Equivalence Checking of Static Affine Programs Using Widening to Handle Recurrences

Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe

Katholieke Universiteit Leuven, Department of Computer Science,
Celestijnenlaan 200A, B-3001 Leuven, Belgium

Abstract. Designers often apply manual or semi-automatic loop and data transformations on array and loop intensive programs to improve performance. The transformations should preserve the functionality, however, and 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 handle 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.

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. 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 accesses.

Figure 1 shows a toy example of a pair of programs for which we would like to prove equivalence. Both programs have the same input array `In` and output array `Out` (a scalar in this case) 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). We neither assume an a priori correspondence between the other arrays (the temporary arrays) of both programs nor between their loops.

The equivalence checking of static affine programs has been previously investigated by Barthou et al. [1, 6] and Shashidhar et al. [20, 21]. A major challenge in this line of research is posed by recurrences, i.e., a statement in a loop that (indirectly) depends on previous iterations of the same statement. 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., [17]) 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 our running example (Figure 1). All the previous work

A. Bouajjani and O. Maler (Eds.): CAV 2009, LNCS 5643, pp. 599–613, 2009.

```
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

```
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    A[i]=B[i]+C[i];}
10 Out=A[N-1];
```

(b) Program 2

Fig. 1. Two programs with a recurrence; assuming that + is commutative they are equivalent

mentioned above relies on the transitive closure operation [14] provided by the Omega library [13] to handle recurrences, effectively restricting 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. However, as stated in Section 9.3.1 of [20], this proposal has not been implemented. Moreover, it is unable to handle the commutativity of + in Figure 1 as the order of the arguments has been reversed for only half of the iterations of the loop.

Furthermore, all the above approaches require both programs to be in dynamic single assignment (DSA) form [9], i.e., such that each array element is written at most once, and none of the 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, loop unrolling, loop splitting, loop peeling and data-reuse transformations. However, unlike those approaches, ours

- handles programs that perform destructive updates without a preprocessing step that converts them to dynamic single assignment form,
- handles both uniform and non-uniform recurrences by not relying on a transitive closure operation, and
- has a publicly available implementation,
- with full support for associative and commutative operations with a fixed number of arguments.

We define the concept of a dependence graph in Section 2, which we then use as input for the equivalence checking method of Section 3. Section 4 has implementation details and Section 5 the final discussion.

2 Program Model

Two programs will be 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 [20, 21] where an array based representation is used and some dependence analysis is implicitly performed during the equivalence checking, we separate the dependence analysis from the equivalence checking, the latter working on the output of the former. This separation allows us to use standard exact dataflow analysis [10] or, in future work, fuzzy dataflow analysis [5]. The resulting *dependence graph* is essentially a DSA representation of the program, but without rewriting the source program in that form as in [20, 21].

For simplicity we assume that input and output arrays are given, that input arrays are only read and output arrays only written, that each read value is either input or has been written before, and that there is only one output array. These assumptions can easily be relaxed. The use of exact dataflow analysis implies the usual restrictions of static affine programs, i.e., static control flow, quasi-affine loop bounds and quasi-affine index expressions. Recall that quasi-affine expressions consist of additions, constant multiplication and integer division by a constant. We also assume that all functions called in the program are pure.

We will now first present the definition of a dependence graph and then explain how such a dependence graph can be constructed from dataflow analysis. In what follows, "$\subset$" means "strict subset" and "$\subseteq$" means "subset". All sets and relations may involve parameters.

Definition 1 (Dependence Graph). *A* dependence graph *is a connected directed graph $G =< V, E >$ with a designated* output vertex $v_0 \in V$ *with in-degree 0 and a set $I \subset V$ of* input vertices*, each with out-degree 0. The graph may have loops and parallel edges. Each vertex $v \in V$ is adorned by a tuple $< d_v, D_v, f_v, r_v, l_v >$, with*

- d_v *a non-negative integer, called the* dimension *of the node*
- D_v *a set of d_v-tuples of integers, called the* iteration domain *of the node*
- f_v *a literal, called the* operation *of the node*
- r_v *a non-negative integer, called the* arity *of the node*
- l_v *a literal, called the* location *of the node*

and each edge $e = (u, v) \in E$ is adorned by a pair $< p_e, M_e >$, with

- p_e *a non-negative integer with $1 \leq p_e \leq r_u$, called the* argument position
- M_e *a set of $(d_u + d_v)$-tuples of integers, called the* dependence relation*.*

Moreover, the following constraints are satisfied. Firstly, $\forall u \in V, \forall \boldsymbol{x} \in D_u, \forall p \in [1, r_u] : \exists! \, e = (u, v) \in E : p_e = p$ and $\boldsymbol{x} \in dom(M_e)$, i.e., the domains of the

dependence relations of the edges emanating from a node for a given argument position partition the domain of the node. The unique edge corresponding to a point $\boldsymbol{x}$ *and an argument position* p *is denoted* $e_G(\boldsymbol{x}, p)$*. Secondly, 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.*

To represent a program by a dependency graph, we use the vertices to represent *computations*. We distinguish three kinds of vertices/computations: An output computation for the output array, an input computation for each input array and a computation for each function call, each operation and each copy statement. The dependence graph of Program 1 (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, one for the addition, one for the computation of f, and one for the computation of g (all from Line 3) and finally one for the copy statement in Line 1. As for the annotations of a node v, the pair (d_v, D_v) denotes the set of elements for which the computation is performed. For input and output computations, d_v is the dimension of the respective arrays and the domain is the set of its elements. As the output of Program 1 is a scalar, the dimension is 0 and the single element is denoted as (). The input array is one dimensional and the domain consists of the set of elements from 0 to $N-1$. For other computations, the dimension is determined by the loop nest controlling the computation, while the domain describes the iterations for which the computation is performed (dimension 1 and elements 1 to $N-1$ for the computations of Line 3). The pair (f_v, r_v) reflects the operation being performed and its arity. For input computations, f_v is the name of the input array while r_v is set to 0; for the output computation, f_v is the name of the output array and r_v is set to 1. Copy operations are represented by id (for identity) with arity 1 and the other operations by the name of the operation and its arity. The l_v annotation refers to the program line where the computation is performed; it is not defined for input and output computations.

