Equivalence Checking of Static Affine Programs using Widening to Handle Recurrences[1]

SVEN VERDOOLAEGE, GERDA JANSSENS and MAURICE BRUYNOOGHE
Department of Computer Science, Katholieke Universiteit Leuven, Belgium

Designers often apply manual or semi-automatic loop and data transformations on array and loop intensive programs to improve performance. It is crucial that such transformations preserve the functionality of the program. This paper presents an automatic method for constructing equivalence proofs for the class of static affine programs. The equivalence checking is performed on a dependence graph abstraction and uses a new approach based on widening to find the proper induction hypotheses for reasoning about recurrences. Unlike transitive closure based approaches, this widening approach can also handle non-uniform recurrences. The implementation is publicly available and is the first of its kind to fully support commutative operations.

Categories and Subject Descriptors: D.2.4 [**Software/Program Verification**]: Correctness proofs

General Terms: Verification
Additional Key Words and Phrases: Commutativity, equivalence checking, polytope model, recurrences, widening

1. INTRODUCTION

Embedded processors for multimedia and telecom systems are severely resource constrained. Developers apply aggressive loop and data transformations based on a combination of automated analysis and manual interventions to reduce memory requirements and power consumption (see, e.g., [Catthoor et al. 2002; Verma and Marwedel 2007]). A crucial question is whether the transformed program is equivalent to the original. We address this problem for the case of static affine programs, i.e., programs with static control flow and piecewise affine expressions for all loop bounds, conditions and array index expressions.

Our method works on a dependence graph abstraction of the program and reasons backward. Using a pair of dependence graphs, the goal of showing that both programs output the same data is reduced to subgoals between the corresponding children of the dependence graphs. A subgoal is discharged as proved when reduced to an equivalence between inputs. This success is also propagated forward in order to compare what actually has been proved with what had to be proved. An induction hypothesis is generated when it is noticed that a subgoal depends on an earlier iteration of a loop (a recurrence).

Generating an adequate induction hypothesis for recurrences is a major challenge in the equivalence checking of static affine programs. In previous work of Barthou et al. [2002], Alias and Barthou [2003], Shashidhar et al. [2005][2] and Shashidhar

[2]Joint work with some of the current authors.

[2008], the transitive closure operation [Kelly et al. 1996] provided by the `Omega` library [Kelly et al. 1996] is used to derive the hypothesis from the base case obtained from analyzing one iteration over the recurrence. This effectively restricts the applicability of those techniques to programs containing only uniform recurrences.

Another challenge is posed by algebraic transformations, i.e., a transformation that depends on algebraic properties of operations, e.g., associativity or commutativity. Of the above, only Shashidhar has a proposal for handling algebraic transformations. Moreover, it is unable to handle transformations that only reorder the arguments of an operation for some iterations of the loops in which the operation appear, as is the case in the example of the next section.

Finally, we note that all the above approaches require both programs to be in dynamic single assignment (DSA) form [Feautrier 1988], i.e., such that each array element is written at most once. Also, none of the aforementioned implementations are publicly available.

Like the previous approaches, we handle recurrences in both programs *fully automatically* and we handle any per statement or per array piecewise quasi-affine loop or data transformation, including combinations of loop interchange, loop reversal, loop skewing, loop distribution, loop tiling, (partial) loop unrolling, loop splitting, loop peeling and data-reuse transformations. However, unlike those approaches, ours

—integrates dependence analysis such that the explicit conversion to dynamic single assignment source code in a preprocessing step, as required by the other approaches, can be avoided,

—handles both uniform and non-uniform recurrences by not relying on a transitive closure operation but instead using a novel widening approach to find the proper induction hypothesis for reasoning about recurrences, and

—has a publicly available implementation,

—with full support for associative and commutative operations with a fixed number of arguments.

It should be noted that we only consider program transformations that change the order of operations, change the location where intermediate results are stored and/or exploit the associative and/or commutative properties of operations. In particular, we do not consider transformations that replace a sequence of operations, e.g., `x1 = x + 1; y = x1 + 1`, by some other sequence of operations, e.g., `y = x + 2`, but instead treat operations as black boxes.

Our approach is shown to handle more cases than related approaches and is shown to scale better on pairs of implementations of the USVD algorithm, which is frequently used in wireless signal processing applications. Our approach has also been instrumental in finding bugs in various optimizations in the `CLooG` [Bastoul 2004] polyhedral scanner. Our experiments show that other approaches would not have been as useful as they are unable to prove equivalence on many inputs, resulting in too many false positives.

This paper is a revision and extension of our earlier work [Verdoolaege et al. 2009]. In particular, the equivalence checking procedure has been formalized as a pair of algorithms with proof of termination and soundness. The procedure has

```
1 A[0]=In[0];
2 for (i=1; i<N; ++i)
3   A[i]=f(In[i])+g(A[i-1]);
4 Out=A[N-1];
```

(a) Program 1, a program with a recurrence

```
1  A[0]=In[0];
2  for (i=1; i<N; ++i) {
3    if (i%2 == 0) {
4      B[i]=f(In[i]);
5      C[i]=g(A[i-1]);
6    } else {
7      B[i]=g(A[i-1]);
8      C[i]=f(In[i]);
9    }
10   A[i]=B[i]+C[i];
11 }
12 Out=A[N-1];
```

(b) Program 2, a program equivalent to Program 1

Fig. 1. Two equivalent programs, assuming that + is commutative.

been validated on a larger set of examples and compared against two other procedures. One is our own reimplementation of the approach of Barthou et al. [2002], the original implementation not being available to us. The other is a simplified variation of Barthou's approach. We have also extended our work to handle more general constructs such as data-dependent accesses in a separate paper [Verdoolaege et al. 2010]. With minor modifications, the termination and soundness proof of the present paper is also applicable to those extensions.

We start in Section 2 with a high level description of our method on the basis of a toy example. In Section 3 we introduce some background material and define the dependence graphs which are used to represent the input programs. Section 4 contains the description of our equivalence checking method. A recursive algorithm propagates requirements for equivalence to base cases and then propagates the results back. We explain our handling of commutative operations and recurrences and we prove termination and soundness. In Section 5 we discuss some implementation details, while in Section 6 we present some examples that highlight the differences with other approaches and show the results of some experiments. Finally, in Section 7, we discuss in more depth related work and we conclude in Section 8.

2. ILLUSTRATIVE EXAMPLE

Figure 1 shows a contrived example designed to illustrate some key aspects of our method. Both programs have the same input array `In` and output array `Out` and the objective is to show that for any value of the input array(s), both programs produce the same value for the output array(s). Note that in this particular example, there is only one input and one output array and that this output array is zero-dimensional, i.e., it is a scalar. The other arrays are called temporary arrays and we assume no a priori relation between temporary arrays from different programs. Likewise, we assume no a priori relation between any pair of loops in the two programs.

The example in Figure 1 involves both of the "challenges" in equivalence checking

of affine programs. Both programs have a recurrence, i.e., the computations in the loops depend on previous iterations of the same computation. Such recurrences render the representation of the values of the output arrays as a symbolic expression in terms of only the input arrays, as advocated by some symbolic simulation based techniques (e.g., [Matsumoto et al. 2007]) impractical, as the whole loop needs to be effectively unrolled, or even impossible, if the number of iterations is unknown at analysis time, as is the case in this example. Furthermore, the equivalence of the two programs depends on the commutativity of the `+` operator, as the order of the arguments has been reversed for each iteration of the loop with odd iterator value.

Let us now try to prove the equivalence of the two programs at an intuitive level. The proof procedure will be formalized in the following sections. We assume that both programs are given the same input in the array `In` and we want to show that they produce the same value for the array `Out`. To show that `Out` in Program 1 is equal to that of Program 2, we distinguish two cases. If `N` is equal to 1, then we need to show that whatever value is copied in Line 1 of Program 1 is equal to the value copied in Line 1 of Program 2. Otherwise, we need to show that whatever value is computed by the `+` operator in Line 3 of Program 1 is equal to the value computed by the same operator in Line 10 of Program 2. Note that these values happen to be stored in `A[0]` and `A[i]`, with $i = N - 1$, respectively, in both programs, but these locations are irrelevant for the purpose of proving equivalence.

In the first case, i.e., when $N = 1$, we need to show that `In[0]` in one program is equal to `In[0]` in the other program, but this is true by assumption. In the other case, i.e., when $N > 1$, we again need to distinguish two cases depending on whether $i = N - 1$ is odd or even. Let us consider the odd case. The values computed by the `+` operators are certainly equal if the values of both arguments are equal, but since `+` is commutative, the arguments need not be given in the same order. In this (odd) case, we try to prove that the value computed by `f` in Line 3 of Program 1 is equal to that computed by the same function in Line 8 of Program 2, as we are not able to prove that it is equal to the value computed by the (different) function `g` in Line 7 of Program 2. Similarly, we try to prove that what is computed by `g` in Line 3 of Program 1 is equal to what is computed by `g` in Line 7 of Program 2.

The proof obligation through `f` leads us to the input again and is easily seen to hold. The proof obligation through `g`, however, leads us back to a proof obligation on `+` in Line 3 and Line 10 respectively (assuming $i = N - 1 \neq 1$). Specifically, we originally had to prove equivalence for $i_1 = N - 1 = i_2$, with $N \geq 2$ and now we have to prove it for $i_1 = N - 2 = i_2$, with $N = 2\alpha \geq 4$, $\alpha \in \mathbb{Z}$. To avoid getting into an infinite loop, we *generalize* (or "widen") the union of these two relations to $i_1 = i_2$, with $1 \leq i_1 \leq N - 1$. The equivalence proof for this widened relation proceeds in the same way as that for the original relation, including a split into an even and an odd case. In both cases, when we arrive back at operator `+` in Line 3 and Line 10 we apply *induction* and assume that equivalence has already been proved for this pair of computations. After applying induction, there are no more proof obligations. The induction hypothesis will be validated in a second phase, as explained in the following sections. Note that the equivalence of the widened relation implies equivalence for both the cases $i = N - 1$ odd and $i = N - 1$ even.

3. PROGRAM MODEL

Two programs are considered to be equivalent if they produce the same output values given the same input values. As we treat all operations performed inside the programs as black boxes, this means that in both programs the same operations should be applied in the same order on the input data to arrive at the output. For our equivalence checking, we therefore need to know which operations are performed and how data flows from one operation to another. In this section, we introduce a program model that captures exactly this information. Unlike the approach of Shashidhar et al. [2005] and Shashidhar [2008], which works on an array based representation and implicitly performs some dataflow analysis during the equivalence checking, we separate the dataflow analysis from the equivalence checking, the latter working on the output of the former. This separation allows us to use either standard exact dataflow analysis [Feautrier 1991] or, in future work, fuzzy dataflow analysis [Barthou et al. 1997]. The resulting *dependence graph* is essentially a DSA representation of the program, but without rewriting the source program in that form as required by Shashidhar et al. [2005] and Shashidhar [2008]. We first describe the kinds of sets and relations that we will use in our model. Then we explain the restrictions we impose on the input programs. We continue with a brief description of dataflow analysis and then formally define our dependence graph abstraction. After describing a transformation of these dependence graphs to handle associative operations, we finally define the concept of equivalence of dependence graphs.

3.1 Sets and Relations

For modeling an input program and during the equivalence checking, we frequently make use of subsets of $\mathbb{Z}^d$ and of binary relations over such sets. Here, d is some non-negative integer and $\mathbb{Z}$ is the set of integers. The integer tuples in these sets and relations typically represent iterations of a loop or elements of an array. The pairs of integer tuples in a relation are separated by either "$\rightarrow$" or "$\leftrightarrow$". The difference between these two is purely cosmetic. In particular, we use "$\leftrightarrow$" for relations that express equivalences and "$\rightarrow$" for relations that are involved in composition and/or application operations, as described below. In our notation, we usually annotate the integer tuples with the name of the corresponding "computation" or array. For the purpose of the present paper, these names carry no semantics and are only meant to remind the reader to which object an integer tuple belongs. We require that all sets and relations can be described using affine equalities and inequalities, i.e., constraints of the form $a_0 + \sum_i a_i\, x_i = 0$ or $a_0 + \sum_i a_i\, x_i \geq 0$, where the a_i are integer constants and the x_i are integer variables. This requirement ensures that all operations we need to perform on our sets and relations can be performed reasonably efficiently. Note that any of our sets and relations can be described in many different ways using affine constraints. However, all the operations we need to perform are (semantically) independent of the chosen representation.

In particular, we need the following operations.

—The *emptiness check* determines whether a given set or relation has any elements.

—The *intersection* of two relations R_1 and R_2 is the relation $R_1 \cap R_2 = \{\, \mathbf{i} \rightarrow \mathbf{j} \mid \mathbf{i} \rightarrow \mathbf{j} \in R_1 \wedge \mathbf{i} \rightarrow \mathbf{j} \in R_2 \,\}$. Similarly for sets.

—The *union* of two relations R_1 and R_2 is the relation $R_1 \cup R_2 = \{ \mathbf{i} \to \mathbf{j} \mid \mathbf{i} \to \mathbf{j} \in R_1 \vee \mathbf{i} \to \mathbf{j} \in R_2 \}$. Similarly for sets.

—The *difference* of two relations R_1 and R_2 is the relation $R_1 \setminus R_2 = \{ \mathbf{i} \to \mathbf{j} \mid \mathbf{i} \to \mathbf{j} \in R_1 \wedge \mathbf{i} \to \mathbf{j} \notin R_2 \}$. Similarly for sets.

—The *product* of two sets S_1 and S_2 is the relation $S_1 \to S_2 = \{ \mathbf{i}_1 \to \mathbf{i}_2 \mid \mathbf{i} \in S_1 \wedge \mathbf{i}_2 \in S_2 \}$. A similar product can also be defined using the $\leftrightarrow$ operator. It is also convenient to define a product operation on relations. The result of this operation is a "second order" relation, i.e., one that maps relations to relations. In particular, the product of two relations R_1 and R_2 is the relation $R_1 \leftrightarrow R_2 = \{ (\mathbf{i}_1 \leftrightarrow \mathbf{i}_2) \to (\mathbf{j}_1 \leftrightarrow \mathbf{j}_2) \mid \mathbf{i}_1 \to \mathbf{j}_1 \in R_1 \wedge \mathbf{i}_2 \to \mathbf{j}_2 \in R_2 \}$.

—The *composition* of two relations R_1 and R_2 is the relation $R_2 \circ R_1 = \{ \mathbf{i} \to \mathbf{j} \mid \exists \mathbf{k} : \mathbf{i} \to \mathbf{k} \in R_1 \wedge \mathbf{k} \to \mathbf{j} \in R_2 \}$.

—The *application* of the relation R to the set S is the set $R(S) = \{ \mathbf{j} \mid \exists \mathbf{i} : \mathbf{i} \to \mathbf{j} \in R \wedge \mathbf{i} \in S \}$. The application of a second order relation to a (first order) relation is similarly defined.

—The *inverse* of a relation R is the relation $R^{-1} = \{ \mathbf{i} \to \mathbf{j} \mid \mathbf{j} \to \mathbf{i} \in R \}$.

