finite number of prefix transducers) to verify soundness. Note that this decision procedure is only valid for *order-free* transducers: those where any order of consuming symbols from the input tape is valid, and will not get stuck [Sundararajah et al. 2017].

The key operation in this decision procedure is constructing the lexicographically earliest (latest) string that an automaton can produce. PolyRec does this by *reducing* the automata for the transformed iteration spaces: at each state in the automaton, take the earliest (latest) possible labeled transition on the first tape and remove the other options. The resulting automaton either has a single path, or multiple paths that differ only on later tapes. Repeating this process for all tapes generates the earliest (latest) string. (There are some subtleties regarding final states; details are in Sundararajah et al. [2017]).

2.3 Skewing

Loop skewing is a well-studied transformation that can restructure a loop nest to expose parallelism when performed in conjunction with other scheduling transformation such as loop interchange [Wolfe 1986]. Consider the code in Figure 3a. Figure 3e shows the iteration space of the code on the left and the same iteration space when skewed. In Figure 3e, circles represent different instances of the computations s_1 in Figure 3a. In skewing, every row of circles in the original iteration space is mapped to the corresponding row in the skewed iteration space but each instance is shifted to the right (positive) side. Note that the shift of a row of circles depends on the row: the 0^{th} row is not shifted, the 1^{st} row is shifted by 1, etc. Because the skew depends on the row, in the transformed iteration space, each instance column index depends on its row. This transformation can be succinctly expressed as $[i, j] \rightarrow [i, i + j]$, where $[i, j]$ is the unique cartesian naming for each instance in the original iteration space based on the loop indices. The red arrows in Figure 3e shows the dependences between instances. The purpose of performing such a transformation is to enable one dimension of the loop nest to be executed in parallel. Figure 3c shows the code after the transformation of loop skewing and interchange. Thus, in the transformed iteration space the dependences are only along the columns, hence the skewed code can be parallelized by column (or by row, after interchange).

```
for (int i = 0 .. N−1)
    for (int j = 0 .. N−1)
        a[i][j] =
          a[i−1][j+1] + 1; //s1
```

(a) Before positive skew.

```
for (int i = 0 .. N−1)
 for (int j = 0 .. N−1)
     a[i][j] =
       a[i−1][j−1] + 1; //s2
```

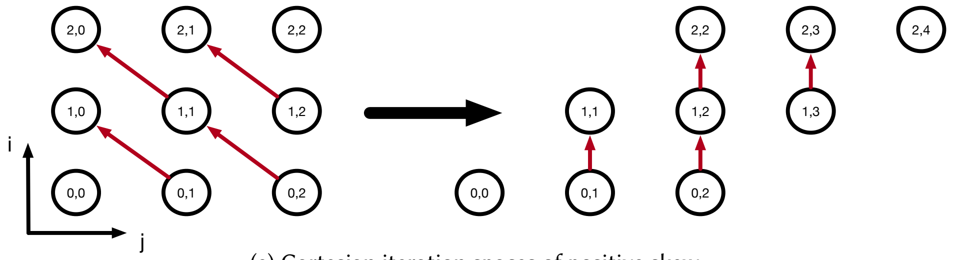
(b) Before negative skew.

```
int lb1 = 0
int ub1 = 2N−1
for (int j = lb1 .. ub1)
    int lb2 = max(0, j−N+1)
    int ub2 = min(j, N−1)
    for (int i = lb2 .. ub2)
        a[i][j−i] =
          a[i−1][j−i+1] + 1;
```

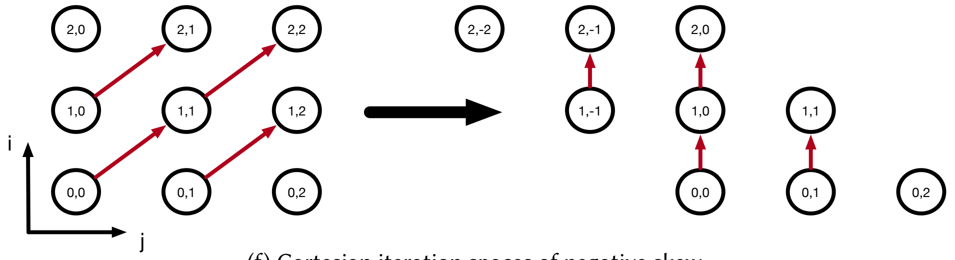
(c) After positive skew.

```
int lb1 = −N+1
int ub1 = N−1
for (int j = lb1 .. ub1)
    int lb2 = max(0, −j)
    int ub2 = min(−j+N−1, N−1)
    for (int i = lb2 .. ub2)
        a[i][j+i] =
          a[i−1][j+i−1] + 1;
```

(d) After negative skew.



(e) Cartesian iteration spaces of positive skew.



(f) Cartesian iteration spaces of negative skew.

Fig. 3. Loop skewing

Loop skewing is a key transformation in the *unimodular framework* [Banerjee 1991], which represents transformations of loop nests with unimodular transformation matrices that capture how instances are renamed from one iteration space to another. Skewing, along with interchange and reversal, form a complete basis for unimodular transformations: any unimodular transformation can be decomposed into a sequence of skewing, interchange, and reversal transformations. Hence, skewing is a crucial component of sophisticated loop transformations, including complex tiling transformations such as diamond tiling [Bandishti et al. 2012; Bertolacci et al. 2015].

So what does skewing loop look like when loops are represented as recursion, as in PolyRec?

Skewing in recursive programs. In the code in Figure 1a, instance-wise analysis reveals the dependence among the instances $(i, n.\text{next})$ and $(i + 1, n)$, since the former writes to n.next.data[i] and the later reads from the same location. This dependence structure is similar to the nest of regular loops described in Section 2.3. A scheduling transformation similar to loop skewing can expose parallelism in the code in Figure 1a. Figure 1b shows such a scheduling transformation where all the invocations of the call on line 4 can be executed in parallel. The key here is that each recursive function carries an extra induction variable that tracks how much the inner "loop" needs to be delayed by in order to implement the skew. The use of min and max in this code is analogous to the use of min and max to implement loop bounds when applying skewing to loop codes.

Unfortunately, PolyRec does not support skewing as a transformation, and hence cannot completely cover the space of transformations of perfectly-nested programs. Interestingly, PolyRec's inability to capture skewing is not merely a matter of not providing an appropriate transformation transducer that describes skewing. Instead, it is a fundamental limitation of PolyRec's representation, as we explain next.

PolyRec's insufficiency for skewing. One key aspect of PolyRec's transducer-based representation of transformations is that it is *monotonic*[3]: while an output k-string can contain fewer symbols than an input k-string (the transducer might consume multiple input symbols and produce only one output symbol), if an input k-string p_1 is a *prefix* of a longer k-string p_2 (meaning p_2 has additional symbols in at least one dimension), and the transformer rewrites p_1 and p_2 into q_1 and q_2, respectively, then q_1 *must* be a prefix of q_2: adding symbols to an input k-string can only *add* symbols to the output k-string.

Nevertheless, even with this restriction, PolyRec's representation can capture a positive skew: for example, consuming a recursive symbol from dimension i and producing output symbols on *both* dimensions i and j can produce the $[i, j] \rightarrow [i, i + j]$ mapping. For example, $[r_1 t_1, r_2 s_1]$ would be transformed into $[r_1 t_1, r_2 r_2 s_1]$. The monotonicity of transducers is a good fit for the additive nature of positive skewing transformations.

Unfortunately, skewing transformations are not always additive. Skewing can also work in the *negative* direction. Figures 3b, 3f, and 3d show a piece of code, the effects of negative skewing on its iteration space, and the transformed code after negative skewing, respectively. This transformation can be succinctly expressed as $[i, j] \rightarrow [i, j - i]$ This transformation is not expressible using PolyRec's representation as it is *non-additive*: for each symbol on the i dimension, a symbol needs to be *removed* from the j dimension. Note in particular that there is no obvious way to associate the j dimension of an instance with a *negative* position in the iteration space.

The key challenge addressed in this paper is how to extend PolyRec to handle skewing, and hence how to extend PolyRec to have some notion of *negative symbols* that enable the expression of non-additive transformations. The next section discusses how we introduce negative symbols to PolyRec, leading to a generalized framework UniRec. The remainder of the paper deals with the implications of this generalization.

3 KEY IDEAS OF UNIREC

This section provides key ideas of UniRec's representations. In this section, first, we will introduce the new representation of iteration spaces, which include *negative tapes and symbols*, and justify the need for this addition. Then we elaborate on the two main challenges that arise due to the addition of negative tapes and symbols. Finally we discuss the key ideas to solve the challenges. In subsequent sections, we discuss these solutions in detail.

Running Example. Figure 1a shows the code of a program that traverses a doubly linked list where every node has an array and this is our running example. In this program, The list is traversed N times and i^{th} traversal access the i^{th} element of the array in the present node and $(i-1)^{th}$ element of the array in the next node. Figure 1b shows after performing skewing and interchange

[3]This is not a property of non-deterministic finite transducers in general, but *is* a property of PolyRec's transducers.

to the running example. The variable len is the length of the list. While the transformation running example can be expressed without len (i.e., the length computation can be incorporated to the transformation), those details are not shown for brevity.

3.1 Representing Iteration Space

As we recall from Section 2.3, PolyRec cannot easily represent skewing transformations because it cannot move an iteration "backwards" in a schedule. In other words, if we take the k-string of a particular instance, the representation of PolyRec is not relaxed enough to map the instance relatively backward in the total order of the instances (i.e. schedule)—adding symbols to a tape can only move an instance later in the schedule. The key insight of UniRec's iteration space representation is that for a dimension performing a *linear recursion*—recursive functions that are isomorphic to loops—we can introduce a new tape, called a *negative tape* where adding symbols (called *negative symbols*) to that tape has the effect of pushing a k-string *earlier in the schedule.* These negative tapes and symbols provide enough power to capture skewing-like transformations for recursive programs.