Edges arise in three different ways. If, within a given statement, the result of some function f is used as the jth argument of some function g, then an edge e is added from the computation u corresponding to g to the computation v corresponding to f, with $p_e = j$ and $M_e = \{(\boldsymbol{i}; \boldsymbol{i}) \mid \boldsymbol{i} \in D_u\}$. Note that $D_u = D_v$ here since both operations appear in the same statement. Also note that for any relation $M \subseteq \mathbb{Z}^{d_u} \times \mathbb{Z}^{d_v}$, we separate the d_u input dimensions from the d_v output dimensions by a ";". In general, either or both of d_u and d_v may be zero. Examples of intra-statement dependences in Figure 2 are the edges from the addition (computation e) to respectively the computations c and d. If the function of computation u takes an array element as its jth argument and this array element has been "computed" by the function of computation v (this includes copy and input computations), then an edge $e = (u, v)$ is added, with $p_e = j$ and M_e relating the iterations of u with the corresponding iterations of v. Dataflow analysis is used to find v and to compute M_e by identifying the last statement iteration that wrote to the array. Since there is exactly one such last iteration for each of the array

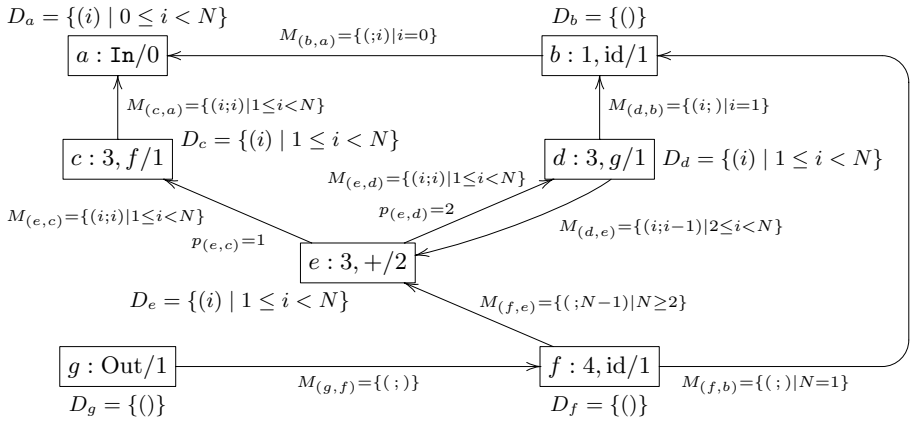

Fig. 2. Dependence graph of Program 1 in Figure 1. Computations are named from a to g with a the input and g the output computation. Each computation v is represented as "$v : l_v, f_v/r_v$" (l_v is absent for input and output computations). To avoid clutter, the dimension is omitted, while the domain is shown next to the box with the node; also the argument position p_e is only indicated on edges emanating from node e (it is 1 on all other edges).