—The *domain* of a relation R is the set $\operatorname{dom} R = \{ \mathbf{i} \mid \exists \mathbf{j} : \mathbf{i} \to \mathbf{j} \in R \}$.

—The *identity relation* on a set S is the relation $1_S = \{ \mathbf{i} \to \mathbf{i} \mid \mathbf{i} \in S \}$.

—The *integer affine hull* of a relation is the smallest affine subspace containing the relation, where an affine subspace is a relation described using only equalities. That is, the integer affine hull of a relation is the relation of elements that satisfy all affine equalities satisfied by all elements in the original relation.

The constraints describing the sets and relations may involve both parameters and existentially quantified variables. The use of parameters allows us to process several instances of the same (pair of) program(s) at the same time. For example, the programs in Figure 1 contain a parameter `N` that determines the size of the `In`, `A`, `B` and `C` arrays and the number of times the loops are executed. By keeping track of this parameter, we can check the equivalence of both programs for any value of `N`. The use of existentially quantified variables allows us to represent *quasi-affine* constraints, i.e., constraints that are affine, but that may contain additional, existentially quantified, variables. For example, the set of iterations (i.e., values of the `i`-iterator) for which the statement in Line 7 of Figure 1(b) is executed, can be represented as

$$D = \{ \mathrm{L}_7(i) \mid \exists \alpha : 1 \leq i < N, i = 2\alpha + 1 \}. \tag{1}$$

The L_7 annotation on the tuples is used to clarify that these tuples represent iterations of the statement in Line 7

3.2 Input Programs

Our equivalence checker takes two static affine programs [Feautrier 1996] or program fragments as input. Here, static means that the control flow is static, i.e., the control flow is independent of the input to the programs. Affine means that all *iteration domains* and all *access relations* can be represented using quasi-affine constraints. An iteration domain is the set of iterator values for which a given statement is executed. For example, the set D in (1) represents the iteration domain

of the statement in Line 7 of Figure 1(b). For simplicity, we assume that the constraints of the iteration domains can be copied directly from the loop bounds and the conditions in the program. All these constructs therefore need to be quasi-affine in terms of the parameters and outer loop iterators. Furthermore, the program is not allowed to contain any `break`s, `continue`s, `goto`s or `while` loops. In principle, more general input programs can also be accepted by first applying abstract interpretation techniques as in PIPS [Irigoin et al. 1991]. We assume that all loops have upper bounds. If not, a fresh parameter can be introduced to act as the upper bound of unbounded loops.

An access relation maps an iteration domain to an index space, i.e., the indices of the elements of an array. For example, the access to the `A` array in the same statement on Line 7 has access relation

$$\{ \mathrm{L}_7(i) \to \mathrm{A}(i-1) \}.$$

Note that this relation maps iterations of the statement in Line 7 (L_7) to elements of the `A` array. Similarly, the access to the `A` array in Line 12 has access relation

$$\{ \mathrm{L}_{12}() \to \mathrm{A}(N-1) \}.$$

Since the corresponding statement is not enclosed in any loops, it has a zero-dimensional iteration domain. The single zero-dimensional element of this domain is denoted "$\mathrm{L}_{12}()$". The fact that we require quasi-affine access relations implies that we do not allow pointer manipulations, unless they have been converted to quasi-affine array accesses in a pointer conversion preprocessing step (see, e.g., [van Engelen and Gallivan 2001; Franke and O'Boyle 2003]).

We assume that like-named functions called within both program fragments are identical (or at least equivalent) and that they are pure. In particular, the program fragments are not allowed to call themselves. That is, we do not allow recursion in those parts of the programs that need to be checked for equivalence. If some of the functions called have been transformed as well, then they should be inlined, provided the inlined functions are also static affine. See, e.g., [Absar et al. 2005].

The programs may manipulate several arrays, where we treat scalars as zero-dimensional arrays. These arrays come in three categories, input arrays, output arrays and temporary arrays. The input and output arrays are assumed to have the same names in both programs. The equivalence checker tries to prove that given the same values for the input arrays, both programs produce the same values for the output arrays. We make no assumption about any correspondence between temporary arrays in both programs. The types of the arrays may be specified by the user or they can be determined automatically. Prior to any such automatic detection, we need to make sure that an input array is never written to since input arrays are assumed to be equal while the two programs may write different values. To ensure this property, we consider all arrays that appear in both programs and (conceptually) create copies of them before the start of the program fragment under investigation. The program text is then modified to refer to the copies. The original arrays are treated as input arrays, while the copies as well as those arrays that have not been copied are treated as either output or temporary arrays. In particular, the output arrays are those that have elements that are written without being subsequently read or overwritten. The detection of subsequent reads or writes

can be performed using dataflow analysis, which is explained in the next section. Note that in practice, we do not actually perform the copying described above, but instead use dataflow analysis to determine whether a given instance of a read access reads from an input array or from an output or temporary array with the same name.

3.3 Dataflow Analysis

In order for us to be able to check the equivalence of two programs, we need to model how these two programs compute the output values from the input values. In particular, for each value used in the program, we need to know where the value was computed. This is the subject of dataflow analysis. In this paper, we apply *exact* dataflow analysis [Feautrier 1991], meaning that for each of the uses of a value in the program, we determine exactly where it was computed. Exact dataflow analysis also requires static affine programs as input and therefore imposes no further restrictions.

The output of dataflow analysis consists of a number of *dependence relations*. Each of these dependence relations maps an element of an iteration domain that reads some value to the unique element of the same or another iteration domain that wrote the value. For example, the statement in Line 4 of Figure 1(a) reads from the `A` array. Dataflow analysis determines that there are two cases. If $N = 1$, then the value read in Line 4, was written in Line 1, with dependence relation

$$\{ \mathrm{L}_4() \to \mathrm{L}_1() \mid N = 1 \}. \tag{2}$$

Otherwise, if $N > 1$, then the value was written in the last iteration of the statement in Line 3, with dependence relation

$$\{ \mathrm{L}_4() \to \mathrm{L}_3(N-1) \mid N \geq 2 \}. \tag{3}$$

3.4 Dependence Graphs

The equivalence checking is performed on abstractions of the input programs that model the computations and the data flow dependences between computations. This program abstraction is formalized in a *dependence graph*. Our dependence graphs differ slightly from those commonly used during the optimization of static affine programs. Most notably, we keep track of the individual operations performed in a statement to be able to match the operations in both programs.

Firstly, we describe the *computations* that are extracted from the program. They are the vertices of the dependence graph. A computation represents one or more instances of an operation f with arity r (the number of arguments of f). Each computation has a fixed dimension $d \in \mathbb{Z}_{\geq 0}$ and each instance of f is identified by an element of $\mathbb{Z}^d$. The set D of all instances is called the iteration domain of the computation. Finally, each computation has a "location" l that can be used to distinguish different computations that perform the same operation. The computations are extracted from a static affine program in the following way.

—for each operation in a line of the program text, a computation is constructed with f and r determined by the operation, l identifying the location in the program text (line and column), d equal to the number of enclosing loops and D equal to the iteration domain of the statement in which the operation appears.

—for each statement that does not perform any operation, but instead simply copies an array element to some array element, a computation is constructed with a special operation "id" of arity 1. The other properties are as in the previous case.

—for each input array, an *input computation* is constructed with as operation the name of the array and arity 0. The dimension is the dimension of the array and the iteration domain is the set of array indices.

—a special one-dimensional input computation $\mathbb{Z}$ with iteration domain $\mathbb{Z}$ that, conceptually, produces the number i in iteration i. This special input computation is only needed if the program contains a statement such as $x = y + i$ inside the body of a loop with loop iteration variable i, where the iteration variable i is used as an operand.

—an *output computation*, with operation Out and arity 1. The dimension is the dimension of the output array and the iteration domain is the set of array indices. If there are multiple output arrays, then the program can be extended with statements copying them into different parts of a single output array, hence we can assume without loss of generality that programs have a single output array.

Input and output computations have a dummy location as there is no corresponding line and column in the program.

Secondly, we describe the *dependences* that are extracted from the program. They are the edges of the dependence graph. A dependence between two computations signifies that one computation iteration produces a value that is used as input by another computation iteration. A dependence is characterized by the two participating computations, an argument position p and a dependence relation M. Dependences arise in four ways:

(1) The value produced by an operation f is used as argument p in an operation g of the same statement. Let u and v be the corresponding computations. As both computations originate from the same statement, they have the same iteration domain D. The dependence relation of the dependence from v to u is the identity relation on D, i.e., $M = \{v(\mathbf{i}) \to u(\mathbf{i}) \mid \mathbf{i} \in D\}$. The computation v is the source of the dependence; u is the target computation.

(2) The value used as argument p by some iterations of an operation f is an element of an input array or an affine expression in the parameters and outer loop iterators. The latter is handled as a read from the "$\mathbb{Z}$" array. A dependence is created from the computation performing f to the appropriate input computation, with as dependence relation the access relation.

(3) The value used as argument p by some iterations of an operation f is last written by some statement s. This statement writes the value to a temporary or output array and the operation reads that value. Let v be the computation corresponding to f and u the computation corresponding to the top-level operation of statement s. The existence of a dependence between v and u as well the corresponding dependence relation M is determined by dataflow analysis as described in Section 3.3.

(4) An element of the output array is last written by a statement s. In this case, a dependence is created from the output computation to the computation corre-

sponding to the top-level operation of statement s. The argument position p is 1 and the dependence relation is the inverse of the access relation of the write access to the output array restricted to those iterations that write an element for the last time.

The computations and dependences form the basis of the dependence graph abstraction.

Definition 3.1 *Dependence Graph.* A *dependence graph* is a connected labeled directed graph $G = \langle V, E, s, t, \lambda_{\mathrm{v}}, \lambda_{\mathrm{e}} \rangle$ with vertices V, called *computations*, and edges E, called *dependences*. There is a designated *output computation* $v_0 \in V$ with in-degree 0 and a set $I \subset V$ of *input computations*, each with out-degree 0.

λ_{v} is a labeling function mapping computations to tuples $\langle d, D, f, r, l \rangle$ where

—d, the *dimension* of the computation, is a non-negative integer.

—D, the *iteration domain* of the computation, is a set of tuples of integers of dimension d.

—f, the *operation* of the computation, is a symbol.

—r, the *arity* of the operation of the computation, is a non-negative integer.

—l, the *location* of the computation is a symbol.

The other elements s, t and $\lambda_{\mathrm{e}}(e)$ define functions on edges: $s(e)$ is the source computation and $t(e)$ is the target computation of an edge; $\lambda_{\mathrm{e}}(e)$ is a labeling function on edges. The label consists of:

—p, the *argument position*, is a positive integer between 1 and $r_{s(e)}$, the arity of the source computation $s(e)$.

—M, the *dependence relation*, is a relation from tuples of integers of dimension $d_{s(e)}$ to tuples of integers of dimension $d_{t(e)}$.

Moreover, the following constraints are satisfied.

(1) The domains of the dependence relations of the dependences emanating from a computation for a given argument position partition the domain of the computation, i.e.,

$$\forall u \in V, \forall \mathbf{x} \in D_u, \forall p \in [1, r_u] : \exists ! \, e \in E : u = s(e) \wedge p_e = p \wedge \mathbf{x} \in \operatorname{dom} M_e.$$

(2) For any cycle in the graph, the composition of the dependence relations M_e along the cycle does not intersect the identity relation, i.e., no element of a domain (indirectly) depends on itself.

(3) For any fixed value of the parameters, the iteration domains of all non-input computations are finite.

Note that a dependence graph may have loops and parallel edges. The effect of a dependence relation M associated to an edge e, with $u = s(e)$ and $v = t(e)$, is twofold. The dependence relation first selects part of the iteration domain D_u of u, in particular, $D_u \cap \operatorname{dom} M$, and then maps those elements from the iteration domain of u to the iteration domain of v. Within the context of the present paper, where we use exact dataflow analysis, the dependence relations are functions. Before arguing that the above extraction procedure results in a dependence graph, i.e., that the resulting graph satisfies the conditions of Definition 3.1, we first give an example.

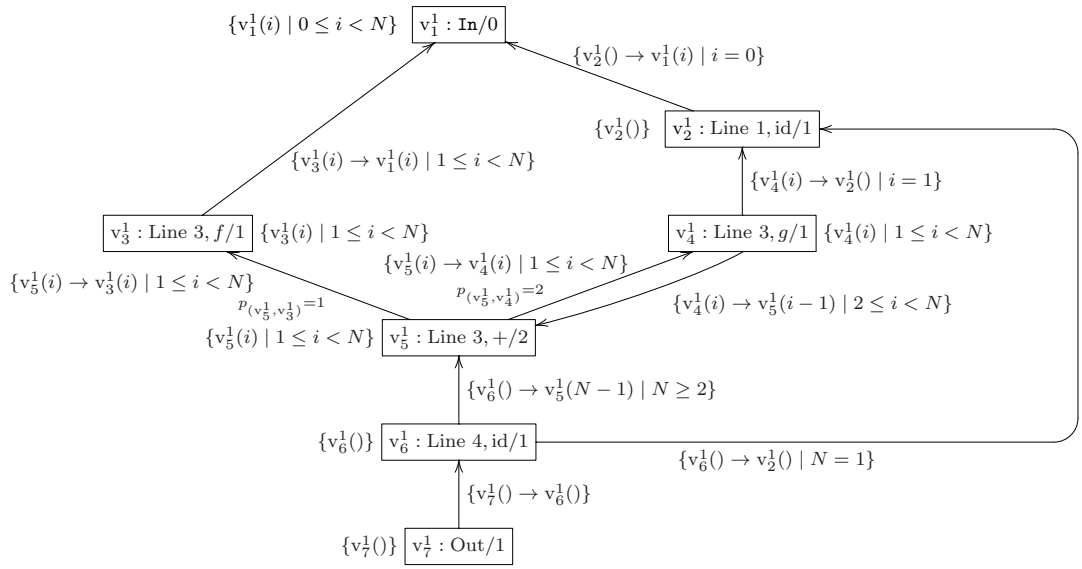


Fig. 2. Dependence graph G_1 of the program in Figure 1(a). Computations are named from v_1^1 to v_7^1 with v_1^1 the input and v_7^1 the output computation. Each computation v is represented as "$v : l, f/r$" (l is omitted for input and output computations). To avoid clutter, the dimension is omitted, while the domain is shown next to the box with the vertex. The dependence relations are shown next to the corresponding edges, while the argument position p_e is only indicated on edges emanating from vertex v_5^1 (it is 1 on all other edges). Domains take as subscript the name of the corresponding vertex and dependence relations the names of the begin and end vertex of the corresponding edge.

Example 3.2. The dependence graph of the program in Figure 1(a) is shown in Figure 2. The graph has one input and one output computation and five other computations, one for the copy statement in Line 4 (v_6^1), one for the addition (v_5^1), one for the computation of f (v_3^1), and one for the computation of g (v_4^1) (all from Line 3) and finally one (v_2^1) for the copy statement in Line 1. The input computation v_1^1 is one-dimensional because `In` is a one-dimensional array. The output computation v_7^1 is zero-dimensional because `Out` is a scalar.