Introducing negative tapes and symbols. Let us consider the running example in Figure 1a. Recall that Figure 2 is the automaton that PolyRec uses to capture the iteration space of this program: there is one tape for each dimension of the program.

In UniRec's representation each dimension is represented by *two* tapes, a *positive* tape and a *negative tape*[4]. The alphabet of the negative tape is t_1^-, r_1^- and it has the ordering of $r_1^- \prec t_1^-$, while the alphabet of the positive tape is t_1^+, r_1^+ and it has the ordering of $t_1^+ \prec r_1^+$. Note that the positive dimension is similar to PolyRec: the word $r_1^+t_1^+$ is lexicographically earlier than the word $r_1^+r_1^+t_1^+$—adding a r_1^+ pushes the second word later in the order. But the negative dimension uses a different ordering on its symbols: the word $r_1^-t_1^-$ is *later* than the word $r_1^-r_1^-t_1^-$. Adding a recursive symbol *moves the word earlier in the total order.* We can similarly add a negative tape for the second dimension.

More generally, for any linear recursion, the *positive* tape orders the symbols in program order, while the *negative* tape orders the symbols in reverse program order. The two main points here are (i) the positive tape is analogous to PolyRec's representation, and (ii) there is an implicit ordering between the negative and positive tapes of the same dimension (negative before positive). Since every (linear) dimension has two tapes, an instance is represented with $2k$-string and the multitape automaton representing the k-dimension iteration space has $2k$ tapes.

Figure 4 shows the iteration space of our running example in UniRec's representation. Note how compared with PolyRec's representation (Figure 2), there are twice as many tapes, and four "paths" to the final state instead of just two. Further details about UniRec iteration space representation is discussed in Section 4.1.

Now UniRec's representation allows us to move an instance both forward and backward in a particular dimension. For example, consider the instance i with control word $[t_1, s_1]$ in PolyRec's representation. In UniRec's representation, it is $[t_1^-, t_1^+, s_1^-, s_1^+]$. If we consider a forward shift to i in first dimension, we can simply add r_1^+ symbol to the positive tape of first dimension (e.g. $[t_1^-, r_1^+t_1^+, s_1^-, s_1^+]$). To shift i *backwards* in time, we instead add r_1^- to the negative tape of the first dimension, yielding $[r_1^-t_1^-, t_1^+, s_1^-, s_1^+]$. Note that this new control word, when flattened, is ordered *before* i's control word: *adding* a symbol to the negative tape moved the instance *earlier.*

[4]Our use of two tapes to define an equivalence class of pairs to represent positive or negative integers is similar to *Brahmagupta*'s formulation of numbers in terms of debts (negative increments) and fortunes (positive increments), whose sum represented the total value of a number.

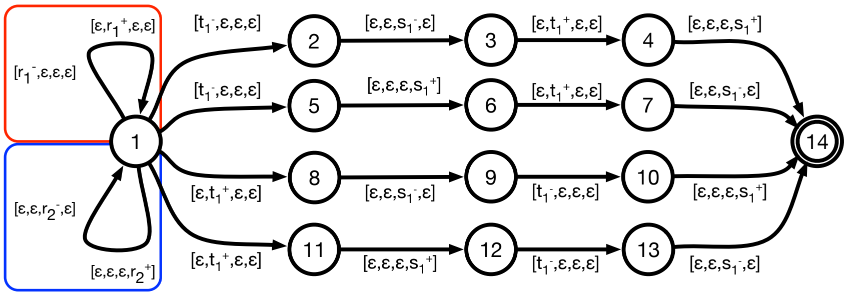


Fig. 4. UniRec iteration space automaton

3.2 Challenges of UniRec's Representation

Now that we have a clear picture of UniRec's instance naming and iteration space representation, we note two key challenges that arise due to this representation.

Breaking uniqueness of naming instances. Let us consider an example of instance $i = [t_1^-, t_1^+, s_1^-, s_1^+]$, first shifted forward by one and then backward by one in the first dimension. From the previous examples of shift, this would yield $[r_1^- t_1^-, r_1^+ t_1^+, s_1^-, s_1^+]$, a different (longer) control word. But note that shifting an instance forward by one and backward by one returns the instance to the same location in the schedule. This control word represents the same instance as the original control word, but the 4-strings are different. This means in UniRec's representation, an instance can be represented by many different $2k$-strings. This fact breaks the fundamental assumption that each instance is uniquely named; there are many different ways to represent an iteration space using UniRec's representation. It may be unclear why non-unique naming for instances is a flaw. The problem arises when we consider the schedule of these instances.

Breaking lexicographical ordering. Consider the string $i = [s_1^-, s_1^+]$ from a one-dimensional recursion. If we shift i forward by one step and backward by one step, we get $i_1 = [r_1^- s_1^-, r_1^+ s_1^+]$, which should represent the same location in the order. If we instead shift i forward by three steps and backwards by two, we get $i_2 = [r_1^- r_1^- s_1^-, r_1^+ r_1^+ r_1^+ s_1^+]$, which should be later in the schedule. However, if we lexicographically compare i, i_1, and i_2, we see that $i_1 \prec i$, since i_1 has more negative symbols in the very first tape. Hence, the lexicographical ordering alone is not sufficient to compare strings representing the same instance. Worse, we also have $i_2 \prec i_1$, since there are more negative symbols in the first tape of i_2 than that of i_1. This contradicts the desired goal that $i \prec i_2$. Hence, introducing negative tapes makes the plain lexicographical ordering to represent the schedule invalid.

3.3 Solution to UniRec's Representation Issue

So how do we solve the dual problems of negative tapes? We introduce *canonicalization*: a process by which all control words that represent the same dynamic instance can be mapped to a single, *canonical control word.*

Canonicalization. The different control words representing the same instance belong to an *equivalence class.* By assigning each equivalence class a canonical member of that equivalence class, we can map any k-string to its canonical, representative control word, and hence solve the uniqueness problem.

Note that for a pair of control to be in an equivalence class, one can be obtained from the other by shifting one or more dimensions forward and backward by the same amount. In other words, to obtain one control word from another, an equal number of positive and negative symbols must be added to one or more dimensions (remember that a dimension is represented by a positive tape and a negative tape). It should be clear that what identifies a control word's equivalence class is the *difference* in the number of positive and negative symbols in each dimension.

This observation yields a simple definition of a canonical control word for an equivalence class: the shortest control word (the one with the fewest recursive symbols). By definition, a canonical control word is one where, for each dimension, *at most one tape* for that dimension contains recursive symbols. This definition also leads to a straightforward canonicalization procedure: "cancel out" matched recursive symbols from the negative and positive tapes until at most one tape still has recursive symbols. For example consider the $2k$-string $[r_1^- r_1^- t_1^-, r_1^+ r_1^+ r_1^+ t_1^+, s_1^-, s_1^+]$. The first dimension has recursive symbols on both positive and negative tapes, so remove matching r_1^- and r_1^+. We end up with the canonical control word $[t_1^-, r_1^+ t_1^+, s_1^-, s_1^+]$.

The only question, then, is whether these canonical control words capture a program's schedule through their lexicographic ordering. If we can verify that the lexicographic ordering works correctly on canonical words, then we can design every other part of UniRec to account for canonicalization.

In a canonicalized string, for a dimension either positive or negative tape has recursive symbols, but not both. Suppose we are comparing canonical control words a and b in a particular dimension, then there are the following cases (note that negative tape comes before the positive tape for every dimension).

- In both a and b, the same tape has recursive symbols (or neither tape does). This is the simple case: we compare the tapes and if they have the same string, move to the next dimension, or the straightforward comparison of strings as discussed in Section 3.1 suffices to order the control words.
- In a, only the positive tape has recursive symbols and in b, only the negative tape does. Note that this means that b precedes a in the schedule. The negative tape precedes the positive tape, and b's string on the negative tape (which has recursive symbols) precedes a's (which does not), so b also lexicographically precedes a.
- A similar argument applies when a has recursive symbols on the negative tape, and b has them on the positive tape.

This shows that for canonical control words, lexicographic ordering captures schedule ordering. Crucially, any comparison must be done on canonicalized strings. In other words, using lexicographical order to express schedule must come *after* canonicalization.

We have established that the key idea to support unimodular loop transformations is to extend the PolyRec apparatus with negative tapes and symbols, and then perform canonicalization. Section 4.3 explains the UniRec machinery to make transformations and dependences integrate canonicalization such that the soundness checking algorithm of PolyRec works for UniRec. Section 5 provides a proof sketch for the correctness of UniRec's soundness checking algorithm.

4 UNIREC ITERATION SPACES AND SCHEDULING TRANSFORMATIONS

4.1 Iteration Spaces

Since there are positive and negative tapes, there are multiple possible paths from initial state to the final state in UniRec's representation as shown in Figure 4. We have only shown 4 paths here but there are 24 valid paths. For example, $t_1^- \rightarrow t_1^+ \rightarrow s_1^- \rightarrow s_1^+$ is a valid path that is not shown in the figure. If there are k dimensions then there are $(2k)!$ valid paths. Note that these alternative paths generate the canonical control words, so do not make the iteration space *bigger.*

4.2 Scheduling Transformations

As we recall scheduling transformations are one-to-one mapping between original iteration space and the transformed one. We provide a way to realize that mapping in this section. First, we formally define *Multitape Transducers* and then we provide a formal composition algorithm for these multitape transducers. After that we illustrate the three main UniRec transformation transducers: skew, interchange and reversal for nested recursive iteration space. Finally, we move onto the discussion about realizing the canonicalization strategy (Section 3.3) using transducers. We provide the construction of a *Canonicalization Transducer* and illustrate the composition operation of a composed UniRec transformation transducer and a canonicalization transducer.