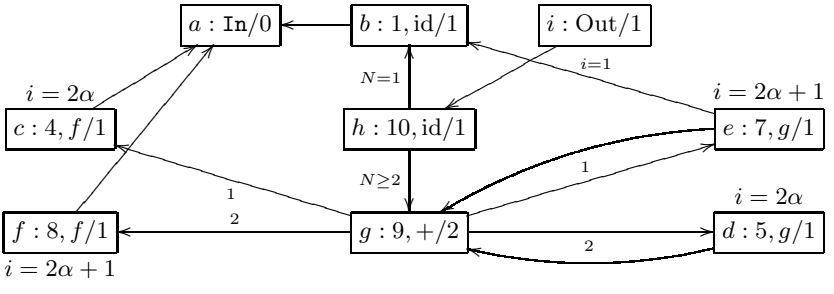

Fig. 3. Dependence graph of Program 2 in Figure 1 (some details omitted)

elements, the first constraint of Definition 1 is satisfied. The second constraint is satisfied because a statement iteration can only depend on a *previous* iteration in the execution order. For example, dataflow analysis identifies an edge from vertex f to vertex b with mapping $\{(;)\}$ (the single iteration in the zero-dimensional iteration domain of f is mapped to the single iteration of b) for $N = 1$ and an edge from vertex f to vertex e with mapping $\{(;N-1)\}$ for $N \ge 2$ (the single iteration of f is mapped to iteration $N-1$ of the one-dimensional iteration domain of computation e). Finally, for each computation v that last wrote an element of the output array (dataflow analysis determines which v), an edge is added from the output computation to v, with as M_e the reversed access relation of the write. In our running example, $M_{(g,f)} = \{(;)\}$ as a scalar of dimension 0 is written. Note

that input computations do not have outgoing edges and the output computation does not have incoming edges.

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 this section.

Definition 2 (Equivalence of Computation Iterations). *An iteration* $\boldsymbol{x}_1 \in D_{v_1}$ *of a computation* $v_1 \in V_1$ *in a dataflow graph* G_1 *is equivalent to an iteration* $\boldsymbol{x}_2 \in D_{v_2}$ *of a computation* $v_2 \in V_2$ *in a dataflow 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* $\boldsymbol{x}_1 = \boldsymbol{x}_2$,
- $f_{v_1} = \mathrm{id}$ *and iteration* $M_{e_{G_1}(\boldsymbol{x}_1,1)}(\boldsymbol{x}_1)$ *of* v_1' *with* $e_{G_1(\boldsymbol{x}_1,1)} = (v_1, v_1')$ *is equivalent to iteration* $\boldsymbol{x}_2$ *of* v_2 *(and similarly for* $f_{v_2} = \mathrm{id}$*), or*
- $(f_{v_1}, r_{v_1}) = (f_{v_2}, r_{v_2})$ *and for each* $p \in [1, r_{v_1}]$*, iteration* $M_{e_{G_1}(\boldsymbol{x}_1,p)}(\boldsymbol{x}_1)$ *of* v_1' *with* $e_{G_1(\boldsymbol{x}_1,p)} = (v_1, v_1')$ *is equivalent to iteration* $M_{e_{G_2}(\boldsymbol{x}_2,p)}(\boldsymbol{x}_2)$ *of* v_2' *with* $e_{G_2(\boldsymbol{x}_2,p)} = (v_2, v_2')$.

Definition 3 (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 equivalent.*

3 Equivalence Checking

In order to prove equivalence of two dependence graphs we basically follow Definition 2 and propagate from the output to the input what correspondences between computation iterations we should prove. Once we hit computations with zero out-degree (either input computations or symbolic constants), we propagate back to the output what we have actually been able to prove. This two-way propagation is different from the approaches in [1, 6, 20, 21], the authors of which essentially only propagate information from output to input. There are several reasons for this difference in approach. Firstly, the discrepancy between what has to be proven and what is actually proven helps in debugging when the equivalence proof fails; secondly, as will become clear, propagating both ways will facilitate a better treatment of commutativity and recurrences.

The propagation from output to input constructs an equivalence tree. The nodes in this equivalence tree are annotated with *correspondences*, that reflect the correspondences that we intend to proof. During the reverse traversal of the equivalence tree, the correspondences are updated to reflect what we have actually been able to prove.

Definition 4 (Correspondence). *Let* G_i $(i \in \{1, 2\})$ *be the dependency graph of program* P_i*. A correspondence consists of a tuple* $(v_1, v_2, R^{\mathrm{want}}, R^{\mathrm{got}})$ *with* v_i *a computation in* G_i *with domain* D_i*,* $R^{\mathrm{want}} \subseteq D_1 \times D_2$*, and* $R^{\mathrm{got}} \subseteq D_1 \times D_2$.

R^{want} contains pairs of computation iterations for which we want to prove equivalence R^{got} is initially undefined and is later updated to the set of pairs of computation iterations that we have actually proven to be equivalent.

The initial equivalence tree consists of a single root node that models the equivalence to be proven between the output arrays of both programs. For this root, we have $R^{\text{want}} = \{(\boldsymbol{i}, \boldsymbol{i}) \mid \boldsymbol{i} \in D\}$ with D the domain of the output computation (i.e., the domain of the output array). It expresses the intention to show that both arrays are identical. R^{got} is initially undefined; it is computed by up propagation when the R^{got} information of its children is available. The proof is successful when $R^{\text{want}} = R^{\text{got}}$. $R^{\text{want}} \setminus R^{\text{got}}$ shows for which elements of the output array equivalence could not be proven.