Examples of intra-statement dependences in Figure 2 are the edges from the addition (computation v_5^1) to respectively the computations v_3^1 and v_4^1. As to edges that result from dataflow analysis, the dependence relations $M_{(v_6^1, v_2^1)}$ and $M_{(v_6^1, v_5^1)}$ are those from Equations (2) and (3) in Section 3.3, respectively. Note that the tuple names have been adjusted to refer to the computations. The output array `Out` is only written in Line 4, with access relation $\{L_4() \to \mathrm{Out}()\}$. The dependence relation on the edge (v_7^1, v_6^1) is therefore the (renamed) inverse of this access relation.

Figure 3 similarly shows the dependence graph for the program in Figure 1(b).

Below, the unique edge in G corresponding to a point $\mathbf{x}$ in the iteration domain of node v and an argument position p is denoted $e_G(v, \mathbf{x}, p)$.

Definition 3.3. Given a dependence graph $G = \langle V, E, s, t, \lambda_v, \lambda_e \rangle$ and a computation $v \in V$ (with operation f_v), the *value* of v at $\mathbf{x}$, with $\mathbf{x} \in D_v$ is defined as

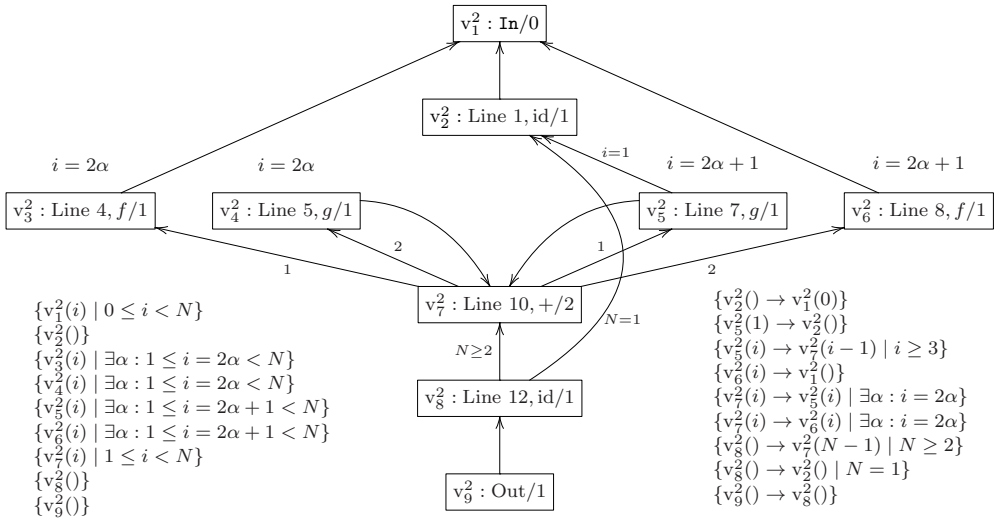


Fig. 3. Dependence graph G_2 of the program in Figure 1(b) with some details omitted. The edges emanating from v_8^2 are annotated with the most salient constraints from their dependence relations. Those emanating from v_7^2 are annotated with their argument positions, while the target computations are annotated with the most salient constraints from their domains. A complete list of domains is shown on the bottom left of the figure, while a partial list of dependence relations is shown on the bottom right.

follows.

—if v is the special input computation $\mathbb{Z}$, then the value of v at $\mathbf{x}$ is $\mathbf{x}$.
—if v is any other input computation, then the value of v at $\mathbf{x}$ is the value of array f_v at position $\mathbf{x}$.
—if v is the output computation or a copy operation, then the value of v at $\mathbf{x}$ is equal to the value of $t(e_G(v, \mathbf{x}, 1))$ at $M_{e_G(v,\mathbf{x},1)}(\mathbf{x})$.
—if v is any other computation, then the value of v at $\mathbf{x}$ is the result of applying f_v to the values of $t(e_G(v, \mathbf{x}, p))$ at $M_{e_G(v,\mathbf{x},p)}(\mathbf{x})$ for each p between 1 and r_v.

Definition 3.4. A dependence graph $G = \langle V, E, s, t, \lambda_\mathrm{v}, \lambda_\mathrm{e} \rangle$ is a *faithful representation* of a given program if for every non-input computation v and for every $\mathbf{x} \in D_v$, the value of v at $\mathbf{x}$ is equal to the value computed by iteration $\mathbf{x}$ of operation f_v at location l_v in the input program.

We can now formulate the following lemma.

Lemma 3.5. *The extraction procedure at the start of this section results in a dependence graph that is a faithful representation of the input program.*

Proof. To prove that the result is a dependence graph, it suffices to show that the three constraints of Definition 3.1 are satisfied.

(1) To see that this property holds, first note that the edges that emanate from a given computation with a given argument position are all of the same kind. If the argument is another operation or an input array, then there is a single

dependence edge leaving the computation for that argument and the property trivially holds. If the argument is a read from an array, then dependence analysis ensures that the property holds. Indeed, there is exactly one write iteration that last wrote to the memory location being read. Similarly, there is exactly one write iteration that last writes to the output array and therefore the property also holds for the edges leaving the output computation.

(2) This property holds because a statement iteration can only depend on a *previous* iteration in the execution order of the input program.

(3) This property holds because the program is static affine.

To complete the proof, we have to show that the graph is a faithful representation of the program. Clearly, a computation is faithful when its operands are input arrays or the result of computations that are faithful. Hence faithfulness follows by induction over the dataflow of the graph. □

3.5 Associative Operations

Associative operators can be nested differently in the two programs. In order not to have to worry about such possibly different nestings, we apply a normalizing preprocessing step that "flattens" associative operators in the dependence graph. This flattening step is similar to the approach of Shashidhar et al. [2005]. For example, a nesting of two binary associative operators introduces a ternary operator (possibly for only part of the domain of the outer node). Intuitively, an expression $+(a, +(b, c))$ with a nesting of two binary operators is replaced by the ternary expression $+(a, b, c)$.

When flattening computations, we may need to split the iteration domains as the nesting could in some cases only occur in part of the domain. The computation is then replaced by two computations, one where flattening is applied and one where it is not. In particular, we apply the following flattening procedure as long as we can find two distinct nodes v and w in a dependence graph such that

—$f_v = f_w = \oplus$, with $\oplus$ some associative operator,
—there is an edge $e \in E$ with $s(e) = v$ and $t(e) = w$,
—v and w do not originate from the same operation in the program, i.e., $l_v \neq l_w$.

The last condition means that we do not flatten recurrences. Before applying the procedure for the first time, the condition is equivalent to $v \neq w$, but after applying the procedure one or more times, we may have several nodes corresponding to the same operation in the program. If no pair can be found satisfying the above conditions, then we are done. Otherwise, part of our dependence graph looks like the fragment shown in Figure 4 and we want to combine v and w into a single computation. However, we can only do this for that part of the iteration domain of v that is actually mapped to w. We therefore replace the computation v by two computations v_1 and v_2 with iteration domains

$$D_{v_1} = D_v \setminus \operatorname{dom} M_{v,w}$$
$$D_{v_2} = D_v \cap \operatorname{dom} M_{v,w},$$

where v_2 corresponds to that part where flattening will be applied and v_1 corresponds to the other part. Now, v_2 can safely be replaced by the combination of the

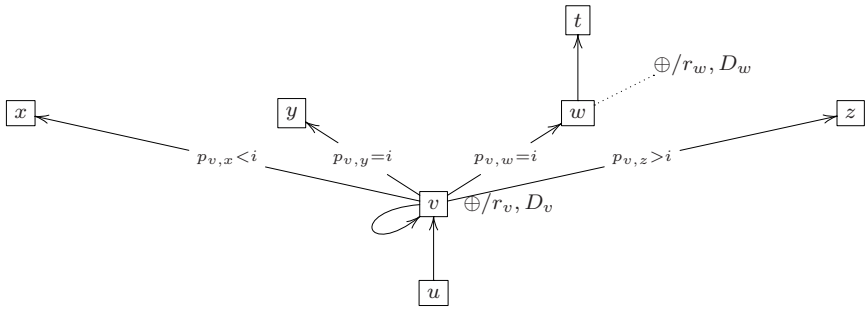


Fig. 4. Part of a dependence graph before applying the associativity transformation.

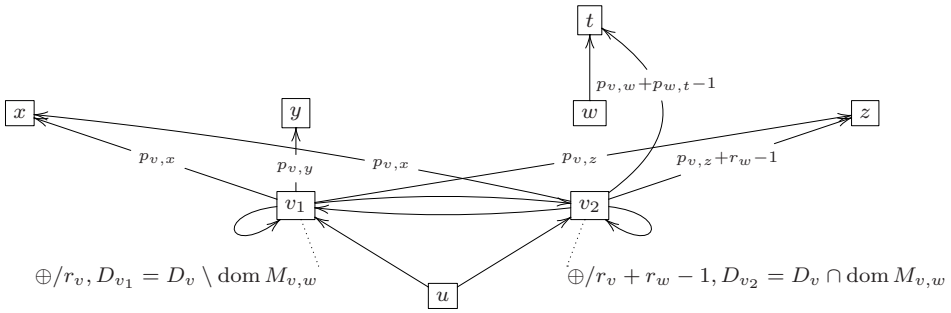


Fig. 5. The same dependence graph fragment of Figure 4, but after applying the associativity transformation.

original v and w into a single computation. The edges leaving v_2 are those leaving w pulled back over the edge between v and w. That is, for any edge (w,t), a new edge (v_2,t) is created, with

$$M_{(v_2,t)} = (M_{(w,t)} \circ M_{(v,w)}) \cap \left(D_{v_2} \to \mathbb{Z}^{d_t}\right).$$

Then the edges entering the original v are duplicated, i.e., every edge (u,v) is replaced by two edges (u,v_1) and (u,v_2), with

$$M_{(u,v_1)} = M_{(u,v)} \cap \left(\mathbb{Z}^{d_u} \to D_{v_1}\right)$$
$$M_{(u,v_2)} = M_{(u,v)} \cap \left(\mathbb{Z}^{d_u} \to D_{v_2}\right).$$

Note that u may be v_2 at this stage. Every edge (v,s) is also replaced by two edges (v_1,s) and (v_2,s), with

$$M_{(v_1,s)} = M_{(v,s)} \cap \left(D_{v_1} \to \mathbb{Z}^{d_s}\right)$$
$$M_{(v_2,s)} = M_{(v,s)} \cap \left(D_{v_2} \to \mathbb{Z}^{d_s}\right),$$

except when $p_{(v,s)} = p_{(v,w)}$. The edge (v,w) is dropped, while any other edge (v,s) with $p_{(v,s)} = p_{(v,w)}$ is replaced by an edge (v_1,s). If there is no such edge, then the node v_1 and the edges entering or leaving v_1 are not created. Note that the edge (v,v), if it exists, is duplicated twice (if v_1 is created).

The argument position of any edge leaving v_1 or entering v_1 or v_2 is the same as

```
b[0] = a[0];                    b[0] = a[N-1];
for (i = 1; i < N; ++i)         for (i = 1; i < N; ++i)
  b[i] = b[i-1] + a[i];           b[i] = a[N-1-i] + b[i-1];
out = b[N-1];                   out = b[N-1];
```

Fig. 6. Two programs for computing $\sum_{0 \leq i < N} \mathtt{a}[i]$.

the argument position of the edge from which it originates. For the edges leaving v_2, we have

$$\begin{cases} p_{(v_2,x)} = p_{(v,x)} & \text{if } p_{(v,x)} < p_{(v,w)} \\ p_{(v_2,t)} = p_{(v,w)} + p_{(w,t)} - 1 & \\ p_{(v2,z)} = p_{(v,z)} + r_w - 1 & \text{if } p_{(v,z)} > p_{(v,w)}. \end{cases}$$

Lemma 3.6. *The flattening procedure transforms a dependence graph that is a faithful representation of a given program into a dependence graph that is a faithful representation of the same program.*

Proof. It is sufficient to check that one iteration of the flattening procedure preserves the required properties. Constraint 1 holds essentially because D_{v_1} and D_{v_2} partition D_v. This ensures that the partitions in the input dependence graph are at most further subdivided in the output dependence graph. Constraint 2 holds because the dependence relations are simply partitioned and composed. These operations cannot introduce cycles. The iterations domains of the new nodes are subsets of those of the original nodes and therefore also constraint 3 is satisfied. Faithfulness is preserved because of associativity of the operation to which the procedure is applied. □

Note that the fact that we do not flatten recurrences means that we cannot detect the equivalence of the two programs in Figure 6 for computing $\sum_{0 \leq i < N} \mathtt{a}[i]$. Following the proposal of Shashidhar [2008] would result in a complete unrolling of the recurrences in these programs, leading to an infinite loop if N is a symbolic parameter.

3.6 Equivalence of Dependence Graphs

The concept of the equivalence of two dependence graphs is defined inductively and follows the intuitive definition of the equivalence of two programs at the start of Section 3. We first inductively define what it means for two iterations of two computations to be equivalent and then define equivalence of dependence graphs in terms of equivalence of their output computations.

Definition 3.7 *Equivalence of Computation Iterations.* An iteration $\mathbf{x}_1 \in D_{v_1}$ of a computation $v_1 \in V_1$ in a dependence graph G_1 is equivalent to an iteration $\mathbf{x}_2 \in D_{v_2}$ of a computation $v_2 \in V_2$ in a dependence graph G_2 if one of the following conditions holds

—v_1 and v_2 are input computations with $f_{v_1} = f_{v_2}$ and $\mathbf{x}_1 = \mathbf{x}_2$,
—$f_{v_1} = \mathrm{id}$ and iteration $M_{e_{G_1}(v_1,\mathbf{x}_1,1)}(\mathbf{x}_1)$ of $t(e_{G_1(v_1,\mathbf{x}_1,1)})$ is equivalent to iteration $\mathbf{x}_2$ of v_2,

—$f_{v_2} = \mathrm{id}$ and iteration $M_{e_{G_2(v_2,\mathbf{x}_2,1)}}(\mathbf{x}_2)$ of $t(e_{G_2(v_2,\mathbf{x}_2,1)})$ is equivalent to iteration $\mathbf{x}_1$ of v_1, or

—$(f_{v_1}, r_{v_1}) = (f_{v_2}, r_{v_2})$ and either

—for each $p \in [1, r_{v_1}]$, iteration $M_{e_{G_1}(v_1,\mathbf{x}_1,p)}(\mathbf{x}_1)$ of $t(e_{G_1(v_1,\mathbf{x}_1,p)})$ is equivalent to iteration $M_{e_{G_2}(v_2,\mathbf{x}_2,p)}(\mathbf{x}_2)$ of $t(e_{G_2(v_2,\mathbf{x}_2,p)})$, or

—$f_{v_1} = f_{v_2}$ is commutative and there exists a permutation π of the arguments such that for each $p \in [1, r_{v_1}]$, iteration $M_{e_{G_1}(v_1,\mathbf{x}_1,p)}(\mathbf{x}_1)$ of $t(e_{G_1(v_1,\mathbf{x}_1,p)})$ is equivalent to iteration $M_{e_{G_2}(v_2,\mathbf{x}_2,\pi(p))}(\mathbf{x}_2)$ of $t(e_{G_2(v_2,\mathbf{x}_2,\pi(p))})$,

Definition 3.8 *Equivalence of Dependence Graphs.* Two dependence graphs are equivalent if the iteration domains of their output computations are identical and if all iterations of these output computations are pairwise equivalent.