4.2.1 UniRec Multitape Transducers. First, we formally describe multitape transducers, as used in UniRec (we do not distinguish positive and negative tapes). The main difference between PolyRec transducer and UniRec transducer is that, in UniRec, a transition can write to multiple tapes at a time with multiple symbols. The UniRec composition algorithm thus must handle multiple tape and multiple symbols rewrites.

Definition 4.1. $\mathcal{T}$ is *non-deterministic, m-to-n finite state transducer* defined as a 8-tuple,

$$\langle m, n, \Sigma_{in}, \Sigma_{out}, Q, Q_0, F, E\rangle, \text{ where:}$$

- m is the number of input tapes.
- n is the number of output tapes.
- Σ_{in} is the finite input alphabet.
- Σ_{out} is the finite output alphabet.
- Q is a finite set of states.
- $Q_0 \in Q$ is the set of *start* states.
- $F \subseteq Q$ is a set of *accept* final states.
- $E \subseteq Q \times (\Sigma_{in} \cup \varepsilon)^m \times (\Sigma_{out} \cup \varepsilon)^n \times Q$ is a finite set of labeled transitions (each labeled with and m-tuple of input symbols and and n-tuple of output symbols).

4.2.2 Composition of UniRec *Transformation Transducers.* Each transducer in UniRec represents a single scheduling transformation. Since a particular program transformation might involve multiple individual transformations (say, skewing followed by interchange), UniRec*composes* transducers to produce a single combined transducer that represents the overall schedule transformation. (This is analogous to composing multiple transformation matrices to produce a single compound transformation matrices in the unimodular framework.)

Definition 4.2. Let $\mathcal{T}' = \langle m', n', \Sigma'_{in}, \Sigma'_{out}, Q', Q'_0, F', E'\rangle$, $\mathcal{T}'' = \langle m'', n'', \Sigma''_{in}, \Sigma''_{out}, Q'', Q''_0, F'', E''\rangle$ and the composed transducer $\mathcal{T} = \mathcal{T}' \circ \mathcal{T}'' = \langle m, n, \Sigma_{in}, \Sigma_{out}, Q, Q_0, F, E\rangle$.

For the composition to be valid, it is required that input of $\mathcal{T}''$ and output of $\mathcal{T}'$ must be the same. The result of the composition $\mathcal{T}$, has the input of $\mathcal{T}'$ and the output of $\mathcal{T}''$.

In a nutshell, the composition of two transducers is a cross product of transitions over the operation of a composable pair of transitions. Algorithm 1 formally describes the composition of the UniRec transducers $\mathcal{T}'$ and $\mathcal{T}''$. Essentially, Algorithm 1 is describing this cross product while breaking it down to three different cases due to the nature of transitions in UniRec transducers. Following are the cases,

Case 1 Transition of $\mathcal{T}'$ has no nonempty symbols on output tapes (i.e. all output tapes got ϵ). This transition can be composed with every transition of $\mathcal{T}''$.

Case 2 Transition of $\mathcal{T}''$ has exactly one output tape with only one nonempty symbol. (e.g. $[r_1, \epsilon, \epsilon, \epsilon]$). This transition can be composed with transition of $\mathcal{T}''$ with no nonempty symbols

Algorithm 1: Composition of Transitions

```
$E \leftarrow \emptyset$
forall $\delta' \in E'$ do
    $(src', i', o', dst') \leftarrow \delta'$
    if ZeroNoneEmpty($o'$) then
        forall $\delta'' \in E''$ do
            $(src'', i'', o'', dst'') \leftarrow \delta''$
            $\delta \leftarrow ((src, src''), i', o'', (dst', dst''))$
            $E \leftarrow E \cup \{\delta\}$
    else if OneNoneEmpty($o'$) then
        forall $\delta'' \in E''$ do
            $(src'', i'', o'', dst'') \leftarrow \delta''$
            if ZeroNoneEmpty($i''$) or ExactMatch($o'$, $i''$) then
                $\delta \leftarrow ((src, src''), i', o'', (dst', dst''))$
                $E \leftarrow E \cup \{\delta\}$
    else
        forall $q'' \in Q''$ do
            $[\delta''_1, \delta''_2 \dots \delta''_l] \leftarrow$ GetMultiTransitions($o', E'', q''$)
            if IsValid($[\delta''_1, \delta''_2 \dots \delta''_l]$) then
                $i \leftarrow i'$
                $o \leftarrow o''_1 \circ o''_2 \cdots \circ o''_l$
                $\delta \leftarrow ((src, src''_l), i, o, (dst', dst''_l))$
                $E \leftarrow E \cup \{\delta\}$
```

on input tapes or the one with input tapes that exactly matches the output tapes of the transition of $\mathcal{T}'$

Case 3 Transition of $\mathcal{T}'$ with multiple output tapes with nonempty symbols. Those tapes may even have multiple symbols in one tape. This case requires special handling since any UniRec transducer does not have transitions with input tapes where multiple tapes have nonempty symbols except the *canonicalization transducer*. GetMultiTransitions function returns an ordered set of transitions starting from a state q''. The output tapes of the transition of $\mathcal{T}'$ can be constructed by combining the input tapes of this set of transitions given by GetMultiTransitions. In addition to that if every transition's ending state in this set is the next one's starting state then it is considered a valid set of transitions of $\mathcal{T}''$. If that's a valid set then it is composed as shown.

4.2.3 UniRec *Transformation Transducers.* When it comes to UniRec transformation transducers, transitions of non-recursive symbols are basically identity transitions—transitions that directly copies the input tapes to output tapes—-and they have the same structure as in iteration spaces. For the UniRec transformation transducers, we provide all the paths from start state to end state with identical transitions of non-recursive symbols. Hence, if there are k dimensions then, there are $(2k)!$ paths from the initial state to the final state. All of these paths are different combination of identity transitions of non-recursive symbol designated to each tape. This large number of paths does not expand the iteration space; it is needed to preserve PolyRec's order-free property: to

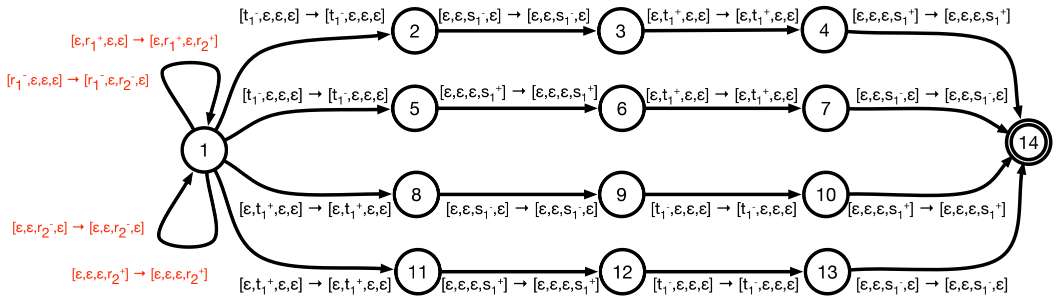

$[\varepsilon,r_1^+,\varepsilon,\varepsilon] \rightarrow [\varepsilon,r_1^+,\varepsilon,r_2^+]$
$[r_1^-,\varepsilon,\varepsilon,\varepsilon] \rightarrow [r_1^-,\varepsilon,r_2^-,\varepsilon]$
$[\varepsilon,\varepsilon,r_2^-,\varepsilon] \rightarrow [\varepsilon,\varepsilon,r_2^-,\varepsilon]$
$[\varepsilon,\varepsilon,\varepsilon,r_2^+] \rightarrow [\varepsilon,\varepsilon,\varepsilon,r_2^+]$
1
2
3
4
5
6
7
8
9
10
11
12
13
14
$[t_1^-,\varepsilon,\varepsilon,\varepsilon] \rightarrow [t_1^-,\varepsilon,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,s_1^-,\varepsilon] \rightarrow [\varepsilon,\varepsilon,s_1^-,\varepsilon]$
$[\varepsilon,t_1^+,\varepsilon,\varepsilon] \rightarrow [\varepsilon,t_1^+,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,\varepsilon,s_1^+] \rightarrow [\varepsilon,\varepsilon,\varepsilon,s_1^+]$
$[t_1^-,\varepsilon,\varepsilon,\varepsilon] \rightarrow [t_1^-,\varepsilon,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,\varepsilon,s_1^+] \rightarrow [\varepsilon,\varepsilon,\varepsilon,s_1^+]$
$[\varepsilon,t_1^+,\varepsilon,\varepsilon] \rightarrow [\varepsilon,t_1^+,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,s_1^-,\varepsilon] \rightarrow [\varepsilon,\varepsilon,s_1^-,\varepsilon]$
$[\varepsilon,t_1^+,\varepsilon,\varepsilon] \rightarrow [\varepsilon,t_1^+,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,s_1^-,\varepsilon] \rightarrow [\varepsilon,\varepsilon,s_1^-,\varepsilon]$
$[t_1^-,\varepsilon,\varepsilon,\varepsilon] \rightarrow [t_1^-,\varepsilon,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,\varepsilon,s_1^+] \rightarrow [\varepsilon,\varepsilon,\varepsilon,s_1^+]$
$[\varepsilon,t_1^+,\varepsilon,\varepsilon] \rightarrow [\varepsilon,t_1^+,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,\varepsilon,s_1^+] \rightarrow [\varepsilon,\varepsilon,\varepsilon,s_1^+]$
$[t_1^-,\varepsilon,\varepsilon,\varepsilon] \rightarrow [t_1^-,\varepsilon,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,s_1^-,\varepsilon] \rightarrow [\varepsilon,\varepsilon,s_1^-,\varepsilon]$

Fig. 5. Two dimensional positive skew transducer

ensure that transducers can rewrite symbols from different tapes in any order, all possible rewrite orders must preserve paths to accept states.[5]

We discuss the construction of skew, interchange, and reversal transducers. Since the part that involves non-recursive symbols of all UniRec transducers is the same, we discuss the handling of recursive symbols in the construction.