In our running example, the output is scalar, hence $R^{\text{want}} = \{(\,;\,)\}$. It is convenient to see more details in concrete examples, so instead of using $(v_1, v_2, R^{\text{want}}, R^{\text{got}})$, we use $\langle (l_{v_1}, f_{v_1}) \leftrightarrow (l_{v_2}, f_{v_2}) \rangle$ and keep R^{want} and R^{got} as separate annotations. Hence, we represent the root as $\langle \text{Out} \leftrightarrow \text{Out} \rangle$ (Figure 4).

The basic step in proving equivalence consists of *propagation*. We distinguish between *down* (from root to leaves) and *up* (from leaves to root) propagation.

Down Propagation. Down propagation reduces the correspondence $(v_1, v_2, R^{\text{want}}, \_)$ of a node n to a set of correspondences $(u_1, u_2, R_c^{\text{want}}, \_)$ in which u_i is the target of an outgoing edge of v_i in the dependency graph G_i. Each of these new correspondences annotates a child of node n. The dataflow encoded in the dependency graphs is used to derive the R_c^{want} relation of the children. More precisely, for each pair of computations u_1 and u_2 with (v_1, u_1) and (v_2, u_2) edges in the respective dependency graphs that refer to the same argument position (same p value), we have a child annotated with the correspondence $(u_1, u_2, R_c^{\text{want}}, \_, \_)$ where

$$R_c^{\text{want}} = \left(M_{(v_1,u_1)} \oplus M_{(v_2,u_2)}\right) R^{\text{want}}. \tag{1}$$

The $\oplus$ operator combines mappings of type $D_{v_1} \to D_{u_1}$ and $D_{v_2} \to D_{u_2}$ into one of type $D_{v_1} \times D_{v_2} \to D_{u_1} \times D_{u_2}$:

$$\{\, (\boldsymbol{i}_1, \boldsymbol{i}_2; \boldsymbol{i}'_1, \boldsymbol{i}'_2) \in \mathbb{Z}^{(d_{v_1}+d_{v_2})+(d_{u_1}+d_{u_2})} \mid (\boldsymbol{i}_1, \boldsymbol{i}'_1) \in M_{(v_1,u_1)} \wedge (\boldsymbol{i}_2, \boldsymbol{i}'_2) \in M_{(v_2,u_2)} \,\}.$$

Recall, 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^{want} is partitioned by the domains of the combined dependence relations, i.e., each element of R^{want} is mapped to an element of the R_c^{want} of exactly one child corresponding to this argument position.

In our running example, the output computations have one outgoing edge, hence one child is created. We obtain the child $\langle (4, \text{id}/1) \leftrightarrow (10, \text{id}/1) \rangle$ with $R^{\text{want}} = \{(\,;\,)\}$; i.e., the copy operations in Line 4 of Program 1 and Line 10 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 $R^{\text{want}} = \emptyset$. Of the two other children, one is constrained with $N = 1$ and the other with $N \geq 2$. The correspondence of the latter, $\langle (3, +/2) \leftrightarrow (9, +/2) \rangle$, has $R^{\text{want}} = \{(N-1; N-1) \mid N \geq 2\}$ expressing that the addition on Line 3 of Program 1 and the addition on Line 9 of Program 2 should compute the same value in iteration $N-1$ (when $N \geq 2$).

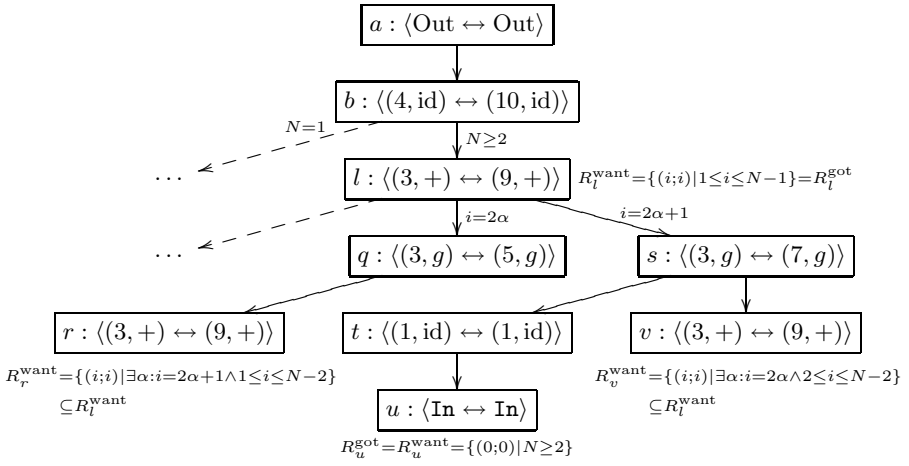

Fig. 4. Final state of part of the equivalence tree of the programs in Figure 1

In principle, this down propagation proceeds until correspondences are reached that cannot be further propagated. This holds when (i) $R^{\text{want}} = \emptyset$, in which case we can set $R^{\text{got}} = \emptyset$. (ii) R^{want} is a relation between input computations referring to the same input array: we can set $R^{\text{got}} = R^{\text{want}} \cap Id$ with Id the identity relation, i.e., the tuples in R^{want} between identical elements of the input arrays are proven, the others cannot be proven. (iii) The functions (f_{v_i} with arity r_{v_i}) of both computations are different; this includes the case of input computations referring to distinct input arrays. As we consider functions as black boxes, we cannot prove equivalences between different functions and we set $R^{\text{got}} = \emptyset$.