PROPOSITION 3.9. *If two dependence graphs constructed from input programs using the extraction procedure of Section 3.4 and the flattening procedure of Section 3.5 are equivalent, then the input programs are equivalent.*

PROOF. Two input computation iterations are only considered equivalent if they refer to the same element of the same input array. By assumption, the input arrays are equal and so equivalent input computation iterations have the same values. By induction all other equivalent computation iterations also have the same values. The values of the output computations correspond to the output arrays, which are therefore also equal. This proves that the two programs are equivalent. □

4. EQUIVALENCE CHECKING

In order to prove equivalence of two dependence graphs we basically follow Definition 3.7 and propagate from the output to the input what correspondences between computation iterations we should prove. These correspondences are maintained in an *equivalence tree* which is dynamically constructed as the equivalence proof proceeds. Once we hit computations with zero out-degree (either input computations or constant functions), we propagate back to the output what we have actually been able to prove. There are several reasons for this two-way propagation. Firstly, the discrepancy between what has to be proved and what is actually proved helps in debugging when the equivalence proof fails; secondly, as will become clear, propagating both ways will facilitate a better treatment of recurrences and commutativity.

The remainder of this section explains our equivalence checking algorithm shown in Algorithm 1. We first describe the equivalence tree. The main steps in our algorithm are then described as handling nodes in this equivalence tree. In particular, we first describe some easy cases (Section 4.2), followed by basic propagation (Section 4.3), propagation over commutative operations (Section 4.4) and propagation over recurrences (Section 4.5). After a more detailed comparison to Shashidhar et al. [2005] (Section 4.6) and a note on tabling (Section 4.7), we conclude with a proof of termination and soundness (Section 4.8).

4.1 The Equivalence Tree

The R_n^{want} relation or what has to be proved. The propagation from output to input constructs an equivalence tree. Each node n in the equivalence tree expresses

```
1 for (i=0; i<N; ++i)
2   t[N-1-i]=f(In[i]);
3 t[0]=g(t[0]);
4 for (i=0; i<N; ++i)
5   Out[i]=h(t[N-1-i]);
```
(a) Program A

```
1 for (i=0; i<N; ++i)
2   t[i]=f(In[i]);
3 t[0]=g(t[0]);
4 for (i=0; i<N; ++i)
5   Out[i]=h(t[i]);
```
(b) Program B

Fig. 7. Two programs that are almost equivalent.

a correspondence between a pair of computations, one from each dependence graph. The two computations involved are denoted $v_{n,1}$ and $v_{n,2}$. More specifically, each node n expresses that certain iterations of $v_{n,1}$ should or have been proved to be equivalent (as in Definition 3.7) to certain iterations of $v_{n,2}$. The desired correspondence between the iterations of both computations is captured by the R_n^{want} relation, a subset of the Cartesian product of the corresponding iteration domains, $R_n^{\text{want}} \subseteq D_{v_{n,1}} \leftrightarrow D_{v_{n,2}}$. This particular annotation of an equivalence node remains unaltered during the execution of the equivalence checker. The other annotations, described below, may change when the equivalence node is processed.

The R_n^{lost} relation or what has not been proved. The final result of a node is stored in the relation $R_n^{\text{lost}} \subseteq R_n^{\text{want}}$, which contains the pairs of computation iterations for which we have *not* been able to prove equivalence. The pairs of iterations that *have* been proved equivalent is then given by the difference of the two relations, i.e., $R_n^{\text{got}} = R_n^{\text{want}} \setminus R_n^{\text{lost}}$. Note that the exposition of Verdoolaege et al. [2009] keeps track of an R^{got} relation instead of an R^{lost} relation. We prefer R^{lost} here because in the ideal case, most R^{lost} relations are empty and therefore easier to manipulate. Furthermore, the use of R^{lost} is required for some extensions of the basic algorithm that are described by Verdoolaege et al. [2010].

Initialization of the equivalence tree. The initial equivalence tree consists of a single root node n_0 that models the equivalence to be proved between the output arrays of both programs. For this root, we have $R_{n_0}^{\text{want}} = \{u(\mathbf{i}) \leftrightarrow v(\mathbf{i}) \mid \mathbf{i} \in D\}$ with D the domain of the output computation (i.e., the domain of the output array). This relation expresses the intention to show that both arrays are identical. The computation of $R_{n_0}^{\text{lost}}$ typically requires the construction and handling of some child nodes. These children, along with their descendants, are visited in depth-first order. At the end of the equivalence checking procedure the $R_{n_0}^{\text{lost}}$ relation of the root node will be set. The proof is successful when $R_{n_0}^{\text{lost}} = \emptyset$.

Example 4.1. Figure 9 shows the equivalence tree for the pair of programs in Figure 7. The second program is obtained from the first by applying a data transformation on the temporary array `t`. However, the transformation has not been applied to the statement calling g. The two programs are therefore not completely equivalent. The dependence graph of the first program is shown in Figure 8. The dependence graph of the second program is very similar. The only difference is in the dependence relations. In particular, the second program has $\{\mathrm{v}_3^{\mathrm{B}}() \rightarrow \mathrm{v}_2^{\mathrm{B}}(0)\}$, $\{\mathrm{v}_4^{\mathrm{B}}(0) \rightarrow \mathrm{v}_3^{\mathrm{B}}()\}$ and $\{\mathrm{v}_4^{\mathrm{B}}(i) \rightarrow \mathrm{v}_2^{\mathrm{B}}(i) \mid i \geq 1\}$. Most of the equivalence tree will

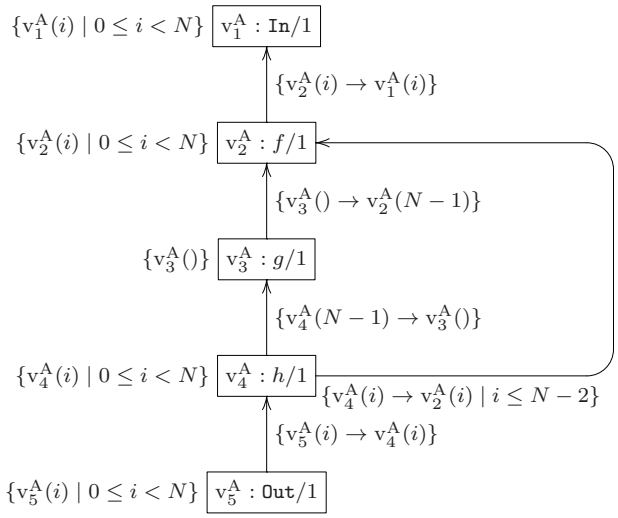


Fig. 8. Dependence graph G_A of the program in Figure 7(a).

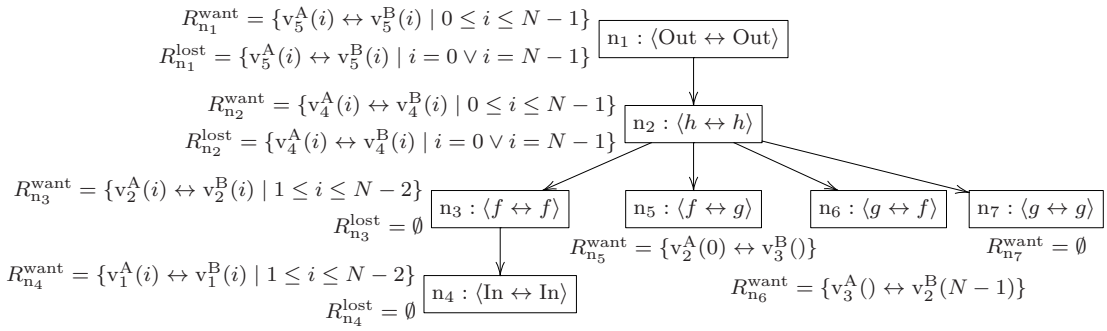


Fig. 9. Equivalence tree for the programs in Figure 7. Each node is represented by its name and the pair of operations involved in the computations. The computations themselves can be read off from the R^{want} annotations.

be explained in later examples. The root node n_1 of the tree expresses the desired equivalence of the output arrays. Since the output arrays are one-dimensional arrays in this example, we have $R^{want}_{n_1} = \{v_5^A(i) \leftrightarrow v_5^B(i) \mid 0 \le i \le N-1\}$.

4.2 Trivial Cases

We first discuss a couple of trivial cases where the R^{lost}_n relation can be determined without the creation of any child nodes. These are the first three cases in Algorithm 1. The remaining cases will be explained in later sections. If the current node n has an empty R^{want}_n, then there is nothing to prove and then also no correspondence that cannot be proved. The R^{lost}_n relation is therefore empty. The opposite conclusion holds if the computations of the node have different operations, neither of which is the copy operation id. According to Definition 3.7, iterations of

Algorithm 1: Try and handle node n in the equivalence tree

Input: dependence graphs $G_i = \langle V_i, E_i, s, t, \lambda_\mathrm{v}, \lambda_\mathrm{e} \rangle$, $i = 1, 2$; node n, with computations v_1 and v_2 and relation R_n^{want}
Output: relation R_n^{lost}
Modifies: node n
Throws: widen(a, R^{want}), with a any ancestor of n

```
1  Initialize R_n^need = ∅
2  try
3      if R_n^want = ∅ then                                              /* Empty */
4          Set R_n^lost = ∅
5      else if id ≠ f_{v1} ≠ f_{v2} ≠ id then                        /* No-match */
6          Set R_n^lost = R_n^want
7      else if both computations refer to the same input then          /* Input */
8          Set R_n^lost = R_n^want \ {u(i) ↔ v(i)}
9      else if there is a non-ancestor node s with s_{v_i} = n_{v_i} and
       R_n^want ∩ R_s^want ≠ ∅ then                                    /* Tabling */
10         Create a new child node n' with same computations and
           R_{n'}^want = R_n^want \ R_s^want
11         Try and handle n' (Algorithm 1)
12         Set R_n^lost = (R_s^lost ∩ R_n^want) ∪ R_{n'}^lost
13     else if there is an ancestor node a with a_{v_i} = n_{v_i} and n is not marked
       then
14         let a be the closest ancestor of n with a_{v_i} = n_{v_i} in
15             if a is a narrowing node or R_n^want ⊆ R_a^want then  /* Induction */
16                 Set R_n^lost = R_n^want \ R_a^want
17                 Add R_a^want ∩ R_n^want to R_a^need
18             else                                                  /* Widening */
                   /* a is not a narrowing node and R_n^want ⊈ R_a^want   */
19                 throw widen(a, R_n^want)
20     else
21         Handle propagation node n (Algorithm 2)
22 catch widen(n, R^want)                                            /* Widening */
23     Remove all children of n
24     Create a new child node n' with same computations and
       R_{n'}^want = R_n^want ∇ R^want
25     Mark n' as widening
26     Try and handle n' (Algorithm 1)
27     Set R_n^lost = R_{n'}^lost ∩ R_n^want                          /* Backpropagation */
```

such computations can never be equivalent and so we fail to prove anything that we wanted to prove, i.e., $R_n^{\text{lost}} = R_n^{\text{want}}$. Finally, if both computations of the node are input computations with identical "operations" (in this case, input array names), then, again according to Definition 3.7, we can only prove identical iterations (array indices) to be equivalent.

Example 4.2. In the equivalence tree in Figure 9, node n_7 has an empty R^{want}, while nodes n_5 and n_6 have conflicting operations. In node n_4, both computations refer to the same input array. Since $R_{\text{n}_4}^{\text{want}}$ is a subset of the identity mapping, we are able to prove it completely.

4.3 Basic Propagation

Propagation is the main step in our equivalence checking algorithm. Given an equivalence node n relating two computations with the same operation, this step will propagate the R_n^{want} relation over all pairs of edges with the same argument position emanating from the two computations in their respective dependence graphs. For each of these pairs of edges, a child node c is created with R_c^{want} the result of this propagation. These R_c^{want} relations are constructed in such a way that equivalence of all of them implies equivalence of the original R_n^{want}. However, we may not be able to prove equivalence for all of them completely and so we have to propagate back what we actually have (not) been able to prove. That is, once the R_c^{lost} relations of all the children have been computed, they are propagated back to compute R_n^{lost} of the original equivalence node. The propagation algorithm is shown in Algorithm 2. We first explain the standard propagation, starting at Line 6, and then the copy propagation, starting at Line 1. The induction condition in Line 10 will be examined in Section 4.5.

More precisely, let T_{n,p_1,p_2} be the set of all pairs of edges leaving computations $v_{n,1}$ and $v_{n,2}$ with argument positions p_1 and p_2, respectively, i.e.,

$$T_{n,p_1,p_2} = \{\, (e_1, e_2) \in E_{G_1} \times E_{G_2} \mid s(e_1) = v_{n,1}, s(e_2) = v_{n,2}, p_{e_1} = p_1, p_{e_2} = p_2 \,\},$$

and let T_n be the set of pairs of edges with the same argument positions, i.e.,

$$T_n = \bigcup_{1 \le p \le r_{v_{n,1}}} T_{n,p,p}, \tag{4}$$

then for each such pair of edges $(e_1, e_2) \in T_n$ a child node c_{e_1,e_2} is created. We are assuming here that $f_{v_{n,1}} = f_{v_{n,2}}$ is a non-commutative operator. For handling commutative operators, we refer to Section 4.4. The relation $R_{c_{e_1,e_2}}^{\text{want}}$ is obtained by propagating R_n^{want} over the dependence relations of both edges, i.e.,

$$R_{c_{e_1,e_2}}^{\text{want}} = (M_{e_1} \leftrightarrow M_{e_2})\, R_n^{\text{want}}, \tag{5}$$

with $M_{e_1} \leftrightarrow M_{e_2}$ the cross product of $M_{e_1} : D_{v_1} \to D_{u_1}$ and $M_{e_2} : D_{v_2} \to D_{u_2}$. Recall that the domains of the dependence relations M of all edges for a given argument position partition the domain of a node. Hence, for each argument position, R_n^{want} is partitioned by the domains of the combined dependence relations, i.e., each element of R_n^{want} is mapped to an element of the R_c^{want} of exactly one child c corresponding to this argument position.