Recursion Skewing and General Linear Transducers. Figure 5 shows a two-dimensional recursion skewing transducer. This transducer represents a positive skew for the nested recursive iteration spaces that is similar to the one described for loop nests in Section 2.3 (i.e. $[i, j] \longrightarrow [i, i + j]$). The important transitions to notice here are the ones looping at the initial state that fills more than one output tapes ($[\epsilon, r_1^+, \epsilon, \epsilon] \longrightarrow [\epsilon, r_1^+, \epsilon, r_2^+]$ and $[r_1^-, \epsilon, \epsilon, \epsilon] \longrightarrow [r_1^-, \epsilon, r_2^-, \epsilon]$). These transitions consume recursive symbols from the first dimension and add symbols to both the first and second dimension. The former handles the positive tape, and the latter handles the negative tape of first dimension. This shifts the instances in the iteration space along the second dimension and the size of shift is determined by the first dimension. The second dimension's recursive symbols are mapped identically (i.e. $[\epsilon, \epsilon, r_2^-, \epsilon] \longrightarrow [\epsilon, \epsilon, r_2^-, \epsilon]$ and $[\epsilon, \epsilon, \epsilon, r_2^+] \longrightarrow [\epsilon, \epsilon, \epsilon, r_2^+]$).

Skewing can be parameterized by a constant. For example, $[i, j] \longrightarrow [i, 2 * i + j]$ is another skew for doubly nested loops where the constant is two. Such a parameterized skew can be expressed the same way before with looping transitions with more symbols added to a tape. In this particular case, the transition that handles the positive tape of the first dimension would be $[\epsilon, r_1^+, \epsilon, \epsilon] \longrightarrow [\epsilon, r_1^+, \epsilon, r_2^+ r_2^+]$. Notice that the positive output tape of the first dimension has two r_2^+ symbols. The transition that handles the negative tape is similar where there are two r_1^- symbols.

We can design the transducer to also handle negative skew of nested recursive iteration spaces similar to $[i, j] \longrightarrow [i, j - i]$ for loop nests. In this case the transitions handling tapes for first dimension would be $[\epsilon, r_1^+, \epsilon, \epsilon] \longrightarrow [\epsilon, r_1^+, r_2^-, \epsilon]$ and $[r_1^-, \epsilon, \epsilon, \epsilon] \longrightarrow [r_1^-, \epsilon, \epsilon, r_2^+]$. Notice that the former transition that handles the positive input tape of first dimension, fills the *negative* tape of second output dimension in the output with r_2^-: essentially, the larger the input i is, the more negative the output second dimension becomes.

It should be apparent that constructing skewing transducers in this way can essentially handle *any* linear, integral scheduling transformation of the input dimensions ($[i, j] \longrightarrow [a*i+b*j, c*i+d*j]$ for integers a, b, c and d), by carefully choosing which output tapes to emit symbols to. An immediate corollary is that any unimodular transformation has a corresponding generic skewing transducer. In particular, UniRec can handle classical loop transformations:

[5]Interestingly, PolyRec only defined order-freeness by placing a structural restriction on transducers that UniRec's transducers do not respect. But they do obey the general property of order-freeness described here.

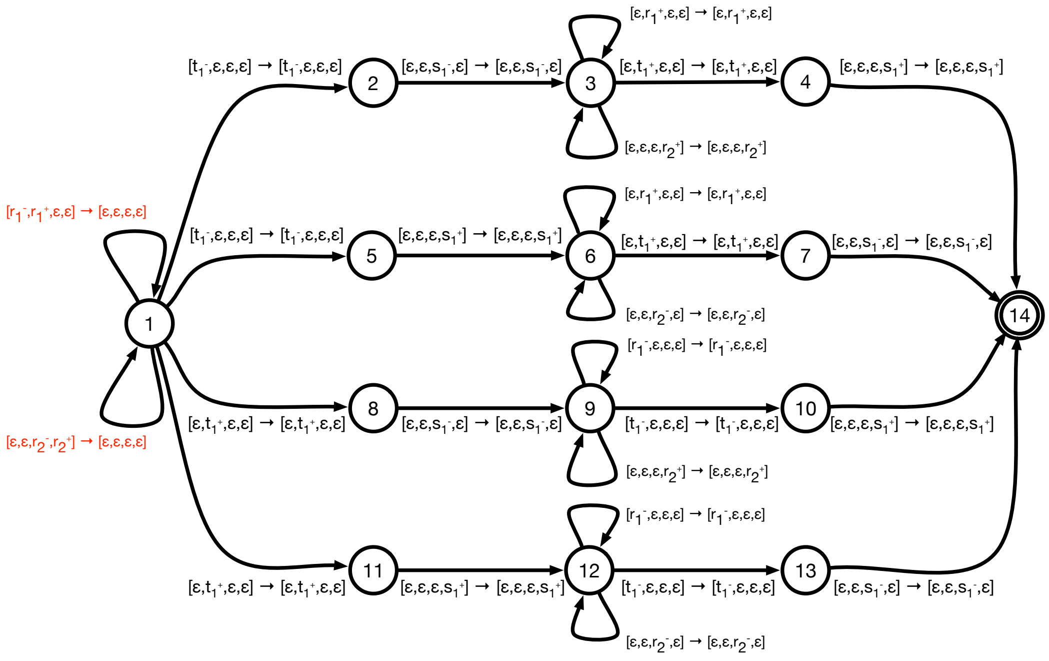

1
2
3
4
5
6
7
8
9
10
11
12
13
14
$[r_1^-,r_1^+,\varepsilon,\varepsilon] \rightarrow [\varepsilon,\varepsilon,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,r_2^-,r_2^+] \rightarrow [\varepsilon,\varepsilon,\varepsilon,\varepsilon]$
$[t_1^-,\varepsilon,\varepsilon,\varepsilon] \rightarrow [t_1^-,\varepsilon,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,s_1^-,\varepsilon] \rightarrow [\varepsilon,\varepsilon,s_1^-,\varepsilon]$
$[\varepsilon,r_1^+,\varepsilon,\varepsilon] \rightarrow [\varepsilon,r_1^+,\varepsilon,\varepsilon]$
$[\varepsilon,t_1^+,\varepsilon,\varepsilon] \rightarrow [\varepsilon,t_1^+,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,\varepsilon,r_2^+] \rightarrow [\varepsilon,\varepsilon,\varepsilon,r_2^+]$
$[\varepsilon,\varepsilon,\varepsilon,s_1^+] \rightarrow [\varepsilon,\varepsilon,\varepsilon,s_1^+]$
$[t_1^-,\varepsilon,\varepsilon,\varepsilon] \rightarrow [t_1^-,\varepsilon,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,\varepsilon,s_1^+] \rightarrow [\varepsilon,\varepsilon,\varepsilon,s_1^+]$
$[\varepsilon,r_1^+,\varepsilon,\varepsilon] \rightarrow [\varepsilon,r_1^+,\varepsilon,\varepsilon]$
$[\varepsilon,t_1^+,\varepsilon,\varepsilon] \rightarrow [\varepsilon,t_1^+,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,r_2^-,\varepsilon] \rightarrow [\varepsilon,\varepsilon,r_2^-,\varepsilon]$
$[\varepsilon,\varepsilon,s_1^-,\varepsilon] \rightarrow [\varepsilon,\varepsilon,s_1^-,\varepsilon]$
$[\varepsilon,t_1^+,\varepsilon,\varepsilon] \rightarrow [\varepsilon,t_1^+,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,s_1^-,\varepsilon] \rightarrow [\varepsilon,\varepsilon,s_1^-,\varepsilon]$
$[r_1^-,\varepsilon,\varepsilon,\varepsilon] \rightarrow [r_1^-,\varepsilon,\varepsilon,\varepsilon]$
$[t_1^-,\varepsilon,\varepsilon,\varepsilon] \rightarrow [t_1^-,\varepsilon,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,\varepsilon,r_2^+] \rightarrow [\varepsilon,\varepsilon,\varepsilon,r_2^+]$
$[\varepsilon,\varepsilon,\varepsilon,s_1^+] \rightarrow [\varepsilon,\varepsilon,\varepsilon,s_1^+]$
$[\varepsilon,t_1^+,\varepsilon,\varepsilon] \rightarrow [\varepsilon,t_1^+,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,\varepsilon,s_1^+] \rightarrow [\varepsilon,\varepsilon,\varepsilon,s_1^+]$
$[r_1^-,\varepsilon,\varepsilon,\varepsilon] \rightarrow [r_1^-,\varepsilon,\varepsilon,\varepsilon]$
$[t_1^-,\varepsilon,\varepsilon,\varepsilon] \rightarrow [t_1^-,\varepsilon,\varepsilon,\varepsilon]$
$[\varepsilon,\varepsilon,r_2^-,\varepsilon] \rightarrow [\varepsilon,\varepsilon,r_2^-,\varepsilon]$
$[\varepsilon,\varepsilon,s_1^-,\varepsilon] \rightarrow [\varepsilon,\varepsilon,s_1^-,\varepsilon]$

Fig. 6. Two dimensional canonicalization transducer

Interchange. Interchange of two loops is an instance of the general linear transducer where $[i, j] \longrightarrow [j, i]$. Conceptually, the transducer essentially swaps recursive symbols from one dimension to the other. The interchange can be accomplished by the following four transitions, $[r_1^-, \epsilon, \epsilon, \epsilon] \longrightarrow [\epsilon, \epsilon, r_2^-, \epsilon]$, $[\epsilon, r_1^+, \epsilon, \epsilon] \longrightarrow [\epsilon, \epsilon, \epsilon, r_2^+]$, $[\epsilon, \epsilon, r_2^-, \epsilon] \longrightarrow [r_1^-, \epsilon, \epsilon, \epsilon]$, and $[\epsilon, \epsilon, \epsilon, r_2^+] \longrightarrow [\epsilon, r_1^+, \epsilon, \epsilon]$. In these transitions, dimensions are interchanged but positive and negative recursive symbols map to the respective recursive symbols in the other dimension.