Up propagation. Once the R^{got} relations are obtained for all children of an equivalence node, up propagation can compute the R^{got} relation for that node. First, consider a single argument position. As we mentioned above, down propagation distributed the R^{want} relation of n over the R^{want} relation of the children of n that refer to the same argument position. Hence up propagation should take the union of the different parts. However, to prove an equivalence, it has to be proven for each of the r argument positions, so the results obtained for the different argument positions must be intersected. More formally, let $(v_1, v_2, R^{\text{want}}, R^{\text{got}})$ annotate node n and let $S_j = \{((v_1, u_1), (v_2, u_2)) \mid p_{(v_1,u_1)} = j \wedge p_{(v_2,u_2)} = j\}$. Finally, let $(u_1, u_2, R^{\text{want}}_{(u_1,u_2)}, R^{\text{got}}_{(u_1,u_2)})$ annotate the child of n with $((v_1, u_1), (v_2, u_2)) \in S_j$. Then, R^{got} is updated to

$$\bigcap_{j \in [1..r]} \bigcup_{((v_1,u_1),(v_2,u_2)) \in S_j} (M^{-1}_{(v_1,u_1)} \oplus M^{-1}_{(v_2,u_2)}) R^{\text{got}}_{(u_1,u_2)} \tag{2}$$

Algebraic operations. Associative operators can be nested differently in the two programs. A direct application of our technique would result in invalid pairings of

argument positions and a failure of the proof. To solve this problem, we basically follow [21] and apply a preprocessing step to "flatten" associative operators in the dependency graph and reorganize the graph so that the functionality of the program is preserved. 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 $+(a, b, c)$.

Commutative operators pose more severe problems. For example, when propagating $\langle(3, +/2) \leftrightarrow (9, +/2)\rangle$ (correspondence l in Figure 4) as described above, the first argument of the first computation is paired with the first argument of the second computation, and also the second arguments are paired. When the operator is commutative as it is the case here, this does not suffice. The solution proposed in [21] (without implementation) is 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 too is insufficient. Assume the above correspondence has $R^{\text{want}} = \{(i; i) \mid 1 \leq i \leq N - 1\}$. (This is not the initial value, but the recurrence handling described below updates it to this value.) 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 [21] gets stuck.

Our approach is to try all permutations when the operator is commutative, and to extend the up propagation step so that it collects the results from the different permutations. Formally, let Π the set of permutations over $[1, \ldots, r]$. With π a permutation in Π, the set S_j used in Equation 2 is redefined as $S_j = \{((v_1, u_1), (v_2, u_2)) \mid p_{(v_1,u_1)} = j \wedge p_{(v_2,u_2)} = \pi(j)\}$. Then, with other symbols retaining the same meaning as in Equation 2, R^{got} is now computed as

$$\bigcup_{\pi \in \Pi} \bigcap_{j \in [1..r]} \bigcup_{((v_1,u_1),(v_2,u_2)) \in S_j} (M^{-1}_{(v_1,u_1)} \oplus M^{-1}_{(v_2,u_2)}) R^{\text{got}}_{(u_1,u_2)} \tag{3}$$

(Definition 2 requires a similar change.)

In our running example, the $+/2$ computation of Program 1 (node e of Figure 2) has one outgoing edge for each argument, while node g in the model of Program 2 has 2 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 4 children with $R^{\text{want}} = R^{\text{got}} = \emptyset$. The other cases result in 4 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 $i = 2\alpha+1$. Two of them are shown as children of node l in Figure 4. Once R^{got} is available in each of these children, up propagation can update R^{got} in this node l.

Recurrences: Induction, Widening and Narrowing. A given pair of computations may depend on itself, requiring special care. Termination is ensured when branches of the equivalence tree are finite and calculations in nodes are finite. The former will be ensured by allowing at most two occurrences of the same

computation pair in a branch (the number of different pairs is finite); the latter by recomputing the R^{want} and R^{got} values in nodes only a finite number of times.

A pair (v_1, v_2) can only reappear in a branch when both dependency graphs have a cycle that passes through respectively v_1 and v_2, i.e., both programs have a recurrence. When a pair actually reappears, it means that both programs have performed the same computation since the first appearance of the pair. In the equivalence tree of our running example (Figure 4), nodes l and r (as well as l and v) form such a pair. Initially (node l) one wants to show that both computations are equivalent for the iteration pair $(N-1; N-1)$; down propagation creates nodes q, r, s, and v. In nodes r and v, the same equivalence has to be proven for the iteration pair $(N-2; N-2)$ (for respectively odd and even N). While in our running example, both programs have computed one iteration, the relationship can be more complex. For example, when one program is derived from the other by loop unrolling, the original one needs several iterations to perform the same computation as the transformed program does in one iteration.