Algorithm 2: Handle propagation node n

Input: $G_i = \langle V_i, E_i, s, t, \lambda_\mathrm{v}, \lambda_\mathrm{e} \rangle$, $i = 1, 2$; node n with computations v_1 and v_2
Output: relation R_n^{lost}
Modifies: node n
Throws: widen(a, R^{want}), with a equal to n or any ancestor of n
Requires: —at least one computation of n is not an input
—one of the following conditions
—n is narrowing or widening
—there is no ancestor with the same computations

1 **if** *at least one of the computations is a copy operation* **then** `/* Copy */`
2 **let** v_i *be the first copy operation* **in**
3 Add child nodes $\{c_j\}_j$ for each edge e_j emanating from v_i
4 Try and handle child nodes c_j (Algorithm 1)
5 Set

$$\begin{cases} R_n^{\text{lost}} = \bigcup_j (M_{e_j}^{-1} \leftrightarrow 1_{D_{v_2}}) R_{c_j}^{\text{lost}} & \text{if } i = 1 \\ R_n^{\text{lost}} = \bigcup_j (1_{D_{v_1}} \leftrightarrow M_{e_j}^{-1}) R_{c_j}^{\text{lost}} & \text{if } i = 2 \end{cases}$$

6 **else** `/* Propagation */`
7 Create child nodes $\{c_{(e_1,e_2)}\}_{(e_1,e_2)}$ for each pair of edges (e_1, e_2) emanating from v_1 and v_2
8 Try and handle child nodes $c_{(e_1,e_2)}$ (Algorithm 1)
9 Set R_n^{lost} according to formula (6) or (7) `/* Backpropagation */`
10 **if** $R_n^{\text{need}} \cap R_n^{\text{lost}} \neq \emptyset$ **then**
11 Remove all children of n
12 Reset $R_n^{\text{need}} = \emptyset$
13 **if** n *is not narrowing* **then** `/* First Narrowing */`
14 Create child node n' with same computations and $R_{n'}^{\text{want}} = R_n^{\text{want}} \setminus R_n^{\text{lost}}$
15 Mark n' as narrowing
16 Try and handle n' (Algorithm 1)
17 Reset $R_n^{\text{lost}} = R_n^{\text{lost}} \cup R_{n'}^{\text{lost}}$ `/* Finish Narrowing */`
18 **else** `/* Second Narrowing */`
19 Reset $R_n^{\text{lost}} = R_n^{\text{want}}$

If any pair of iterations in R_c^{want} in any of the children c could not be proved equivalent, then the corresponding pair of iterations of R_n^{want} cannot be proved equivalent either. The R_n^{lost} relation is therefore simply the union of all R_c^{lost} relations mapped back to the original space (Line 9 of Algorithm 2), i.e.,

$$R_n^{\text{lost}} = \left(\bigcup_{(e_1,e_2) \in T_n} \left(M_{e_1}^{-1} \leftrightarrow M_{e_2}^{-1} \right) R_{c_{e_1,e_2}}^{\text{lost}} \right) \cap R_n^{\text{want}}. \tag{6}$$

Note that the union is taken both over all argument positions and over all edges with a given argument position. The intersection with R_n^{want} may seem redundant,

and indeed it usually is. It is only needed in case this backpropagation is applied to a node created not during propagation, but during widening, see Section 4.5.

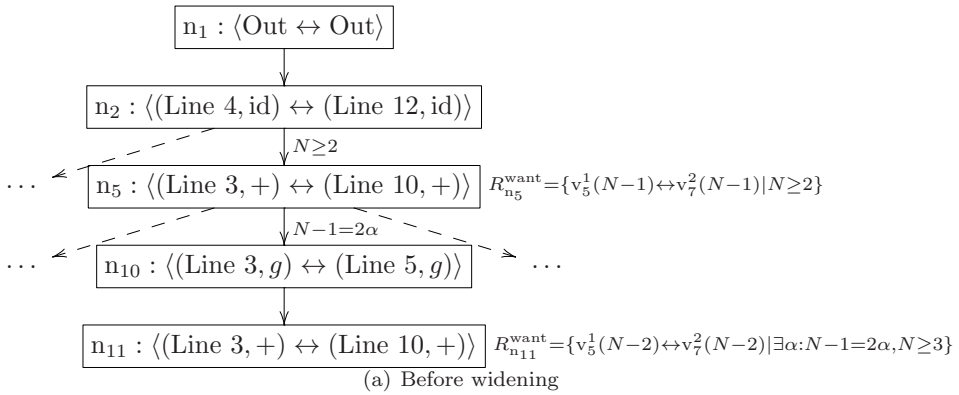
If one or both of the computations of the equivalence node n refers to a copy operation id, then propagation and backpropagation is only performed on the first (or only) copy computation (Line 5 of Algorithm 2). In particular, assume computation $v_{n,1}$ refers to a copy operation, then T_n only contains edges emanating from $v_{n,1}$ and M_{e_2} in (5) and (6) is replaced by an identity mapping. The reason for always picking the first computation to perform propagation on in case both refer to a copy operation, is that this consistency will be advantageous for induction and tabling, discussed below.

Example 4.3. Figure 10 shows a partial equivalence tree for the dependence graphs in Figure 2 and Figure 3, which correspond to the programs in Figure 1. The figure shows the tree in two stages of its lifetime, one before the widening step explained in Section 4.5 and one after. Most of this tree will be explained in later examples. The root node n_1 expresses a correspondence between output computation v_7^1 in dependence graph G_1 (Figure 2) and output computation v_9^2 in dependence graph G_2 (Figure 3), with $R^{\text{want}}_{\mathrm{n}_1} = \{\, \mathrm{v}_7^1() \leftrightarrow \mathrm{v}_9^2() \,\}$(not shown in the figure). The output computations have one outgoing edge, hence one child is created. Since the mappings on these edges are both identity mappings, we obtain the child $\langle(\text{Line } 4, \mathrm{id}/1) \leftrightarrow (\text{Line } 12, \mathrm{id}/1)\rangle$ (as in the figure, we do not identify the computation by its name, but by part of its label) with $R^{\text{want}}_{\mathrm{n}_2} = \{\mathrm{v}_6^1() \leftrightarrow \mathrm{v}_8^2()\}$, i.e., the copy operations in Line 4 of Program 1 and Line 12 of Program 2 should compute the same value. Each of these computations has two outgoing edges, but the constraints $N = 1$ and $N \geq 2$ are pairwise incompatible, so two of the four children have $R^{\text{want}} = \emptyset$ and are therefore not shown in the figure. Of the two other children, one is constrained with $N = 1$ and the other with $N \geq 2$. The correspondence of the latter, $\langle(\text{Line } 3, +/2) \leftrightarrow (\text{Line } 10, +/2)\rangle$, has $R^{\text{want}}_{\mathrm{n}_5} = \{\mathrm{v}_5^1(N-1) \leftrightarrow \mathrm{v}_7^2(N-1) \mid N \geq 2\}$ expressing that the addition on Line 3 of Program 1 and the addition on Line 10 of Program 2 should compute the same value in iteration $N - 1$ (when $N \geq 2$).

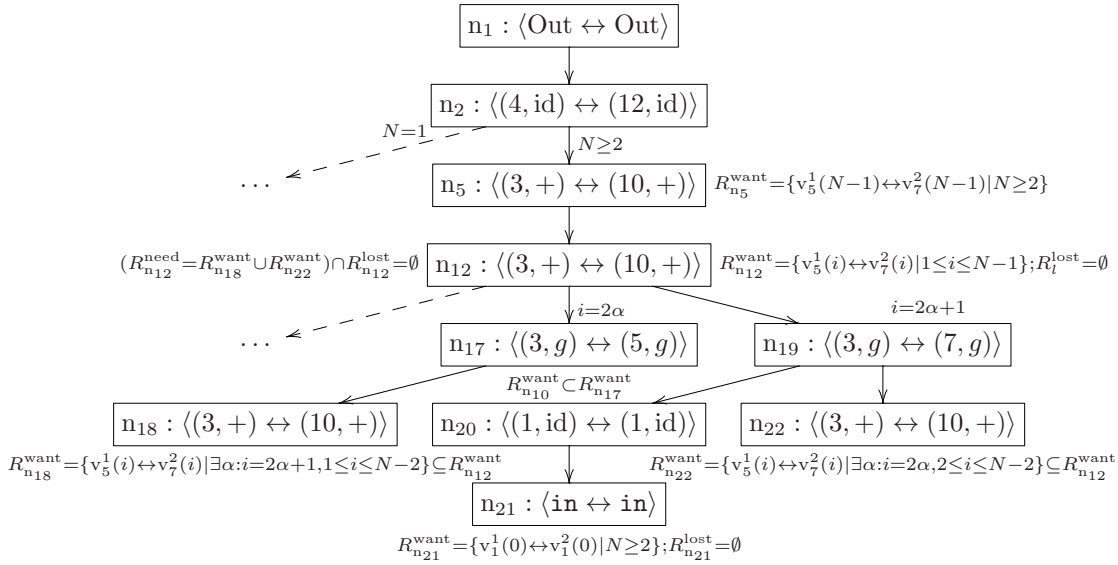
Example 4.4. Consider once more the equivalence tree in Figure 9. Since nodes n_5 and n_6 refer to conflicting operations, their R^{lost} relations are equal to their R^{want} relations. Propagating these relations back to node n_2 results in $R^{\text{lost}}_{\mathrm{n}_2} = \{\mathrm{v}_4^{\mathrm{A}}(i) \leftrightarrow \mathrm{v}_4^{\mathrm{B}}(i) \mid i = 0 \vee i = N-1\}$.

4.4 Propagation over Commutative Operations

Propagation over commutative operations requires a bit more work. According to Definition 3.7, a pair of iterations in the R^{want} relation of an equivalence node corresponding to commutative operations is considered equivalent if there is a permutation of the arguments such that the arguments in one dependence graph are equivalent to the permuted arguments in the other dependence graph. Since we do not know in advance which of the permutations should be selected for which of the elements in R^{want}, we have to consider all permutations for all elements. That is, we do not only create children for the pairs of edges in T_n (4), but instead for all pairs of edges in $\cup_{\pi \in \Pi} T_n^{\pi}$, with Π the set of all permutations of the $r_{v_{n,1}}$ arguments



(a) Before widening



(b) After widening

Fig. 10. Partial equivalence tree for the dependence graphs in Figure 2 and Figure 3.

and T^π_n the sets of pairs of edges with arguments permuted according to π, i.e.,

$$T^\pi_n = \bigcup_{1 \leq p \leq r_{v_{n,1}}} T_{n,p,\pi(p)}.$$

Obviously, we cannot just apply (6) during backpropagation because a given pair of iterations will in most cases only be proved equivalent for one particular permutation of the arguments. The pairs of iterations that we cannot prove equivalent are then those that cannot be proved equivalent using *any* permutation, i.e.,

$$R^{\text{lost}}_n = \bigcap_{\pi \in \Pi} \left(\bigcup_{(e_1, e_2) \in T^\pi_n} \left(M^{-1}_{e_1} \leftrightarrow M^{-1}_{e_2} \right) R^{\text{lost}}_{c_{e_1, e_2}} \right) \cap R^{\text{want}}_n. \tag{7}$$

This approach may seem to lead to an explosion in the number of cases that need to be considered, but usually the number of arguments of a commutative operator is fairly small and there is only one permutation that applies to all elements of R^{want}. The other permutations then quickly lead to a contradiction. Our running example is an exceptional case where different elements of R^{want} do require different permutations.

Example 4.5. In our running example, the $+/2$ computation of Program 1 (node v_5^1 of Figure 2) has one outgoing edge for each argument, while node v_7^2 in Figure 3 (the dependence graph of Program 2) has two outgoing edges for each, yielding 8 possible combinations. However, combinations leading to nodes where one computation has operator f and the other computation has operator g result in four children with $R^{\text{want}} = R^{\text{lost}} = \emptyset$. The other cases result in four children that contribute to the proof, namely $\langle (3, f/1) \leftrightarrow (4, f/1) \rangle$ and $\langle (3, g/1) \leftrightarrow (5, g/1) \rangle$ with constraint $i = 2\alpha$, and $\langle (3, f/1) \leftrightarrow (8, f/1) \rangle$ and $\langle (3, g/1) \leftrightarrow (7, g/1) \rangle$ with $i = 2\alpha + 1$. One of these is shown as a child of node n_5 in Figure 10. Once R^{lost} is available in each of the children, backpropagation can update R^{lost} in this node n_5.

4.5 Propagation in Presence of Recurrences

If there are any recurrences in both input programs, then a given pair of computations may (indirectly) depend on itself and this situation requires special care. Our proof procedure will only terminate if the depth of the equivalence tree is finite. This in turn means that any given pair of computations may only appear a finite number of times on any branch of the tree. In this section, we describe how we ensure that this property holds.

4.5.1 *Induction.* Let n be the current node under investigation. Assume there is an ancestor of n with the same pair of computations and let a be the closest such ancestor to n. In the most fortuitous case we have $R_n^{\text{want}} \subseteq R_a^{\text{want}}$, i.e., what we want to prove in n is a subset of what we were already trying to prove in a. In this case, there is no need to perform any propagation on node n as that would only lead to duplicate work. Instead, we optimistically assume that we will be able to prove the whole of R_a^{want} and so we mark node n as being completely proved, i.e., we set $R_n^{\text{lost}} = \emptyset$. Of course we later need to verify that this assumption was justified. We use a R^{need} relation, initialized to the empty relation, to collect all such assumptions. In the case at hand, R_a^{need} is extended with R_n^{want}. The general case of induction handling starts at Line 15 of Algorithm 1. Since $R_n^{\text{want}} \subseteq R_a^{\text{want}}$ in the case we are discussing here, we have $R_n^{\text{lost}} = R_n^{\text{want}} \setminus R_a^{\text{want}} = \emptyset$. Note that induction is only applied when n is not marked, where a node can be marked narrowing or widening. The condition ensures that n is a genuine new node resulting from a propagation step and not a copy of its parent after performing a widening or narrowing step as described further on in this section. Once we have computed R_a^{lost}, we need to check if we have actually proved all our assumptions. This check is performed in Line 10 of Algorithm 2. What happens when we have not been able to prove all our assumptions is explained later in this section. Note that due to the second constraint of Definition 3.1 there is no risk of circular reasoning ("unfounded sets") when applying induction. No individual iteration can (indirectly) depend on itself, hence no pair of individual iterations can depend on itself.

4.5.2 *Widening.* Now let us consider the case where $R_n^{\text{want}} \not\subseteq R_a^{\text{want}}$. We cannot simply perform propagation on node n as that could lead to an infinite sequence of equivalence nodes with the same pair of computations. Note that for any fixed value of the parameters, the iteration domains are bounded, but the values of the parameters are typically not and so we may indeed end up with ever growing R^{want} relations. But even if the values of the parameters are bounded, then the sequence of equivalence nodes may still be very long if we were to continue applying propagation, as we would effectively be unrolling the loops containing the recurrences.

Instead, we draw inspiration from the widening/narrowing technique of abstract interpretation [Cousot and Cousot 1992] and apply a *widening* operator ∇. Such a widening operator turns a possibly infinite ascending chain, e.g., taking the union with R_n^{want} in each descendant with the same pair of computations, into an eventually stationary chain. As our widening operator, we will essentially use the integer affine hull, but restricted to the respective iteration domains. That is, we intersect the integer affine hull with $D_{v_{n,1}} \leftrightarrow D_{v_{n,2}}$ and use the resulting relation as $R_{n'}^{\text{want}}$ of a newly created child n' of node a, replacing the entire tree that was originally rooted at a. Since the need for widening is *detected* while handling the descendant, whereas it needs to be *handled* at the level of the ancestor, we use exception handling to return directly to the ancestor. In particular, the descendant throws an exception in Line 19 of Algorithm 1, which is caught by the try-catch block of the ancestor in Line 22. Backpropagation on the child n' is performed in essentially the same way as before. Since the parent-child relation here is not based on any edges in the dependence graphs, the mappings M_{e_1} and M_{e_2} in (6) are taken to be identity mappings. In other words, (6) simplifies to

$$R_a^{\text{lost}} = R_{n'}^{\text{lost}} \cap R_a^{\text{want}}.$$

To see that taking the integer affine hull is indeed a widening operator, note that the first time it is applied it sets $R_{n'}^{\text{want}}$ to the intersection of $D_1 \leftrightarrow D_2$ with some affine subspace. Any additional widening step is only performed when R_d^{want} of a descendant d of this n' includes an element not in $R_{n'}^{\text{want}}$ (but still in $D_1 \leftrightarrow D_2$) and the widening operator will then increase the dimension of the affine subspace. So, after a finite number of widening steps, $R^{\text{want}} = D_1 \leftrightarrow D_2$, ensuring termination. If any further equivalence node d' with the same pair of computations is encountered, then we will have $R_{d'}^{\text{want}} \subseteq R_{a'}^{\text{want}}$, with $R_{a'}^{\text{want}}$ the result of the last widening step. The affine hull not only ensures termination of the widening sequence, it is also a reasonable heuristic as an affine program will only remain affine if it is transformed using a (piecewise) affine transformation.