Note that interchange is defined not only for linear dimensions (that are isomorphic to loops) but also for general recursive dimensions (such as tree traversals) [Sundararajah et al. 2017]. We can associate *empty tapes* as unused negative tapes for general recursive dimensions, thus UniRec interchange rewrites can also swap linear dimensions with non-linear dimensions. This allows UniRec to represent interchanges that swap loops with recursion [Jo and Kulkarni 2011].

Reversal. Reversing a loop captures the linear transformation $[i, j] \longrightarrow [-i, j]$. In UniRec, this can be captured by essentially swapping the positive and negative tapes of a single dimension. In the case of running example, reversal of first dimension can be expressed with the following looping transitions, $[r_1^-, \epsilon, \epsilon, \epsilon] \longrightarrow [\epsilon, r_1^+, \epsilon, \epsilon]$ and $[\epsilon, r_1^+, \epsilon, \epsilon] \longrightarrow [r_1^-, \epsilon, \epsilon, \epsilon]$.

4.3 Canonicalization Transducer

Previously in Section 3.3, we have discussed the concept of *canonicalization* of the iteration space to preserve lexicographic ordering of control words. This section describes a *canonicalization transducer* that automates the canonicalization process in the form of a transducer that can be composed with a (possibly compound) UniRec transducer. Note that unlike normal UniRec transducers that are a one-to-one mapping of one iteration space to another, a canonicalization transducer is a realization of a *many-to-one* function that takes many different control words representing the same instance and maps them to their canonical control word.

Figure 6 shows the two dimensional canonicalization transducer that works for the running example. This is a *nondeterministic finite state transducer* similar to transformation transducers. The canonicalization transducer has two type of edges, *nullifying edges* (color red), which erase symbols from the control words, and *identity edges* (color black), which copy symbols from the input control word to the output control word. In Figure 6, edge loop at state 1, $[r_1^-, r_1^+, \epsilon, \epsilon] \rightarrow [\epsilon, \epsilon, \epsilon, \epsilon]$ is a nullifying edge, and edge loop at state 3, $[\epsilon, r_1^+, \epsilon, \epsilon] \rightarrow [\epsilon, r_1^+, \epsilon, \epsilon]$ is an identical edge.

Conceptually, a canonicalization transducer "cancels out" matched positive and negative symbols from each dimension of the control word. Consider the control word $i = [r_1^- t_1^-, r_1^+ r_1^+ t_1^+, r_2^- s_1^-, r_2^+ r_2^+ s_1^+]$ that represents an instance. The canonical word of this instance is $[t_1^-, r_1^+ t_1^+, s_1^-, r_2^+ s_1^+]$. If we pass i through the canonicalization transducer, the nullifying edges remove matching recursive symbols from first and second dimension at state 1 and at this point the input tapes look as same as the canonical word. After that we must take the state 2, and state 3 path to replicate t_1^- and s_1^- in the negative tapes of first and second dimension using the identical edges. We loop at state 3 to replicate the recursive symbols r_1^+ and r_2^+. Then we must take the path to state 4 and state 14 to replicate t_1^+ and s_1^+. At this point, the input tapes are all empty, we are in an accepting state, and at the output tapes we have the canonical word $[t_1^-, r_1^+ t_1^+, s_1^-, r_2^+ s_1^+]$.

The construction of canonicalization transducer is not arbitrary. The intuition behind its design can be illustrated as follows. At the initial state (state 1), looping edges must be added for every dimension in the intended iteration space. Then a path must be provided to the final state (state 14) from initial state for every possible combination of recursive symbols left in either positive or negative tape of each dimension. Hence there will be 2^k paths, where k is the number of dimensions. The chain of edges in every path can be divided into two halves. The first half contains the chain of edges with identical edges with non-recursive symbols of the nullified tapes in that path. For example, in the path that connects state 1, state 2, state 3, state 4, and state 14, the first two edges $[t_1^-, \epsilon, \epsilon, \epsilon] \rightarrow [t_1^-, \epsilon, \epsilon, \epsilon]$ and $[\epsilon, \epsilon, s_1^-, \epsilon] \rightarrow [\epsilon, \epsilon, s_1^-, \epsilon]$ are identical edges that contain non-recursive symbols of negative tapes of first and second dimensions, since they are nullified in this particular path. The rest of this path contains the identical edges of non-recursive symbols in the tapes that are not nullified ($[\epsilon, t_1^+, \epsilon, \epsilon] \rightarrow [\epsilon, t_1^+, \epsilon, \epsilon]$ and $[\epsilon, \epsilon, \epsilon, s_1^+] \rightarrow [\epsilon, \epsilon, \epsilon, s_1^+]$). The state which is right in the middle of these two halves (state 3), provides looping identical edges of the recursive symbols that are not nullified in this path ($[\epsilon, r_1^+, \epsilon, \epsilon] \rightarrow [\epsilon, r_1^+, \epsilon, \epsilon]$ and $[\epsilon, \epsilon, \epsilon, r_2^+] \rightarrow [\epsilon, \epsilon, \epsilon, r_2^+]$). Every path from start state to final state are constructed this way.

4.4 Composing UniRec Transducer and Canonicalization Transducer

Composing a (possibly compound) UniRec transducer and a canonicalization transducer produces a new transducer that implements the scheduling transformation and produces only canonical control words, which can be compared using lexicographic order. It is exactly this composed transducer that we use to check the soundness of a transformation sequence (Section 5).

The following are the steps to compose paths of a UniRec transducer with a canonicalization transducer, illustrated through an example of skew transducer (Figure 5) and canonicalization transducer (Figure 6).

- Complete the transitions with recursive symbols of UniRec transducer with non-recursive symbols (e.g. Transition $[r_1^-, \epsilon, \epsilon, \epsilon] \rightarrow [r_1^-, \epsilon, r_2^-, \epsilon]$ is completed as $[r_1^- t_1^-, t_1^+, s_1^-, s_1^+] \rightarrow [r_1^- t_1^-, t_1^+, r_2^- s_1^-, s_1^+]$).
- If the completed transition got recursive symbols on both positive and negative tape of a dimension, compose it with the nullifying transition of canonical transducer.
- After nullifying matching recursive symbols, take all possible paths to compose the rest. This means if there is one recursive symbol left for each dimension then, there exist only one path

(e.g. $[r_1^- t_1^-, t_1^+, s_1^-, s_1^+] \rightarrow [r_1^- t_1^-, t_1^+, r_2^- s_1^-, s_1^+]$ is fully composed with transitions in the state 1, state 11, state 12, state 13, and state 14). Otherwise based on the recursive symbols left, compose with multiple paths.

The result of this composition pushes the looping transitions of UniRec transition from the initial state to the middle of identity transitions of non-recursive symbols in a canonicalized fashion.

5 DEPENDENCES AND CHECKING SOUNDNESS

In this section, first we discuss the dependence representation used in UniRec. This representation is similar to the representation in PolyRec with positive and negative tapes. We discuss challenges associated with dependence representation and see how this can be fixed by the canonicalization. At the end of this section we argue that the dependence test of PolyRec is valid for UniRec.

5.1 Witness Tuples in UniRec

A dependence relation is a set of pair of instances, where every *source* instance precedes its *sink* instance. A *witness tuple* is basically a generator for this set of pair of instances [Sundararajah and Kulkarni 2019]. UniRec's formal definition of witness tuple is the same as in PolyRec:

Definition 5.1. A witness tuple $\langle R_\alpha, (R_\beta, R_\gamma)\rangle$ *captures* a set of dependences $D \subseteq R_{\mathcal{A}} \times R_{\mathcal{A}}$ (pairs of instances from $R_{\mathcal{A}}$) if: $\forall (x, y) \in D. \exists a \in R_\alpha, b \in R_\beta, c \in R_\gamma. x = a \cdot b \wedge y = a \cdot c$.

In other words, if the witness tuple can generate all dependence pairs in D (note: $\cdot$ is a concatenation operation).

Note that witness tuples are describing sets of pairs of instances in the original iteration space. Because UniRec's negative tapes only come into play once transformations are applied, it is apparent that witness tuples themselves do not need to use the negative tapes at all: a UniRec witness tuple is thus a straightforward extension of a PolyRec tuple, with no negative recursive symbols. Recall that the PolyRec witness tuple for the running example is $\langle [(r_1)^*, (r_2)^*], ([t_1, r_2 s_1], [r_1 t_1, s_1])\rangle$. The corresponding UniRec witness tuple is $\langle [\epsilon, (r_1^+)^*, \epsilon, (r_2^+)^*], ([t_1^-, t_1^+, s_1^-, r_2^+ s_1^+], [t_1^-, r_1^+ t_1^+, s_1^-, s_1^+])\rangle$

5.2 Checking Soundess

Once we have UniRec witness tuples, the soundness check mirrors that of PolyRec's, proceeding prefix-by-prefix, with special handling to manage canonicalization. So for a given prefix $a \in R_\alpha$:

(1) Run a through the composed transformation transducer, T. The states Q that a transitions to (it will not reach final states) are used as the start states in a *prefix transducer* T_α that is otherwise the same as T. Note that this is the *non-canonicalized* transformation transducer. T_α represents the "intermediate state" of running a control word through the transformation transducer.
(2) Generate suffix transducers $T_{\alpha,\beta}$ and $T_{\alpha,\gamma}$ by composing the identity transducers for R_β and R_γ, respectively, with T_α.
(3) Generate *canonicalized suffix automata* by composing $T_{\alpha,\beta}$ and $T_{\alpha,\gamma}$ with canonicalization transducers and projecting the output tapes. We call the automata generated from $T_{\alpha,\beta}$ the *source* suffix automaton, and the one generated from $T_{\alpha,\gamma}$ the *sink* suffix automaton.
(4) Compare the *latest* possible k-string generated by the source suffix automaton and the *earliest* possible k-string generated by the sink suffix automaton. As long as the former lexicographically precedes the latter, the transformations preserve the dependences with prefix a

UniRec then enumerates all possible prefix transducers (as in PolyRec) to show that all dependences generated by the witness tuple are preserved.