More formally, let node a, annotated with $(v_1, v_2, R_a^{\text{want}}, R_a^{\text{got}})$ (with R_a^{got} undefined), be the first occurrence (the "ancestor") of the pair of computations (v_1, v_2) on a branch of an equivalence tree. Let node d annotated with $(v_1, v_2, R_d^{\text{want}}, R_d^{\text{got}})$ be the second occurrence (the "descendant"). If $R_d^{\text{want}} \subseteq R_a^{\text{want}}$, we can simply perform induction, setting $R_d^{\text{got}} = R_d^{\text{want}}$. When up-propagation reaches node a, we will of course need to validate our induction hypothesis. This will be discussed below. If $R_d^{\text{want}} \not\subseteq R_a^{\text{want}}$, then we want to extend the induction hypothesis R_a^{want} to include R_d^{want} so that we can perform induction also in this case. However, we cannot simply set the new R_a^{want} to the union of the old R_a^{want} and R_d^{want}, as this effectively corresponds to loop unrolling. This process would not terminate when the number of iterations is bounded by a symbolic constant (as in our running example) and would not scale when the bound is known.

Instead, we draw inspiration from the widening/narrowing technique of abstract interpretation [8] and apply a *widening* operator ∇. Such a widening operator turns a possibly infinite ascending chain, e.g., taking the union with R_d^{want} in each step as described above, into an eventually stationary chain. As our widening operator, we will essentially use the integer affine hull. However, as the induction hypothesis R_a^{want} needs to be a subset of $D_1 \times D_2$, with D_i the domain of v_i, we will intersect the affine hull with the aforementioned set. In the first iteration, R_a^{want} is then set to the intersection of $D_1 \times D_2$ with some affine subspace. Any additional widening step is only performed when R_d^{want} includes an element not in R_a^{want} (but still in $D_1 \times D_2$) and the widening operator will then increase the dimension of the affine subspace. So, after a finite number of widening steps, $R_a^{\text{want}} = D_1 \times D_2$, ensuring termination. At termination, we will have $R_d^{\text{want}} \subseteq R_a^{\text{want}}$. 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.

In our running example, the ancestor (node l) is created with $R_l^{\text{want}} = \{(N-1; N-1)\}$. Down propagation creates node r (or v, depending on the parity of N)

and computes $R_r^{\text{want}} = \{(N-2; N-2)\}$. This is not part of the initial induction hypothesis and so we need to revise the induction hypothesis. The affine hull of $\{(N-1; N-1)\}$ and $\{(N-2; N-2)\}$ is $\{(i; i)\}$ and intersection with $D_1 \times D_2$ yields the induction hypothesis shown in Figure 4.

As mentioned before, when up-propagation returns to a, we need to check whether the induction steps that have been made are valid. This is the case if $R_d^{\text{want}} \subseteq R_a^{\text{got}}$ for each descendant d, i.e., when $\cup_{d \in D} R_d^{\text{want}} \subseteq R_a^{\text{got}}$ with D the set of descendants for the computation pair (v_1, v_2). Note that there is no risk of circular reasoning ("unfounded sets") due to the second constraint of Definition 1: No individual iteration can (indirectly) depend on itself, hence no pair of individual iterations can depend on itself.

If R_a^{got} does not include R_d^{want}, then the performed induction is not founded by what we actually can prove. This means that R_a^{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 then try to take successive subsets of R_a^{want}. However, as in the first phase, we need to be careful not to end up in a possibly infinite sequence. As in abstract interpretation, we therefore perform a (finite) number of *narrowing* steps. Our narrowing operator is fairly simply. The first time it is applied, we set $R_a^{\text{want}} = \cup_{d \in D} R_d^{\text{want}}$ with D the set of descendants that have used the induction hypothesis. Then, in the descendants, we perform induction, setting $R_d^{\text{got}} = R_d^{\text{want}} \cap R_a^{\text{want}}$. In particular, we do not allow any more widening steps on a once we have entered the narrowing phase. If the resulting R_a^{got} still does not include some R_d^{want}, then the second time we apply the narrowing operator, we simply set $R_a^{\text{want}} = \emptyset$.

The handling of recurrences can be summarized by the following algorithm. It uses a flag f to remember the status of the ancestor node. The flag is initialized to *undef* when a node is created by down propagation.