4.5.3 *Narrowing.* Finally, we need to consider what happens when it turns out we have been overly optimistic in our induction hypothesis, i.e., when $R_n^{\text{need}} \cap R_n^{\text{lost}} \neq \emptyset$ (Line 10 of Algorithm 2). In this case, the performed induction is not founded by what we actually can prove. This means that R_n^{want}, the current hypothesis, is an over-approximation of the correct induction hypothesis, or at least of the induction hypothesis that we are able to prove. In a second phase, we can still attempt to prove some part of R_n^{want}. However, as in the first phase, we need to be careful not to end up in a possibly infinite sequence of (now) successive subsets of R_n^{want}. Similar to abstract interpretation, we therefore perform a (finite) number

of *narrowing* steps that decrease R^{want} until it can be proved, which is definitely the case when it becomes empty. Our narrowing operator is fairly simply. The first time it is applied (Line 13), we replace the tree rooted at n by a new child node n' and set $R^{\text{want}}_{n'} = R^{\text{want}}_n \setminus R^{\text{lost}}_n$. Then, in any descendant d with the same pair of computations, we unconditionally apply induction, setting $R^{\text{lost}}_d = R^{\text{want}}_d \setminus R^{\text{want}}_{n'}$ (see Line 15 of Algorithm 1). In particular, we do not allow any more widening steps on this pair of computations. Note that nodes descending from n' can still be the subject of widening steps for a different pair of computations (within the same or another recurrence). When propagation on a narrowing node n has been performed and the resulting R^{lost}_n still intersects with its R^{need}_n, then we perform a second and final narrowing step and create a child n' with the empty set as $R^{\text{want}}_{n'}$. Actually, we need not create the child, we can simply set $R^{\text{lost}}_n = R^{\text{want}}_n$ (Line 18 of Algorithm 2).

4.5.4 *Example.* We illustrate the handling of recurrences with an example.

Example 4.6. After applying propagation to node n_{10} of the equivalence tree in Figure 10(a), we end up in node n_{11} which refers to the same pair of computations as node n_5. We have $R^{\text{want}}_{\text{n}_{11}} = \{\, \text{v}^1_5(N{-}2) \leftrightarrow \text{v}^2_7(N{-}2) \mid \exists \alpha : N{-}1 = 2\alpha, N \geq 3 \,\}$, while $R^{\text{want}}_{\text{n}_5} = \{\, \text{v}^1_5(N{-}1) \leftrightarrow \text{v}^2_7(N{-}1) \mid N \geq 2 \,\}$, i.e., $R^{\text{want}}_{\text{n}_{11}} \not\subseteq R^{\text{want}}_{\text{n}_5}$. We therefore apply widening and obtain $R^{\text{want}}_{\text{n}_{12}} = R^{\text{want}}_{\text{n}_5} \nabla R^{\text{want}}_{\text{n}_{11}} = \{\, \text{v}^1_5(i) \leftrightarrow \text{v}^2_7(i) \mid 1 \leq i \leq N-1 \,\}$. The new node n_{12} replaces the entire tree rooted at n_5, as shown in Figure 10(b). We now have $R^{\text{want}}_{\text{n}_{12}} \not\subseteq R^{\text{want}}_{\text{n}_5}$, but n_{12} is marked (as widening) and therefore no widening is applied to this node (Line 13 of Algorithm 1). Instead, we perform propagation on n_{12} in much the same way as we did on n_5 in Example 4.5. Another propagation step on the resulting node n_{17} yields node n_{18}, with the same pair of computations as node n_{12}. Since the condition $R^{\text{want}}_{\text{n}_{18}} \subseteq R^{\text{want}}_{\text{n}_{12}}$ in Line 15 holds, we apply induction and set $R^{\text{lost}}_{\text{n}_{18}}$ to $\emptyset$ and update $R^{\text{need}}_{\text{n}_{12}}$. A very similar story happens in node n_{22}. After propagating the results back to node n_{12}, we see that $R^{\text{lost}}_{\text{n}_{12}} = \emptyset$, meaning that all the induction hypotheses have been validated and there is no need for a narrowing phase.

4.6 Comparison to Shashidhar et al. [2005]

Like those of Barthou et al. [2002] and Alias and Barthou [2003], the approach of Shashidhar et al. [2005] and Shashidhar [2008], essentially only propagates information from output to input, whereas we propagate information in both directions.

Our approach for handling commutative operations is also different from the solution proposed by Shashidhar et al. [2005] (without implementation). They propose to consider all permutations of the arguments of the second computation separately and to use a look ahead mechanism to figure out which permutation is correct. However, this proposal would not work on our running example as neither of the two possible permutations is correct on its own. One only holds for the even values of i ($i = 2\alpha$) and the other only for odd values of i ($i = 2\alpha + 1$); the proof attempt of Shashidhar et al. [2005] gets stuck.

Our recurrence handling differs substantially from that of Shashidhar et al. [2005]. The program model used in that work makes it non trivial to find the ancestor/descendant pair over which both programs have performed the same computation. They need an unfolding operation to identify the pair, then they compute the across

dependence relation that corresponds to the computation performed between ancestor and descendant and use that relation in a complex operation that involves the calculation of the transitive closure (implemented in the `Omega` library) that yields the equivalences to be proved for the edges leaving the recurrence. This computation requires the recurrences to be uniform while our method can also handle non uniform recurrences. Furthermore, their representation of proof obligations only allows an element of an output array to depend on a single element of another array along any path in the program. In particular, if a program contains a loop with body `A[i] = A[i-1] + B[i]`, then they are unable to express that `A[N]` depends on `B[i]` for *all* iterations `i` of the loop. After stepping over the recurrence, they will therefore ignore all but one of these elements `B[i]`.

4.7 Tabling

It is quite common for a program to reuse some of the data it has computed. While checking for the equivalence of two such programs we may then end up in an equivalence node that we have already (partially) proved equivalent. In such a case, we want to avoid repeating the proof and instead also reuse the equivalence proof. In particular, assume that the current node n has the same pair of computations as some non-ancestor node s and that their R^{want} relations overlap. We may then simply copy the results obtained in s for the overlapping part. A new child n' is created for handling the possibly empty remainder of R_n^{want}, i.e., $R_n^{\text{want}} \setminus R_s^{\text{want}}$. In Algorithm 1, tabling starts at Line 9.

Note that the results in s may depend on some induction hypotheses. In order to ensure the validity of the copied results, we need to impose that the equivalence tree is traversed in depth-first order. Let a be the closest common ancestor of n and s. If the conclusions in s depend on induction steps performed with respect to a node on the path between s and a, then these induction steps have already been validated. Otherwise, the tree rooted at that node, including s, would have been removed already. If there is a dependence on node a or any of its ancestors, then these assumptions will still be validated. If they turn out to be unsubstantiated, then the corresponding tree will again be removed and this tree includes both s and n. In no case is an assumption then allowed to escape validation. Note that the table used for tabling can also be used to detect recurrences.

4.8 Termination and soundness

We will now proceed to prove termination and soundness of our equivalence checking procedure shown in Algorithm 1. Our procedure is not complete since the problem of checking the equivalence of static affine programs is undecidable [Barthou et al. 2002]. For the purpose of the proofs, we will consider steps that cut away parts of the equivalence tree as not actually removing any nodes, but just marking them as having been deleted. For example, when we say the equivalence tree is finite, then this means that the total number of nodes ever visited is finite and not just the number of nodes that are left over at the end of the procedure. During our proofs we will also use the convention that a relation (R^{want}, R^{lost}, R^{need} or any boolean combination of these relations) evaluates to true iff each pair of computation iterations in the relation computes the same value.

LEMMA 4.7. *Algorithm 1 terminates. That is, given two finite dependence graphs G_1 and G_2, the corresponding equivalence tree is finite.*

PROOF. There are four steps that create new children, at most $|E_1||E_2|$ by Forward Propagation (Line 3 and Line 7 of Algorithm 2), with E_i the edges of G_i, one by Widening (Line 24 of Algorithm 1), one by Narrowing (Line 14 of Algorithm 2) and one by Tabling (Line 10 of Algorithm 1). Propagation and tabling create the initial children of a node, while widening and narrowing create at most one additional child for any given node and are then never applied on the same node again. The number of children of any node is therefore bounded by $|E_1||E_2| + 1$.

Now let us consider the depth of the tree and let us first ignore tabling. If a node n has the same pair of computations as an ancestor of n, then Propagation is only applied if n is narrowing or widening (Line 13 of Algorithm 1). These narrowing and widening nodes appear as direct children of other nodes with the same pair of computations. Any pair of computations can therefore appear at most $1+d_1+d_2+1+1$ times in any branch of the equivalence tree. The first is the initial occurrence of the pair, immediately followed by at most $d_1 + d_2$ widening nodes, with d_i the dimension of the iteration domain of computation v_i, possibly followed by one narrowing node and one leaf node. Since there is only a finite number of computation pairs, the depth of the equivalence tree is finite. Tabling can increase the number of nodes with the same pair of computations on a branch, but only by a finite amount. The equivalence tree is therefore finite. □

The following lemma expresses that equivalences between computations that the algorithm derives in the nodes of the equivalence tree ($R_n^{\text{want}} \setminus R_n^{\text{lost}}$) indeed hold under the assumption that the equivalences expressed in the R^{need} relation of the ancestors of that node hold.

LEMMA 4.8. *For any undeleted and handled node n, any pair of iterations in the relation $R_n^{\text{want}} \setminus R_n^{\text{lost}}$ computes the same value if every pair of iterations in R_a^{need} for any ancestor a of n computes the same value, i.e.,*

$$\left(\bigwedge_{a \in \mathcal{A}(n)} R_a^{\text{need}} \right) \Rightarrow R_n^{\text{want}} \setminus R_n^{\text{lost}}, \tag{8}$$

with $\mathcal{A}(n)$ the set of ancestors of n.

PROOF. First note that this property is stable, in the sense that once proved at a certain point in the execution, it will remain true throughout the rest of the execution because a node n is never changed after it has been handled and further steps can only add elements to the R_a^{need} and never remove elements from these relations. The only exception is Line 12 of Algorithm 2, but this happens immediately after any node that may depend on R_n^{need} has been deleted. We will proof this lemma by induction on the order in which handling of nodes is finished, i.e., a depth-first postordering of the equivalence tree. We consider nine cases.

—Empty (Line 3 of Algorithm 1)
$R_n^{\text{want}} \setminus R_n^{\text{lost}} = \emptyset$, so (8) trivially holds.

—Input (Line 7 of Algorithm 1)
$R_n^{\text{want}} \setminus R_n^{\text{lost}} = R_n^{\text{want}} \cap \{ u(\mathbf{i}) \leftrightarrow v(\mathbf{i}) \}$ holds by Definition 3.7.

—Tabling (Line 9 of Algorithm 1)
The induction hypothesis holds for both the non-ancestor s and for the child n', hence

$$\left(\bigwedge_{t\in\mathcal{A}(s)} R_t^{\text{need}}\right) \Rightarrow R_s^{\text{want}} \setminus R_s^{\text{lost}} \tag{9}$$

and

$$\left(\bigwedge_{t\in\mathcal{A}(n')} R_t^{\text{need}}\right) \Rightarrow R_{n'}^{\text{want}} \setminus R_{n'}^{\text{lost}}. \tag{10}$$

Because we traverse the tree in depth-first order, any ancestor t of s that is not also an ancestor of n, i.e., any node on the path (s, a), with a the closest common ancestor of s and n, has been handled and therefore has $R_t^{\text{need}} \cap R_t^{\text{lost}} = \emptyset$, i.e., $R_t^{\text{need}} \subseteq R_t^{\text{want}} \setminus R_t^{\text{lost}}$. (Either this condition already holds in Line 10 of Algorithm 2, or it is made to hold in Line 12.) Let t_1 be the parent of s, then by applying the induction hypothesis to t_1, $R_{t_1}^{\text{need}} \subseteq R_{t_1}^{\text{want}} \setminus R_{t_1}^{\text{lost}}$ on the lefthand side of (9) can be replaced by the conjunction of R_t^{need} for all ancestors t of t_1. Since $\mathcal{A}(t_1) \subset \mathcal{A}(s)$, this means we can simply drop $R_{t_1}^{\text{need}}$. This process can be continued for all nodes on the path between s and a, resulting in

$$\left(\bigwedge_{t\in\mathcal{A}(a)} R_t^{\text{need}}\right) \Rightarrow R_s^{\text{want}} \setminus R_s^{\text{lost}}.$$

In this formula, a can safely be replaced by n, because this replacement only adds extra conditions. Since n' has the same pair of computations as n, n will never be used as a closest ancestor during induction (Line 15 of Algorithm 1), so we have $R_n^{\text{need}} = \emptyset$ and n may be dropped from the antecedent in (10). Combining these results, we have

$$\left(\bigwedge_{a\in\mathcal{A}(n)} R_a^{\text{need}}\right) \Rightarrow (R_s^{\text{want}} \setminus R_s^{\text{lost}}) \cup (R_{n'}^{\text{want}} \setminus R_{n'}^{\text{lost}}) = R_n^{\text{want}} \setminus R_n^{\text{lost}},$$

as required.

—Induction (Line 15 of Algorithm 1)
We have $R_n^{\text{want}} \setminus R_n^{\text{lost}} = R_n^{\text{want}} \cap R_a^{\text{want}} \subset R_a^{\text{need}}$ and so the lemma trivially holds.

—Widening (Line 27 of Algorithm 1)
Since n' has the same pair of computations as n, n will never be used as a closest ancestor during induction (Line 15 of Algorithm 1), so we have $R_n^{\text{need}} = \emptyset$. As in the case of Tabling, we may therefore drop n from the antecedent of the results of the lemma on node n'.

—First Narrowing (Line 13 of Algorithm 2)
By induction, the property holds for n'. Again n may be dropped from the antecedent and we obtain

$$\left(\bigwedge_{a\in\mathcal{A}(n)} R_a^{\text{need}}\right) \Rightarrow R_{n'}^{\text{want}} \setminus R_{n'}^{\text{lost}} = (R_n^{\text{want}} \setminus R_n^{\text{lost}}) \setminus R_{n'}^{\text{lost}} = R_n^{\text{want}} \setminus (R_n^{\text{lost}} \cup R_{n'}^{\text{lost}}),$$

with R_n^{lost} the value before the update and $R_n^{\text{lost}} \cup R_{n'}^{\text{lost}}$ the updated value of R_n^{lost}.