Note that steps 1 and 2 are identical to the PolyRec dependence test, and interested readers are referred to Sundararajah et al. [2017] for a more detailed explanation of how these transducers and automata are constructed. Step 3 is new to UniRec: it canonicalizes the strings so that lexicographic comparison is possible. Step 2's composition uses Algorithm 1, while Step 3 uses the path composition described in Section 4.4.

The soundness of this dependence test follows from the definition of the witness tuple: for a given $a \in \alpha$, we know that for any source suffix $b \in \beta$ and sink suffix $c \in \gamma$, $a \cdot b \prec a \cdot c$. In the transformed space, if the *latest* possible string generated from the source suffix is earlier than the *earliest* possible string generated from the sink suffix, then the relative ordering of all the pairs of instances is preserved. The key challenge in UniRec is showing that after canonicalization, it is still decidable to find the latest (or earliest) string from a suffix automaton (Step 4 in the procedure).

Lemma 5.2. *From a suffix automaton constructed according to* UniRec*'s composition procedure , it is possible to find the earliest (or latest) possible string.*

Proof. We first note that if an automaton only generates recursive symbols on the positive (or negative) tape for any dimension, then PolyRec's tape-by-tape construction for finding earliest (latest) strings works. We proceed by showing how, for each dimension of the suffix automaton, either the positive tape or the negative tape can be dropped without affecting the results of the comparison, resulting in a single tape per dimension, allowing for PolyRec's comparison process to work.

The *prefix* k-string used to generate the suffix automaton is finite. We then consider the *suffix* relation. Each dimension of the suffix relation is either *finite* (contains a fixed number, possibly zero, of recursive symbols) or *infinite* (contains $(r^+)^*$).

As described in Section 4.2.3, general UniRec transducers can be thought of as transformation matrices that place a linear combination of recursive symbols from input dimensions onto the output dimension (e.g., $a * i + b * j$, where i is the number of recursive input symbols on the first dimension and j is the number of recursive input symbols on the second).[6] Thus, the composition of those transducers also is a linear combination of recursive symbols.

For each dimension of the linear combination, we consider the suffix relation. If all of the suffix dimensions involved in the linear combination (i.e., with non-zero coefficients) are finite, then the output dimensions is also finite, and canonicalization will result in either the positive or negative tape being empty. (Specifically, if the suffix relation dimensions are finite, they can be combined with the prefix, the linear combination can be computed, and we can determine whether the result is positive or negative.) In this case, the empty tape can be dropped from the suffix automaton.

If a dimension in the suffix relation is infinite, then the linear combination of symbols cannot be computed statically (indeed, because this is equivalent to i, j, or both, being infinite, the combination can take on infinite possible values). We further note that i, j, etc., must be positive (because the witness tuple relations only use positive symbols).

Assume that we are computing the earliest string. By examining the coefficients of the linear combination, we can determine whether it is possible for the combination to be $-\infty$ (a negative coefficient multiplied by an infinite input dimension). In this case, there must be a string whose canonicalization still permits an infinite number of symbols on the negative tape (and hence an infinitely "early" instance in that dimension). We can thus drop the positive tape of the suffix automaton without affecting the computation of "earliest." Otherwise, we drop the negative tape.

[6]Note that if the input dimension is non-linear, the only transformations available are interchange, which are permutation transformations that can always be applied at the end, and code motion, which only affects alphabet ordering. Hence, without loss of generality, we can assume that we are dealing with transformations that use linear dimensions as inputs.

If instead we are computing the latest string, we determine whether it is possible for the linear combination to be $+\infty$ and, if so, drop the negative tape, otherwise drop the positive tape.

Thus, we can always reduce the suffix automaton to one that has but a single tape in each dimension. Once we have such an automaton, PolyRec's comparison procedure applies, and the overall decision procedure proceeds as above. □

Operationally, UniRec does not directly compute the linear transformations on each tape. Rather, it can determine whether any tape goes to $\pm\infty$ by checking if there is a non-empty looping transition on that tape after composition with the canonicalization transducer. It can then keep the necessary tapes based on whether the given automaton is queried for the earliest or latest string.

6 IMPLEMENTATION

In this section we discuss the implementation of UniRec framework and associated challenges. The implementation of UniRec closely follows the implementation of PolyRec. The major differences between the implementation of PolyRec and UniRec is the construction of transformation transducers, canonicalization transducers and the composition routines. We take the same approach as in PolyRec with code generation where code can be generated independent of the process of checking correctness of a sequence of transformations. Once UniRec determines the sequence of transformations is safe it is straightforward to generate the transformed code.

Generation of transducers. Both UniRec transformation transducers and canonicalization transducer depends on size of the nest. The canonicalization transducer can be produced only from the number of dimensions k. In this work our main focus is linear recursion; UniRec's treatment of non-linear (general) recursion is identical to PolyRec's. Hence, any UniRec transformation transducer transforming linear recursion can be directly produced from unitary matrices used in the unimodular framework that are formed by composing interchange, reversal and skew. For instance, Figure 5 can be generated from the matrix $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$. Intuitively, the 1s in the diagonal correspond to the identity transitions in the initial state and the other 1 corresponds to the transition $[r_1^-, \epsilon, \epsilon, \epsilon] \rightarrow [r_1^-, \epsilon, r_2^-, \epsilon]$. This is the skew size and if it is more than one, there would be that number of r_2^- symbols in the output. Similarly, transducers for interchange, reverse and any other unimodular transformation can be generated. These transducers are composed with each other using the UniRec transducer composition algorithm. The resulting transducer, canonicalization transducer and the witness tuples are used in conjunction to verify the correctness of the sequence of transformations. In our implementation, first we compose the suffixes from witness tuples with the prefix transducer (derived from the composed transformations transducer) and then the result is composed with canonicalization transducer before running the decision procedure. This is a design decision and the composition order does not matter.

Code generation. Unlike PolyRec, code generation becomes sophisticated in this work due to skewing, because it changes the boundaries of the iteration space. Hence, *bound computation* is an important part of the code generation strategy. Our UniRec transducers do not store any information related to bounds of the iteration space. Hence, we compute bounds for linear dimensions as done in the unimodular framework [Banerjee 1991; Wolfe 1986]. But in our case not all *induction variables* are integers. Some of them are references to *nodes* of a pointer-based data structure (e.g., a cur pointer pointing to the current node in a linked list). Therefore, for each *node-based* induction variable, we create an integer variable called an *auxiliary induction variable.* Auxiliary induction variables are updated simultaneously with their corresponding node-based induction variables as shown in our transformed version of the running example in Figure 1b, in line 6 and line 7. When we have an auxiliary induction variable, we can perform *Presburger arithmetic* reasoning on them. The ability to perform the same style of reasoning on top of a pointer-based recursive structure that uses a node-based induction variable is sufficient for generating code. In simpler terms, if we can define update functions for the node-based induction variable that adhere the rules of addition and subtraction on integers, we can generate transformed code in our framework.

When induction variable is node-based, the bounds of the recursion in that dimension would be based on the count of the nodes in the data structure (i.e., length). In other words, bounds in transformed nests are parameterized by the length of the recursive data structure. In the case where a structure's length is unknown, we add glue code to keep track of the length and this glue code comes with booleans and integer variables getting passed. This significantly complicates guard conditions in transformed code. We have not shown this in our running example for the purposes of readability. Also, generating such glue code is not a fully automated procedure, since it requires hints about the data structure on which the nest operates. Our code generator is implemented as an embedded DSL (Domain Specific Language) in C++.

In most cases, we explore parallelism after transforming the sequential code and we achieve that by adding OpenMP annotations at suitable places in the transformed code. This final step is done manually and can be easily automated.

7 EVALUATION

Our evaluation of UniRec targets two questions. First, whether our framework is capable of *correctly checking the validity of the composed* UniRec *transformations* on perfectly-nested recursive codes. We investigate several case studies that use different data structures with varying dependence structures, and study compositions involving all three UniRec transformations (interchange (IC), reversal (RV), and skew (SK)). Second, whether UniRec *transformations are useful.* We show that these transformations can enhance the performance of complex code: careful, composed transformations can make seemingly serial codes parallelizable.

Experimental Platform. We have written our nested recursive traversals using a DSL embedded in C++ for the convenience of generating transformed code, and used *GCC 7.5.0* to compile them. Parallel execution is achieved using manually inserted *OpenMP* pragmas. The execution platform for performance runs is a 4-core Intel Core i7-8650U 1.90 GHz CPU with 128 KB of L1 cache, 1 MB of L2 cache and 8 MB of L3 cache running Ubuntu 20.04.