- If down propagation creates a node d, with $(v_1, v_2, R_d^{\text{want}}, R_d^{\text{got}}, \mathit{undef})$, and there exists an ancestor node a with $(v_1, v_2, R_a^{\text{want}}, R_a^{\text{got}}, f_a)$ then
  - if $f_a = \mathit{undef}$, then set $f_a = \mathit{widening}$ (to indicate that a is the root of a recurrence);
  - if $f_a = \mathit{widening}$ and $\neg(R_d^{\text{want}} \subseteq R_a^{\text{want}})$ then $R_a^{\text{want}} \leftarrow R_a^{\text{want}} \nabla R_d^{\text{want}}$ (widening step) and remove all descendants of node a.
  - else $R_d^{\text{got}} = R_d^{\text{want}} \cap R_a^{\text{want}}$ (induction step, it will be checked later whether it was valid).
- If up propagation computes R_a^{got} for a node a, with $(v_1, v_2, R_a^{\text{want}}, R_a^{\text{got}}, f_a)$, and the node has a set D of descendant nodes with each node $d \in D$ annotated with $(v_1, v_2, R_d^{\text{want}}, R_d^{\text{got}}, f_d)$ then
  - if $\cup_{d \in D} R_d^{\text{want}} \subset R_a^{\text{got}}$ then done (the induction steps turn out to be valid and R_a^{got} can be used to perform up propagation on the parent of a)
  - else if $f_a = \mathit{widening}$ and $\neg(\cup_{d \in D} R_d^{\text{want}} \subseteq R_a^{\text{got}})$ then $f_a = \mathit{narrowing}$, $R_a^{\text{want}} = \cup_{d \in D} R_d^{\text{want}}$, and $R_a^{\text{got}} = \mathit{undef}$ (narrowing step) and remove all descendants of node a
  - else $(f_a = \mathit{narrowing})$ $R_a^{\text{got}} = \emptyset$ (no correct hypotheses found)

Proposition 1. *The equivalence checking algorithm terminates and for each node in the equivalence tree (including the root node) with correspondence* $(v_1, v_2, R^{\text{want}}, R^{\text{got}})$*, if* $(\boldsymbol{x}_1, \boldsymbol{x}_2) \in R^{\text{got}}$ *then iteration* $\boldsymbol{x}_1$ *of* v_1 *is equivalent to iteration* $\boldsymbol{x}_2$ *of* v_2*.*

Our recurrence handling differs substantially from [21]. 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 dependency mapping that corresponds to the computation performed between ancestor and descendant and use that mapping in a complex operation that involves the calculation of the transitive closure (implemented in the Omega library [13]) that yields the equivalences to be proven 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]`.

Tabling. A dependency graph is not necessarily a tree. This means that two computations may have some common subcomputation. Tabling can therefore be used to reuse already proven equivalences. A very simple table could store proven tuples $(v_1, v_2, R_t^{\text{got}})$. When an equivalence has to be proven for the pair of computations (v_1, v_2), one needs only to prove it for $R^{\text{want}} \setminus R_t^{\text{got}}$. The same table can also be used to detect recurrences.

4 Implementation

The proof procedure of Section 3 has been implemented as part of our C++ `isa` (`http://www.kotnet.org/~skimo/loop/isa-0.08.tar.bz2`) prototype tool set. This set contains a polyhedral extractor from C based on SUIF [3] and an exact dependence analysis tool. We use our own C `isl` library, based on `piplib` [11], to manipulate sets of integers defined by linear inequalities and integer divisions. We avoid the Omega library [13] as it suffers from some unimplemented corner cases. Each set/relation is represented by a union of "basic sets", each of which is defined by a conjunction of linear inequalities. If a requested relation R^{want} 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. It also contains various other

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

Fig. 5. A pair of equivalent programs with a non-uniform recurrence

optimizations to avoid redundant computations and it supports multiple output arrays.

It is difficult to compare running times with the most closely related tool of [20] since the latter is not available to us and since the reported running times do not mention the CPU type. Furthermore, our `isl` library is relatively new and uses exact integer arithmetic, while the tool of [20] uses the more mature and presumably heavily optimized Omega library, which has only machine integer precision. As an indication, however, for the example in Figure 1, our tool takes about 0.04s on an Intel Core2@3GHz, 0.04s for the example of Figure 5 with a non-uniform recurrence and 0.02s for the example from [21], while for the USVD example pair from [20], with several hundred lines of code, the tool takes about 0.5s, most of which is spent reading in intermediate data structures. This kernel is often used in embedded systems and is the most complicated case study of [20].

For a more extensive experiment, we turned to the code generation tool `CLooG` [7], which previously used `PolyLib` to perform its iteration domain manipulations, but was recently extended to optionally use 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 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. Of 105 tests, 97 were proven to be equivalent. Five contained a construct in the output that we currently cannot parse, while three produced memory overflows (1 during dependence analysis and 2 during equivalence checking). These overflows are probably due to the presence of a large number of integer divisions. The size of the 97 pairs of checked programs ranges from 2 to 800 lines (9478 lines in total), with running times up to 22 seconds (most are well below 1 second) and 62 seconds in total. The number of widening steps performed ranges from 0 to 228, with a grand total of 1018 widening steps.