—Second Narrowing (Line 18 of Algorithm 2)
In this case, $R_n^{\text{want}} \setminus R_n^{\text{lost}} = \emptyset$ and so the lemma trivially holds.

—Propagation (Line 6 of Algorithm 2)
If the condition in Line 10 holds, then we apply one of the narrowing steps discussed before. Otherwise, we have $R_n^{\text{need}} \cap R_n^{\text{lost}} = \emptyset$. By induction, we also have that (8) holds in all the children of n. Using Definition 3.7 and (6) or (7) as appropriate, we conclude

$$\left(\bigwedge_{a \in \{n\} \cup \mathcal{A}(n)} R_a^{\text{need}} \right) \Rightarrow R_n^{\text{want}} \setminus R_n^{\text{lost}}.$$

To eliminate the extra R_n^{need} in the antecedent, we appeal to a second induction on essentially the execution order. By the second condition of Definition 3.1, no element of an iteration domain depends on itself and therefore also no pair of iterations can depend on itself. Fix a value for the parameters and take any fixed element $(\mathbf{i}, \mathbf{i}')$ from $R_n^{\text{need}} \subseteq R_n^{\text{want}} \setminus R_n^{\text{lost}}$ for that value of the parameters. Repeat the proof for this particular element without performing any widening. Since we have fixed the value of the parameters, this is a finite process. If we can prove equivalence without performing induction on this pair of computations, then the $(\mathbf{i}, \mathbf{i}')$ can be dropped from the antecedent of the above formula. Otherwise, the resulting R^{need} only contains elements that are obtained through successive applications of the dependence relations and therefore does not contain the element $(\mathbf{i}, \mathbf{i}')$. This process can be repeated on this resulting R^{need} and no element will ever reappear in any of the successive R^{need} relations. Since the iteration domains are finite for fixed values of the parameters (third condition of Definition 3.1), this process terminates and we can drop $(\mathbf{i}, \mathbf{i}')$ from the antecedent also in this case. By repeating for all values in R_n^{need} and all values of the parameters, we can drop the whole R_n^{need} from the antecedent.

—Copy Propagation (Line 1 of Algorithm 2)
This case is proved in the same way as Propagation and thereby completes the proof.

□

THEOREM 4.9. *If $R_{n_0}^{\text{lost}} = \emptyset$, with n_0 the initial node in the equivalence tree, then the two dependence graphs are equivalent.*

PROOF. After termination of the algorithm (Lemma 4.7), the initial node n_0 has been handled. Since n_0 is the root node, it has no ancestors, i.e., $\mathcal{A}(n_0) = \emptyset$. By Lemma 4.8, this in turn means that

$$R_{n_0}^{\text{want}} \setminus R_{n_0}^{\text{lost}} = R_{n_0}^{\text{want}}$$

holds. □

5. IMPLEMENTATION DETAILS

The proof procedure of Algorithm 1 has been implemented as part of our C++ `isa` (`http://www.kotnet.org/~skimo/loop/isa-0.12.tar.bz2`) prototype tool set. The tool set contains a polyhedral model extractor [Verdoolaege and Grosser 2012] from C based on LLVM/clang and an exact dependence analysis tool.

As to the manipulation of integer sets and relations, it is important that any library used to perform these manipulations should support parameters and existentially quantified variables. Previous approaches to equivalence checking of static affine programs used the `Omega` library [Kelly et al. 1996] for this purpose, as it provides the required functionality. Fu et al. [2006] later also identified this library as being the only library available at the time that was suited for equivalence checking. However, the library has been unmaintained for many years and suffers from various unimplemented corner cases, rendering it unreliable. Only very recently, some (possibly all) of these issues have been resolved in the `Omega+` library [Chen 2009].

Instead, we use our own thread-safe C library called `isl` [Verdoolaege 2010], which uses `GMP` to perform all its integer manipulations in exact integer arithmetic and is available from `http://freecode.com/projects/isl/`. The interface of the library is somewhat reminiscent of that of `Omega`, but the underlying implementation is completely different. The library provides several advanced operations such as the integer affine hull needed for our widening operation and dependence analysis [Feautrier 1991]. The implementation of the integer affine hull is based on finding integer points using an integer linear feasibility solver based on generalized basis reduction [Cook et al. 1991] and then computing the affine hull [Karr 1976]. An efficient integer linear feasibility solver is also crucial for our equivalence checking procedure as it needs to check at many stages whether an integer set or relation is empty.

Each set/relation is represented by a union of "basic sets", each of which is defined by a conjunction of linear inequalities. If an R^{want} relation is a union of basic sets, a node is created for each of its basic sets. All nodes with the same pair of computations are kept in a list accessible through a hash table keyed on the given pair, which is used both for tabling and detecting recurrences. The implemented algorithm differs slightly from the exposition above. In particular, we never remove any node from the equivalence tree or restart a proof, but instead extend the tree while keeping track of all the induction hypotheses that have been made. The implementation also contains various other optimizations to avoid redundant computations.

In order to be able to perform some experimental comparison with related work, we have also implemented the technique of Barthou et al. [2002]. This technique essentially constructs an automaton with states corresponding to pairs of nodes from two dependence graphs and transitions between two states if equality of one requires equality of the other. The initial state corresponds to equal outputs and leaf states may be failures states or success states (i.e., equal inputs). The regular expression for all paths from the initial state to each leaf states are converted into an accessibility relation by replacing concatenation by composition, branches by union and cycles by transitive closures. In order to have equivalence, the ac-

```
for (k=0; k<256; k++)
  t1[k]=A[2*k]+f(B[k+1]);
for (k=10; k<138; k++)
  t2[k]=B[k-8];
for (k=10; k<266; k++){
  if(k >= 138)
    t2[k]=B[k-8];
  t3[k-10]=f(A[2*k-19])+t2[k];
}
for (k=255; k>=0; k--)
  C[3*k]=t1[k]+t3[k];
```

```
for (k=0; k<256; k++) {
  t4[k]=f(A[2*k+1])+A[2*k];
  t5[k]=B[k+2]+t4[k];
  C[3*k]=f(B[k+1])+t5[k];
}
```

Fig. 11. A pair of equivalent programs from Shashidhar et al. [2005].

cessibility relations should be empty for failure states and covered by equality of input array indices for success states. In our implementation, we use `VAUCANSON` [Lombardy et al. 2004] to compute regular expressions and `isl` to perform all relation manipulations. It should be noted that an implementation based on `Omega` would not be sound as `Omega` computes underapproximations of transitive closures [Kelly et al. 1996, Section 6.4], meaning that some equivalence conditions may get lost. The transitive closure operation of `isl`, on the other hand, computes overapproximations, possibly resulting in spurious conditions, but never dropping any required conditions. Our implementation has also been shown to be more accurate than `Omega`'s on this type of inputs [Verdoolaege et al. 2011].

Besides the approach of Barthou et al. [2002] itself, we have also implemented a simplified variation. In this variation, we do not compute a regular expression for all paths in the automaton, but instead compute the accessibility relation directly as the transitive closure of the pairs of dependence relations.

The tool of Shashidhar [2008] was developed outside of our university and we were unable to obtain a working copy of the tool for performing a comparative experiment. Given the fundamental problems of the technique of Shashidhar et al. [2005] highlighted in Section 4.6, we did not feel it worthwhile to also reimplement this technique.

6. EXAMPLES AND EXPERIMENTS

Figure 11 reproduces the motivating example of Shashidhar et al. [2005]. The programs are only equivalent if the `+` operator is treated as both associative and commutative. Figure 12 shows a pair of programs with output arrays `out` that are not equivalent (unless $N = 0$) because `b[0]` is assigned a different value in the two programs. However, Shashidhar [2008] will not detect this error (except for $N = 1$). His equivalence checker will notice that the values of `A[N]` need to be equal, which in turn (for values of N large enough) means that both the values of `b[N-1]` and those of `A[N-1]` need to be equal. The first pair is easily seen to be equal. For the second pair, the equivalence checker will detect a recurrence and move straight to the base case, i.e., that the values of `A[0]` should be equal, completely ignoring the fact the values of `b[i-1]` should also be equal for all values of `i` along the recurrence.

```
if (N >= 0) {
  a[0] = 5;
  b[0] = 3;
  for (i = 1; i <= N; ++i) {
    a[i] = b[i-1] + a[i-1];
    b[i] = in[i-1];
  }
  out = a[N];
}
```

```
if (N >= 0) {
  a[0] = 5;
  b[0] = 2;
  for (i = 1; i <= N; ++i) {
    a[i] = b[i-1] + a[i-1];
    b[i] = in[i-1];
  }
  out = a[N];
}
```

Fig. 12. A pair of non-equivalent programs.

```
A[0] = in;
for (i = 1; i <= N; ++i)
    A[i] = f(g(A[i/2]));
out = g(A[N]);
```

```
A[0] = g(in);
for (i = 1; i <= N; ++i)
    A[i] = g(f(A[i/2]));
out = A[N];
```

Fig. 13. A pair of equivalent programs with a non-uniform recurrence.

```
sum = 0;
for (i = 0; i < N; ++i)
    for (j = 0; j < i; ++j)
        sum = sum + a[i][j];
out = sum;
```

Fig. 14. A program with a piecewise uniform recurrence.

Figure 13 shows a pair of programs that contain a non-uniform recurrence. The approaches of Barthou et al. [2002] and Shashidhar et al. [2005] cannot handle such non-uniform recurrences. Note that the `A` arrays of both programs store different values, so our tool cannot prove the equivalence of `A`. It can, however, prove the equivalence of the `out` arrays. Figure 14 shows a program with a piecewise uniform dependence. That is, the first iteration of the inner loop depends on the previous iteration of the outer loop, while the other iterations of the inner loop depend on the previous iteration of this inner loop. Computing the transitive closure of this dependence relation itself is fairly easy. However, the approaches of Barthou et al. [2002] and Shashidhar et al. [2005] operate on *pairs* of dependence relations. This means that the transitive closure should express that the same number of steps are taken over both dependences. For this example, encoding this property is impossible using only affine constraints. These approaches are therefore unable to prove that the given program is equal to itself.

Figure 15 shows two versions of a program computing the function $\sum_{i=0}^{n} i$. In the second version, the `i`-loop has been partially unrolled by a factor of two. Rewriting the loops in terms of recursive calls results in two programs that are essentially the same as those in the first example of an equivalence that the rules of Godlin and Strichman [2008] cannot prove [Godlin and Strichman 2008, Sec-

```
out = 0;
if (n >= 0) {
   for (i=1; i<=n; ++i)
      out = out + i;
}
```

(a) Original program

```
out = 0;
if (n >= 0) {
   for (i=1; i<=n/2; ++i) {
      out = out + (2*i-1);
      out = out + 2*i;
   }
   if (n % 2 == 1)
      out = out + n;
}
```

(b) After partial loop unrolling

Fig. 15. Partial loop unrolling.

tion 6]. Our proof procedure has no problem proving the equivalence of these two programs. Incidentally, both programs are also equivalent to the program "`out = n >= 0 ? n * (n+1) / 2 : 0;`", but we currently do not handle this transformation. In this particular case, weighted parametric counting [Verdoolaege and Bruynooghe 2008] could be used to fairly easily prove this equivalence as well.

Table I shows some experimental results obtained using our tool. For each pair of input programs, we list the number of statements in the programs, the number of computations in the corresponding dependence graphs, the maximal dimension of the iteration domains, the time taken by both the dependence analysis and the equivalence checker, the number of widenings and the number of narrowings. All experiments were performed on an Intel Xeon W3520 @ 2.66GHz, the server equivalent of an i7 920. The first row refers to the motivating example. The next few rows refer to the examples discussed above. The USVD kernel is often used in embedded systems and is the most complicated case study of Shashidhar [2008]. For this USVD kernel, we show the results of comparing the two input programs with themselves and with each other. As can be seen from the results, we currently do not take advantage of any syntactical equivalence between the two input programs. In USVD 1, some loops have been partially unrolled, which explains the higher number of statements and the higher running time.

For a more extensive experiment, we turned to the polyhedral scanner `CLooG` [Bastoul 2004], which previously used `PolyLib` to perform its iteration domain manipulations, but now uses our own `isl` instead. Due to various differences in the internals of these tools, the outputs for `CLooG`'s regression tests may not be textually identical, and we therefore want to verify that they are equivalent. Since the original statements are not available for these tests, we instead verify that the iterations of all statements are performed in the same order in both versions by

program 1	program 2	cases	stats	comps	d	da time	ec time	∇	Δ
Program 1	Program 2	1	10	16	1	0.011	0.005	1	0
Figure 11 left	Figure 11 right	1	8	18	1	0.005	0.004	0	0
Figure 12 left	Figure 12 right	1	10	14	1	0.006	0.003	1	1
Figure 13 left	Figure 13 right	1	6	12	1	0.005	0.003	1	0
Figure 14	Figure 14	1	6	10	2	0.009	0.003	2	0
Figure 15(a)	Figure 15(b)	1	6	8	1	0.007	0.006	1	0
USVD 1	USVD 1	1	620	628	2	2.620	1.770	4	0
USVD 2	USVD 2	1	134	142	3	0.721	0.086	10	0
USVD 1	USVD 2	1	377	385	3	1.670	0.335	4	0
CLooG-isl 1	CLooG-PL 1	1	133	135	3	0.309	0.199	51	0
CLooG-isl 2	CLooG-PL 2	1	1090	1092	3	9.765	4.134	108	0
CLooG-isl 3	CLooG-PL 3	1	21	23	5	0.235	0.152	20	0
CLooG-isl 4	CLooG-PL 4	1	20	22	2	1.531	0.295	10	0
CLooG-isl	CLooG-PL	113	3558	3804	5	23.540	8.584	923	0
CLooG-isl-c	CLooG-PL-c	113	3558	3804	5	23.662	10.936	1017	0
CLooG-isl	CLooG-PL'	113	3975	4221	5	25.860	19.161	1528	102
CLooG-isl-c	CLooG-PL'-c	113	3975	4221	5	25.788	20.671	1684	0

Table I. Experimental results of equivalence checking. Meaning of the columns: program 1 and 2: input programs; cases: number of pairs of programs; stats: number of assignment statements; comps: number of computations; d: maximal dimension of iterations domains; da time: dependence analysis time (in seconds); ec time: equivalence checking time (in seconds); ∇: number of widenings; Δ: number of narrowings.

passing around a token. Since each statement now writes to the same scalar, these tests constitute true stress tests for both the dependence analysis and the equivalence checking. In particular, using the original statements would result in a much easier equivalence checking problem. The final four rows of the table summarize the results of these experiments using CLooG-0.16.3. The first of these rows represents the case where we simply take the outputs of identical versions of CLooG, but using different polyhedral libraries. The number of statements in these tests ranges from 4 to 1090 with running times up to 4 seconds (all but one are well below 1 second) and 9 seconds in total. The number of widening steps performed ranges from 0 to 108, with a grand total of 923 widening steps. The preceding rows show details of some individual test cases: `reservoir/QR`, `swim`, `thomasset` and `reservoir/liu-zhuge1`. Of the 113 test cases, two cannot be proved equivalent. The reason is that CLooG takes some constraints on the parameters as input. The generated code is only valid for values of the parameters that satisfy these constraints, but the constraints are not explicitly available in the generated code. In the second of the final four rows, we represent the results when these constraints on the parameters are given as an extra argument to the equivalence checker. In this case, all 113 test cases are proved to be equivalent. The final two rows compare outputs where not only the polyhedral library used is different, but also different options are used. In particular, in the primed version, the `backtrack` option is turned on. In this case, 7 test cases are not equivalent for values of the parameters not satisfying the extra parameter constraints.