Benchmarks. The recent interest in using pointer-based data structures such as lists and trees for storing sparse tensors [Chou and Amarasinghe 2021; Šimeček et al. 2020] has inspired us to categorize our case studies in terms of the type of *Nested Recursive Data Structures* they use. Our benchmark programs are recursive nests performing *PolyBench*-style [pol 2015] computations on these data structures. The order of recursive functions in these nests combined with the locations accessed (reads and writes) by the computation performed within the nest gives rise to different dependence relations. When we discuss these dependence relations, we use the witness tuples and distance vectors interchangeably for brevity.

7.1 Traversals on List of Arrays

Consider a traversal of a doubly linked list where each node contains an array x of size N. Figure 7 shows two such examples. The arrows between the cells of the array indicates read/write relationships between elements of the arrays. We use induction variables n and i to traverse the list and the array, respectively, in their corresponding recursive functions. Based on the order of these recursive functions, we get different dependence structures for the same computation. For instance, Figure 7a shows the dependence structure for the computation `n.x[i] = n.next.x[i+1]+1`

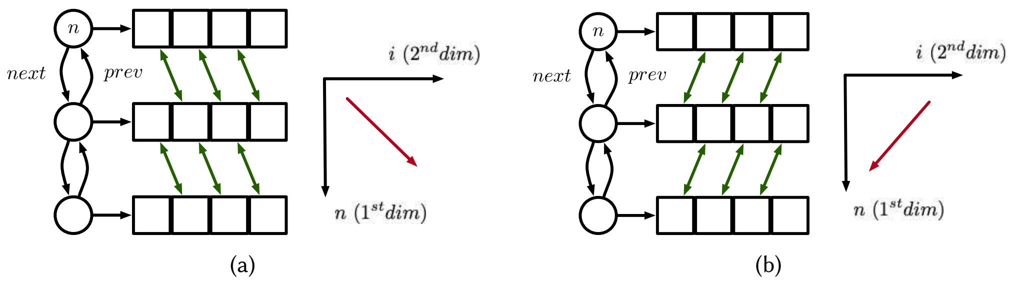


Fig. 7. List of array data structures (a) Traversal (n, i) with dependence $[1, 1]$ for n.x[i] = n.next.x[i+1]+1 (b) Traversal (n, i) with dependence $[1, -1]$ for n.x[i] = n.next.x[i-1]+1

where outer function traverses the list and inner function traverses the array. The witness tuple for this dependence is $\langle [(r_1^-)^*, (r_1^+)^*, (r_2^-)^*, (r_2^+)^*],\ ([t_1^-, t_1^+, s_1^-, s_1^+],\ [t_1^-, r_1^+ t_1^+, s_1^-, r_2^+ s_1^+])\rangle$, corresponding to a $[1, 1]$ distance vector. For brevity, we henceforth use distance vectors unless a full witness tuple is needed for disambiguation.

Considering traversal (i, n) for the computation n.x[i] = n.next.x[i+1]+1, the dependence is $[1, -1]$ Figure 7b shows that for traversal (n, i) and computation n.x[i] = n.next.x[i-1]+1 the dependence is $[1, -1]$. For the same computation if we consider the traversal (i, n) then the dependence is $[1, 1]$.

In these four cases (two dependence structures, two nesting orders), UniRec selects transformation sequences so that at least one "loop" can be parallelized. Reversal is denoted RV(I) and RV(O) for the inner and outer dimensions, respectively. Skew is abbreviated as SK(I, k) for skewing inner dimension wrt outer dimension and SK(O, k) for skewing outer dimension wrt to inner dimension by a constant. Interchange is denoted IC. UniRec validates all of these transformations.

Outer recursion parallelism: Table 1 shows the performance results of transforming the programs to make the outer loop parallel. The cases where the array loop is on the outside scale well, as OpenMP can naturally parallelize code over arrays. When the outer loop is over a linked list, the parallel code spawns an OpenMP task for each outer iteration, leading to overhead. Thus, although the code scales as threads are added, the serial code is faster. Table 3 shows the time of serial code vs. generated code without any parallelism constructs with -O3 optimizations. We observe that generated serial code is slower in some cases because of the bound checks and leaving recursive functions as it is without converting them into loops. This may have made difficult for the compiler to perform classical transformations.

Inner recursion parallelism: Table 2 shows the four cases of transforming the code to make the inner loop parallel, and then interchanging it to the outer loop. Once these transformations are done, this case looks just like the previous case. So, for example, the transformed, parallelized code from the case in Row 1 of Table 2 looks just like the transformed, parallelized code from the case in Row 1 of Table 1.

We observe that this demonstrates the value of composing transformations: because having the array loop on the outside yields faster code, we can build a composed transformation that *both* makes the array loop parallel and moves it to the outside.

7.2 Traversals on Array of Lists

Next, we consider an imperfect rectangular mesh, as seen in Figure 8, implemented as arrays of lists that connect to each other. Consider the traversal (i, j) where it goes through every vertical linked list and perform the computation n.next.x += n.x. This traversal has an unusual dependence structure that is normally not considered in unimodular framework. Due to the reductive

Table 1. Outer recursion parallelism [$N = 10000; length = 10000$].

Traversal	Computation	Dependence	Transformations	Serial Time	Parallel Time #Th. 4	Parallel Time #Th. 8
(i, n)	n.x[i] = n.next.x[i-1]+1	$[+1, -1]$	SK(O,+1)*RV(I) or SK(I,+1)	1952 ms	541 ms	339 ms
(i, n)	n.x[i] = n.next.x[i+1]+1	$[+1, +1]$	SK(I,+1)*RV(I) or SK(I,-1)	1998 ms	553 ms	360 ms
(n, i)	n.x[i] = n.next.x[i-1]+1	$[+1, -1]$	SK(O,+1)*RV(I) or SK(I,+1)	67 ms	1352 ms	795 ms
(n, i)	n.x[i] = n.next.x[i+1]+1	$[+1, +1]$	SK(I,+1)*RV(I) or SK(I,-1)	78 ms	560 ms	345 ms

Table 2. Inner recursion parallelism [$N = 10000; length = 10000$].

Traversal	Computation	Dependence	Transformations	Generated code
(n, i)	n.x[i] = n.next.x[i-1]+1	$[+1, -1]$	(SK(I,+1) or SK(I,-1)*RV(I))*IC	Same as row 1 of Table 1
(n, i)	n.x[i] = n.next.x[i+1]+1	$[+1, +1]$	(SK(I,-1) or SK(I,+1)*RV(I))*IC	Same as row 2 of Table 1
(i, n)	n.x[i] = n.next.x[i-1]+1	$[+1, +1]$	(SK(I,-1) or SK(I,+1)*RV(I))*IC	Same as row 3 of Table 1
(i, n)	n.x[i] = n.next.x[i+1]+1	$[+1, -1]$	(SK(I,+1) or SK(I,-1)*RV(I))*IC	Same as row 4 of Table 1

Table 3. Serial efficiency of the generated code (-O3)[$N = 10000; length = 10000$].

Traversal	Computation	Dependence	Serial Time	Serial Time of Gen. Code
(i, n)	n.x[i] = n.next.x[i-1]+1	$[+1, -1]$	1992 ms	2060 ms
(i, n)	n.x[i] = n.next.x[i+1]+1	$[+1, +1]$	1968 ms	1943 ms
(n, i)	n.x[i] = n.next.x[i-1]+1	$[+1, -1]$	73 ms	4883 ms
(n, i)	n.x[i] = n.next.x[i+1]+1	$[+1, +1]$	84 ms	1974 ms

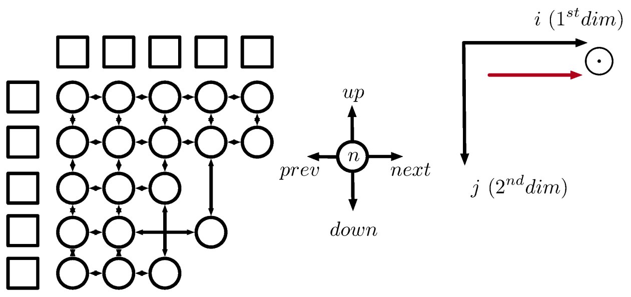


Fig. 8. Traversal (i, j) with the computation n.next.x += n.x.

nature of the computation in the horizontal direction, every iteration depends on all the iteration that happen before them. Hence, the dependence can be written as $[i, 0]$ in distance vector and $\langle[(r_1^-)^*, (r_1^+)^*, (r_2^-)^*, (r_2^+)^*], ([t_1^-, t_1^+, s_1^-, s_1^+], [t_1^-, (r_1^+)(r_1^+) * t_1^+, s_1^-, s_1^+])\rangle$ in witness tuple. We perform IC to make the outer dimension parallel. We ran this traversal on a mesh with the max size of 10000×10000. The serial code took 637 ms to complete and when we ran the parallelized code with 4 and 8 threads, it took 229 ms and 208 ms respectively.