5 Discussion and Conclusion

Unlike transitive closure based approaches [6, 21], our widening approach does not require uniform recurrences. Note that standard uniformization techniques [15]

would only introduce an extra (easy) transitive closure, without resolving the original difficult transitive closure. However, our method will not be able to detect all kinds of equivalences, as the widening assumes that a piecewise affine transformation has been applied to the recurrences. 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 5. We are investigating if delaying the widening by one step or the use of more advanced widening or narrowing operators can solve these problems. The flattening of nested associative operators during preprocessing cannot handle reductions with a variable number of arguments such as in $\sum_{0 \le i < N} \mathtt{a}[i]$. Proving the equivalence of different ways of computing such reductions requires a further extension of our system.

Some forms of data-dependent or non-affine constructs can be handled by applying an if-conversion preprocessing [2] and/or using fuzzy [5] instead of exact dataflow analysis. Many other approaches exist to equivalence checking, including translation validation, e.g., [19], or fractal symbolic analysis [16]. Some of these approaches handle more general transformations than ours, but they typically rely on compiler hints or heuristics. SMT solvers such as CVC3 [4], used by many approaches, do not perform inductions. General theorem provers such as ACL2 [12] 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. See [20] for a more detailed comparison to related work.

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 elements. 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., [18].

We conclude that our method is the first static affine program equivalence checker that handles non-uniform recurrences with full support for commutativity and a publicly available implementation.

Acknowledgements. Research supported by FWO-Vlaanderen (G.0232.06N).

References

[1] Alias, C., Barthou, D.: On the recognition of algorithm templates. In: Int. Workshop on Compilers Optimization Meets Compiler Verification, Warsaw, April 2003. ENTCS, vol. 82, pp. 395–409. Elsevier Science, Amsterdam (2003)

[2] Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.D.: Conversion of control dependence to data dependence. In: POPL 1983, pp. 177–189. ACM, New York (1983)

[3] Amarasinghe, S., Anderson, J., Lam, M.S., Tseng, C.-W.: An overview of the SUIF compiler for scalable parallel machines. In: Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing (1995)

[4] Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)

[5] Barthou, D., Collard, J.-F., Feautrier, P.: Fuzzy array dataflow analysis. J. Parallel Distrib. Comput. 40(2), 210–226 (1997)
[6] Barthou, D., Feautrier, P., Redon, X.: On the equivalence of two systems of affine recurrence equations. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 309–313. Springer, Heidelberg (2002)
[7] Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT 2004: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, Washington, DC, USA, 2004, pp. 7–16. IEEE Computer Society, Los Alamitos (2004)
[8] Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
[9] Feautrier, P.: Array expansion. In: ICS 1988: Proceedings of the 2nd international conference on Supercomputing, pp. 429–441. ACM Press, New York (1988)
[10] Feautrier, P.: Dataflow analysis of array and scalar references. International Journal of Parallel Programming 20(1), 23–53 (1991)
[11] Feautrier, P., Collard, J., Bastoul, C.: Solving systems of affine (in)equalities. Technical report, PRiSM, Versailles University (2002)
[12] Kaufmann, M., Moore, J.S., Manolios, P.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Norwell (2000)
[13] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The Omega library. Technical report, University of Maryland (November 1996)
[14] Kelly, W., Pugh, W., Rosser, E., Shpeisman, T.: Transitive closure of infinite graphs and its applications. Int. J. Parallel Program. 24(6), 579–598 (1996)
[15] Manjunathaiah, M., Megson, G.M., Rajopadhye, S.V., Risset, T.: Uniformization of affine dependance programs for parallel embedded system design. In: Ni, L.M., Valero, M. (eds.) ICPP 2002, Proceedings, pp. 205–213. IEEE Computer Society, Los Alamitos (2001)
[16] Mateev, N., Menon, V., Pingali, K.: Fractal symbolic analysis. In: ICS 2001: Proceedings of the 15th international conference on Supercomputing, pp. 38–49. ACM, New York (2001)
[17] Matsumoto, T., Seto, K., Fujita, M.: Formal equivalence checking for loop optimization in C programs without unrolling. In: IASTED Proc. ACST 2007: Advances in Computer Science and Technology, Anaheim, CA, USA, pp. 43–48. ACTA Press (2007)
[18] Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, pp. 330–341 (2004)
[19] Necula, G.C.: Translation validation for an optimizing compiler. SIGPLAN Not. 35(5), 83–94 (2000)
[20] Shashidhar, K.C.: Efficient automatic verification of loop and data-flow transformations by functional equivalence checking. Ph.D thesis, Katholieke Universiteit Leuven, Belgium (May 2008)
[21] Shashidhar, K.C., Bruynooghe, M., Catthoor, F., Janssens, G.: Verification of source code transformations by program equivalence checking. In: Bodik, R. (ed.) CC 2005. LNCS, vol. 3443, pp. 221–236. Springer, Heidelberg (2005)