Note that there is a fairly strong correlation between the dependence analysis time and the equivalence checking time, showing that the equivalence checking time is mostly dependent on the complexity of the dependence graph, rather than

			Barthou et al. [2002]		simplified	
program 1	program 2	cases	equivalent	time (s)	equivalent	time (s)
Program 1	Program 2	1	0/1	0.004	0/1	0.011
Figure 11 left	Figure 11 right	1	0/1	0.002	0/1	0.005
Figure 12 left	Figure 12 right	1	0/1	0.012	0/1	0.016
Figure 13 left	Figure 13 right	1	0/1	0.016	0/1	0.021
Figure 14	Figure 14	1	0/1	0.146	0/1	0.147
Figure 15(a)	Figure 15(b)	1	1/1	0.011	1/1	0.013
USVD 1	USVD 1	1	0/0	N/A	1/1	12.246
USVD 2	USVD 2	1	0/0	N/A	0/0	N/A
USVD 1	USVD 2	1	0/0	N/A	1/1	7.292
CLooG-isl 1	CLooG-PL 1	1	0/1	6.979	1/1	7.189
CLooG-isl 2	CLooG-PL 2	1	0/0	N/A	0/0	N/A
CLooG-isl 3	CLooG-PL 3	1	0/0	N/A	0/0	N/A
CLooG-isl 4	CLooG-PL 4	1	0/0	N/A	0/0	N/A
CLooG-isl	CLooG-PL	113	51/82	1838.893	55/77	4756.963

Table II. Experimental results of equivalence checking for transitive closure based methods. Meaning of the columns: program 1 and 2: input programs; cases: number of pairs of programs; equivalent: number of program pairs that could be proved equivalent over number of program pairs that could be handled; time: time in seconds required for the handled pairs of programs.

the number of statements or the dimension of the iteration domains. Also note that the only pairs of programs in Table I that require any narrowing are those that are not actually equivalent. This shows that our widening operator is usually fairly accurate and does not lead to an over-approximation. In other experiments, not listed in the table, we have seen that the widening step may in rare cases also perform an inappropriate generalization, from which it will then be difficult to recover. In particular, this may occur in the presence of integer divisions more intricate than those in Figure 13. We are investigating if delaying the widening by one step or the use of more advanced widening or narrowing operators can solve these problems.

Table II summarizes the results of applying our implementation of the approach of Barthou et al. [2002] to the same tests in columns 4 and 5. A first observation is that the equivalence checker does not always produce an answer, either because it times out (after 1 hour) or because it runs out of memory (2 GB), as it does in 9, respectively 22 out of 113 CLooG test cases. It should be noted that we have made no attempt to optimize our implementation of this approach and that we may not be using VAUCANSON in the most optimal way. The second and more important observation is that of those cases that can be handled, many cannot be proved equivalent. In the case of the program in Figure 12, this is the right conclusion. In other cases, this is due to the limitations of this approach highlighted above. Columns 6 and 7 present the results of the simplified variation on the approach of Barthou et al. [2002]. Although this variation can correctly prove correctness in more instances than the original approach, it appears to require more memory and therefore also runs out of memory in more instances.

7. RELATED WORK

Many approaches have been proposed for checking the equivalence of two programs, but few of these approaches handle non-trivial recurrences. Translation validation (e.g., [Necula 2000]) checks the equivalence of the input and output of compiler passes, but typically relies on compiler hints or heuristics. In particular, the "permute" rule used by these approaches to prove equivalence of loops requires a bijective function between loop iterations and this function is either assumed to be provided by the compiler [Zuck et al. 2005] or inferred using a simple heuristic [Kundu et al. 2009]. These approaches also require any reordered pair of statement iterations that access the same memory to commute. By contrast, we use array dataflow analysis and automatically derive the correspondences between iterations of loops that have undergone any affine transformation, including all of the simple loop transformations considered by Kundu et al. [2009]. Fractal symbolic analysis [Mateev et al. 2001] takes two programs as input and applies a number of simplification rules until the two programs can be proved equivalent by symbolic analysis. Similarly to translation validation, the simplification rules are derived from the transformation that has been applied to obtain one program from the other. Furthermore, there is a risk of simplifying too much. SMT solvers such as CVC3 [Barrett and Tinelli 2007], used by many approaches, do not perform inductions. General theorem provers such as ACL2 [Kaufmann et al. 2000] can perform induction, but even for the simple case of Figure 1 an encoding of the equivalence problem by an expert required a manual specification of the induction hypothesis, while we perform induction fully automatically.

The motivation for regression verification [Godlin and Strichman 2008; Godlin and Strichman 2009] is similar to ours, but their approach is largely complementary. They handle a larger class of programs, including programs with recursion, but they do not handle loops, unless they have been converted to recursion in a preprocessing step. However, because of the way they match functions in the different programs, such a conversion precludes them from checking the equivalence of any pair of programs that have undergone any non-trivial loop transformation. A very simple such loop transformation, not handled by regression verification, was shown in Figure 15. By contrast, loop transformations are the main focus of our work.

The most closely related approaches are those of Shashidhar et al. [2005] and Barthou et al. [2002]. As explained before, both these approaches are based on transitive closures and therefore require uniform recurrences, unlike our widening based approach. Note that standard uniformization techniques [Manjunathaiah et al. 2001] would only introduce an extra (easy) transitive closure, without resolving the original difficult transitive closure. These approaches also do not (fully) handle associative or commutative operations and require the input programs to be in DSA form. Neither of these approaches, including our own, supports data-dependent or non-affine constructs. A limited form of data-dependent indexing is supported by an extension of the work presented here [Verdoolaege et al. 2010]. Handling more general such constructs would require the use of fuzzy dataflow analysis [Barthou et al. 1997] instead of exact dataflow analysis.

Another way of looking at our work is that we discover invariants between array indices of two programs. Tuples satisfying the invariant identify equal array ele-

ments. While the discovery is guided by the assumed invariant between program outputs, non-trivial new invariants are induced when handling recurrences. Induction of variants —between scalars— is an active research area (e.g., [Müller-Olm and Seidl 2004]).

8. CONCLUSION

We have presented a novel, fully automated, approach to the equivalence checking problem of static affine programs that uses a widening operator instead of relying on a transitive closure operator. Our method is not restricted to uniform recurrences, supports commutative and associative operations with a fixed number of arguments and has a publicly available implementation.

Our approach suffers some limitations. While some of these have been lifted in our later work, some others still remain. In particular, our approach should be extended to also handle reductions, i.e., associative operators with an unbounded number of arguments. Although we have tackled a limited form of data dependent index expressions in later work, further work is also needed to handle more general data dependent and non-affine constructs. Part of the solution is likely to involve a replacement of exact dataflow analysis by fuzzy dataflow analysis, but the equivalence checking procedure will also have to be adapted accordingly.

Acknowledgements

We thank Louis-Noël Pouchet for showing us how to use `VAUCANSON` and the anonymous reviewers for their suggestions. This research was supported by FWO-Vlaanderen, project G.0232.06N.

REFERENCES

Absar, M. J., Marchal, P., and Catthoor, F. 2005. Data-access optimization of embedded systems through selective inlining transformation. In *Proceedings of the 2005 3rd Workshop on Embedded Systems for Real-Time Multimedia, ESTImedia 2005, September 22-23, 2005, New York Metropolitan Area, USA*, M. Miranda and S. Ha, Eds. IEEE Computer Society, 75–80.

Alias, C. and Barthou, D. 2003. On the recognition of algorithm templates. In *Int. Workshop on Compilers Optimization Meets Compiler Verification*. ENTCS, vol. 82. Elsevier Science, Warsaw, 395–409.

Barrett, C. and Tinelli, C. 2007. CVC3. In *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, W. Damm and H. Hermanns, Eds. Lecture Notes in Computer Science, vol. 4590. Springer-Verlag, 298–302. Berlin, Germany.

Barthou, D., Collard, J.-F., and Feautrier, P. 1997. Fuzzy array dataflow analysis. *J. Parallel Distrib. Comput. 40,* 2, 210–226.

Barthou, D., Feautrier, P., and Redon, X. 2002. On the equivalence of two systems of affine recurrence equations. In *Euro-Par Conference*. Lect. Notes in Computer Science, vol. 2400. Springer-Verlag, Paderborn, 309–313.

Bastoul, C. 2004. Code generation in the polyhedral model is easier than you think. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, Washington, DC, USA, 7–16.

Catthoor, F., Danckaert, K., Kulkarni, C., Brockmeyer, E., Kjeldsberg, P., Van Achteren, T., and Omnés, T. 2002. *Data access and storage management for embedded programmable processors*. Kluwer Academic Publishers, Boston, USA.

Chen, C. 2009. Omega+ library. `http://ctop.cs.utah.edu/downloads/omega.tar.gz`.

Cook, W., Rutherford, T., Scarf, H. E., and Shallcross, D. F. 1991. An implementation of the generalized basis reduction algorithm for integer programming. Cowles Foundation Discussion Papers 990, Cowles Foundation, Yale University. Aug.

Cousot, P. and Cousot, R. 1992. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, M. Bruynooghe and M. Wirsing, Eds. LNCS 631, Springer-Verlag, Leuven, Belgium, 269–295.

Feautrier, P. 1988. Array expansion. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*. ACM Press, 429–441.

Feautrier, P. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming 20,* 1, 23–53.

Feautrier, P. 1996. *The Data Parallel Programming Model*. LNCS, vol. 1132. Springer-Verlag, Chapter Automatic Parallelization in the Polytope Model, 79–100.

Franke, B. and O'Boyle, M. 2003. Array recovery and high-level transformations for DSP applications. *ACM Transactions on Embedded Computing Systems 2,* 2 (May), 132–162.

Fu, Q., Bruynooghe, M., Janssens, G., and Catthoor, F. 2006. Requirements for constraint solvers in verification of data-intensive embedded system software. In *Proceedings of the 1st Workshop on constraints in Software Testing, Verification and Analysis*, B. Blanc, A. Gotlieb, and C. Michel, Eds. 46–57.

Godlin, B. and Strichman, O. 2008. Inference rules for proving the equivalence of recursive procedures. *Acta Inf. 45,* 6, 403–439.

Godlin, B. and Strichman, O. 2009. Regression verification. In *46th Design Automation Conference (DAC'09)*. 466–471.

Irigoin, F., Jouvelot, P., and Triolet, R. 1991. Semantical interprocedural parallelisation: An overview of the PIPS project. In *ACM International Conference on Supercomputing, ICS'91*.

Karr, M. 1976. Affine relationships among variables of a program. *Acta Informatica 6*, 133–151.

Kaufmann, M., Moore, J. S., and Manolios, P. 2000. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA.

Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., and Wonnacott, D. 1996. The Omega library. Tech. rep., University of Maryland. Nov.

Kelly, W., Pugh, W., Rosser, E., and Shpeisman, T. 1996. Transitive closure of infinite graphs and its applications. *Int. J. Parallel Program. 24,* 6, 579–598.

Kundu, S., Tatlock, Z., and Lerner, S. 2009. Proving optimizations correct using parameterized program equivalence. *SIGPLAN Not. 44,* 6 (June), 327–337.

Lombardy, S., Régis-Gianas, Y., and Sakarovitch, J. 2004. Introducing VAUCANSON. *Theor. Comput. Sci. 328,* 1-2, 77–96.

Manjunathaiah, M., Megson, G. M., Rajopadhye, S. V., and Risset, T. 2001. Uniformization of affine dependance programs for parallel embedded system design. In *ICPP 2002, Proceedings*, L. M. Ni and M. Valero, Eds. IEEE Computer Society, 205–213.

Mateev, N., Menon, V., and Pingali, K. 2001. Fractal symbolic analysis. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*. ACM, New York, NY, USA, 38–49.

Matsumoto, T., Seto, K., and Fujita, M. 2007. Formal equivalence checking for loop optimization in C programs without unrolling. In *ACST'07: Proceedings of the third conference on IASTED International Conference*. ACTA Press, Anaheim, CA, USA, 43–48.

Müller-Olm, M. and Seidl, H. 2004. Precise interprocedural analysis through linear algebra. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004*. 330–341.

Necula, G. C. 2000. Translation validation for an optimizing compiler. *SIGPLAN Not. 35,* 5, 83–94.

Shashidhar, K. C. 2008. Efficient automatic verification of loop and data-flow transformations by functional equivalence checking. Ph.D. thesis.

Shashidhar, K. C., Bruynooghe, M., Catthoor, F., and Janssens, G. 2005. Verification of source code transformations by program equivalence checking. In *CC 2005, Proceedings*. Lecture Notes in Computer Science, vol. 3443. Springer-Verlag, Berlin, 221–236.

van Engelen, R. A. and Gallivan, K. A. 2001. An efficient algorithm for pointer-to-array access conversion for compiling and optimizing DSP applications. In *Innovative Archs. for Future Gen. High-Perf. Processors and Systems*. IEEE, 80–89.

Verdoolaege, S. 2010. isl: An integer set library for the polyhedral model. In *Mathematical Software - ICMS 2010*, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds. Lecture Notes in Computer Science, vol. 6327. Springer Berlin / Heidelberg, 299–302.

Verdoolaege, S. and Bruynooghe, M. 2008. Algorithms for weighted counting over parametric polytopes: A survey and a practical comparison. In *The 2008 International Conference on Information Theory and Statistical Learning*, M. Beck and T. Stoll, Eds.

Verdoolaege, S., Cohen, A., and Beletska, A. 2011. Transitive closures of affine integer tuple relations and their overapproximations. In *Proceedings of the 18th international conference on Static analysis*. SAS'11. Springer-Verlag, Berlin, Heidelberg, 216–232.

Verdoolaege, S. and Grosser, T. 2012. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France.

Verdoolaege, S., Janssens, G., and Bruynooghe, M. 2009. Equivalence checking of static affine programs using widening to handle recurrences. In *Computer Aided Verification 21*. Springer, 599–613.

Verdoolaege, S., Palkovic, M., Bruynooghe, M., Janssens, G., and Catthoor, F. 2010. Experience with widening based equivalence checking in realistic multimedia systems. *Journal of Electronic Testing 26,* 2, 279–292.

Verma, M. and Marwedel, P. 2007. *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*. Springer Publishing Company, Incorporated.

Zuck, L. D., Pnueli, A., Goldberg, B., Barrett, C. W., Fang, Y., and Hu, Y. 2005. Translation and run-time validation of loop transformations. *Formal Methods in System Design 27,* 3, 335–360.