7.3 Traversals on Tree of Arrays

Finally, we consider a binary tree where each node contains array x of size N. Figure 9 shows two iterations of a computation where the outer loop repeatedly traverses the tree, and in each iteration accesses different elements in the nodes' arrays. The broken arrows indicate what locations are accessed during an instance of the computation. Similar to Section 7.1 we use n and i to traverse tree and arrays, respectively. Note that the recursive traversal of the tree *is not linear*—it is a *general* recursion. If we observe the iteration space of tree recursion and compare it with linear recursion, we can see that the only difference here is an additional recursive symbol. PolyRec has already shown how to perform interchange and reversal on this kind of recursive nest. In UniRec, we

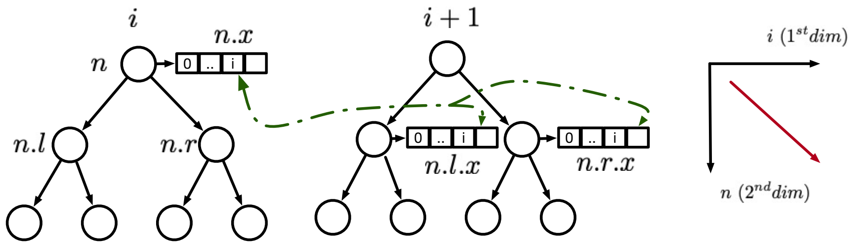


Fig. 9. Tree traversal (i, n) with computation n.x[i] = n.l.x[i+1] + n.r.x[i+1]

are left to handle skewing for tree recursion. *Note that no prior framework can handle this type of transformation on this type of program.*

We have already seen examples of skewing a dimension that uses n as an induction variable w.r.t. a linear dimension in Section 7.1. UniRec generalizes this transformation to skew *both* recursive symbols by the same amount. In a tree recursion, the number of recursive symbols on the tree dimension corresponds to the depth where the computation is performed in that tree. Intuitively, this means we are skewing the tree recursion in the direction of its depth. When it comes to skewing of general recursion, we can visualize it as a linear recursion in the direction of its depth. This helps to think about skewing any general recursion w.r.t. a linear one.

Considering the nested recursive traversal of order (i, n) shown in Figure 9, the dependence in witness tuple format is $\langle [(r_1^-)^*, (r_1^+)^*, (r_{2l}^-|r_{2r}^-)^*, (r_{2l}^+|r_{2r}^+)^*],\ ([t_1^-, t_1^+, s_1^-, s_1^+],\ [t_1^-, r_1^+ t_1^+, s_1^-, (r_{2l}^+|r_{2r}^+) s_1^+])\rangle$. This similar to the $[1, 1]$ dependence structure as seen in right side of Figure 9, but instead of one recursive symbol, there are two in the inner dimension. We perform SK(O, -1) or SK(I, -1)*IC obtain a parallel outer dimension. We performed the skew on a tree of depth 20 with arrays of size 20 in its nodes. We parallelize the outer dimension after the interchange. The serial code ran in 520 ms and with 4 and 8 threads, it took 317 ms and 289 ms, respectively.

8 DISCUSSION

In this discussion we qualitatively assess some future directions for UniRec. The focus of this discussion is two fold. First, we look into features that would make the space of transformations richer. Second, we analyze features that may contribute to the completeness and automation of UniRec.

General fusion and other scheduling transformations. The UniRec framework can currently handle all the unimodular transformations (interchange, reversal, and skew). There are other useful transformations for loops such as scaling and retiming ($[i, j] -> [a * i, j]$ and $[i, j] -> [i + c, j]$)that do not change the relative order of instances but produce efficient code. Even though we do not facilitate code generation for these transformations, UniRec can handle this them from a correctness perspective. We leave the problem of improving code generation of UniRec for future work. Improved versions of classical loop transformation frameworks (e.g. polyhedral framework) support loop fusion and fission where multiple loop nests are combined and vice versa, respectively. Similar fusion transformations have been introduced to recursive functions in the past [Rajbhandari et al. 2016a,b; Sakka et al. 2017, 2019]. Considering the representation of iteration spaces and transformations, we can include fusion and fission transformations to UniRec using the same technique as polyhedral framework in which we introduce dummy dimensions to represent the ordering of instances across multiple nests. But, checking the correctness of these transformations may require additional modifications to the dependence representation and soundness check and we will address these challenges in future work.

Data access constraints. The current UniRec framework is mainly centered around the dependences between the iterations of the recursive nest and the transformations that can be performed to this nest. We do not exploit the rich information available in the data structure where the nest operates on. In the case of unimodular framework, the data structure is always a dense multidimensional array. A dense multidimensional array has random access in every dimension independent of other dimensions. Hence, for a given loop nest with affine access and bounds are enough to decide the correctness of a unimodular transformation. But in UniRec, the recursive nests operate on various different data structures as described in Section 7. One dimension of a data structure may be depend on another. For instance, if we consider list of arrays, first you must be able to access the list node in order to access an array element. These constraints sometimes prevent realizing transformations that are correct. But in this work, when we choose transformations we do not consider such data access constraints. This raises many questions, such as finding ways in which representing the data access constraints and constructing a test to check whether a given composed sequence of transformations is possible to realize given the data access constraints. We can even think a step ahead to see whether given data access constraints can be rearranged to realize a given transformation. For example, if a list of list can be converted into a two dimensional array then it opens up the space for more transformations. This is commonly known as a *Data Layout Transformation.*

Generalized dependence analysis. An under-explored area of inquiry in the space of transforming recursive nests is automatically finding the dependence relations among iterations. In UniRec, we can check whether a sequence of transformations is correct or not if the dependence relations are given as witness tuples. But yet we do not have a fully automated way to generate this dependence relations for the class of programs we consider. This is usually known as a *Dependence Analysis.* Loop transformation frameworks usually use the *Omega Test* [Pugh 1991] which is restricted to dense multidimensional arrays with affine access functions. A *Generalized Dependence Analysis* boils down to incorporating the access to the data structure the program operates on. There are a few dependence analyses that work on recursive programs. For instance, *Tree Dependence Analysis* [Weijiang et al. 2015] works on tree traversals which is a variation of Larus and Hilfinger [Larus and Hilfinger 1988]'s dependence analysis. But none of them are general enough to apply on the programs that we consider. Having an automated, generalized dependence analysis helps to choose transformations. For example, we can choose a transformation that makes a dimension completely independent by looking at the dependence relation/witness tuple. We envision that by encoding the data structure access constraints, and mapping between iterations and read/write accesses, we should be able to get a dependence relation. We can then extract the witness tuple representation from this dependence relation. This would give us the full automation of the transformation pipeline in UniRec.

9 CONCLUSION

PolyRec was the first compositional framework for recursive programs that provides a comprehensive strategy for representing schedules, transformations of those schedules, and dependences, allowing nested recursive programs be soundly transformed. But PolyRec does not cover the full generality of loop transformation frameworks, due to its inability to capture transformations like skewing. We extended PolyRec to have a new representation that is amenable to representing skewing for any general linear recursion. We call this framework UniRec. We showed that despite UniRec's more complicated representation, it can still reason in a decidable manner about the soundness of transformations on programs with recursion and loops. Hence, it fully subsumes the unimodular transformation framework, since UniRec can handle combinations of recursion and loops, and all the unimodular transformations. Our prototype implementation is able to analyze composed unimodular transformations on a perfect recursive nest that operates on various pointer-based data structures, yielding performance improvements from both locality and parallelism that no prior framework could expose.

ACKNOWLEDGMENTS

We appreciate the feedback from the anonymous reviewers for their suggestions and comments that helped to improve this paper. This work was supported in part by the National Science Foundation awards CCF-1908504 and CCF-1919197. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

2015. PolyBench: Polyhedral Benchmark Suite. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/ Accessed: 2022-09-16.

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

Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling stencil computations to maximize parallelism. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2012.107

Utpal Banerjee. 1991. Unimodular Transformations of Double Loops. In *Languages and Compilers for Parallel Computing*.

Ian J. Bertolacci, Catherine Olschanowsky, Ben Harshbarger, Bradford L. Chamberlain, David G. Wonnacott, and Michelle Mills Strout. 2015. Parameterized Diamond Tiling for Stencil Computations with Chapel Parallel Iterators. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 197–206. https://doi.org/10.1145/2751205.2751226

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* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

Stephen Chou and Saman Amarasinghe. 2021. Dynamic Sparse Tensor Algebra Compilation. https://doi.org/10.48550/ARXIV.2112.01394

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

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 Iannetta, Laure Gonnord, and Gabriel Radanne. 2021. *Parallelizing Structural Transformations on Tarbres.* Technical Report RR-9405. 21 pages. https://hal.inria.fr/hal-03208466

Youngjoon Jo and Milind Kulkarni. 2011. Enhancing Locality for Recursive Traversals of Recursive Structures. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. ACM, New York, NY, USA, 463–482. https://doi.org/10.1145/2048066.2048104

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

Ken Kennedy and John R. Allen. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

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

Salwa Kobeissi, Alain Ketterlin, and Philippe Clauss. 2020. Rec2Poly: Converting Recursions to Polyhedral Optimized Loops Using an Inspector-Executor Strategy. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Alex Orailoglu, Matthias Jung, and Marc Reichenbach (Eds.). Springer International Publishing, Cham, 96–109.

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* (Atlanta, Georgia, USA) *(PLDI '88)*. ACM, New York, NY, USA, 24–31. https://doi.org/10.1145/53990.53993

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* (Albuquerque, New Mexico, USA) *(Supercomputing '91)*. ACM, New York, NY, USA, 4–13. https://doi.org/10.1145/125826.125848

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

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* (Salt Lake City, Utah) *(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* (Barcelona, Spain) *(CC 2016)*. ACM, New York, NY, USA, 152–162. https://doi.org/10.1145/2892208.2892228

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

Laith Sakka, Kirshanthan Sundararajah, Ryan R. Newton, and Milind Kulkarni. 2019. Sound, Fine-Grained Traversal Fusion for Heterogeneous Trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 830–844. https://doi.org/10.1145/3314221.3314626

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

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

Ivan Šimeček, Claudio Kozický, Daniel Langr, and Pavel Tvrdík. 2020. Space-Efficient k-d Tree-Based Storage Format for Sparse Tensors. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (Stockholm, Sweden) *(HPDC '20)*. Association for Computing Machinery, New York, NY, USA, 29–33. https://doi.org/10.1145/3369583.3392692

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

Michael Wolfe. 1986. Loop Skewing: The Wavefront Method Revisited. *Int. J. Parallel Program.* 15, 4 (Oct. 1986), 279–293. https://doi.org/10.1007/BF01407876