

Equivalence Checking of Static Affine Programs using Widening to Handle Recurrences

Sven Verdoolaege

Gerda Janssens

Maurice Bruynooghe

Report CW 565, September 2009



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Equivalence Checking of Static Affine Programs using Widening to Handle Recurrences *

Sven Verdoolaege

Gerda Janssens

Maurice Bruynooghe

Report CW 565, September 2009

Department of Computer Science, K.U.Leuven

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 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.

Keywords : polytope model; equivalence checking; recurrences; commutativity; widening

*This research was supported by FWO-Vlaanderen, project G.0232.06N.

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 accesses.

A major challenge in the equivalence checking of static affine programs is posed by recurrences, i.e., a statement in a loop that (indirectly) depends on previous iterations of the same statement. The handling of recurrences requires the construction of an induction hypothesis that is adequate to carry the verification condition across the recurrences. In previous work of Barthou et al. (2002); Alias and Barthou (2003); Shashidhar et al. (2005) and Shashidhar (2008), the transitive closure operation (Kelly et al., 1996b) provided by the *Omega* library (Kelly et al., 1996a) 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. However, this proposal has not been implemented (Shashidhar, 2008, Section 9.3.1). 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.

Furthermore, all the above approaches require both programs to be in dynamic single assignment (DSA) form Feautrier (1988a), 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, (partial) 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 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.

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. We introduce a set of rewrite rules that manipulate an equivalence tree whose root eventually concludes whether equivalence between both programs has been proven, we summarize the method in a pair of algorithms and we prove its termination and soundness. In Section 5 and Section 6 we discuss respectively some implementation details and some experiments. Finally, in Section 7, we discuss in more depth related work and we conclude in Section 8. This paper is a revision and extension of our earlier work (Verdoolaege et al., 2009a). In particular, the equivalence checking procedure has been formalized as a set of tree rewriting rules and a pair of algorithms with proof of termination and soundness and it has been validated on a larger set of examples.

```

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

```

Figure 1: Two equivalent programs, assuming that + is commutative.

2 Illustrative Example

Fig. 1 shows an 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` 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 will assume no a priori relation between temporary arrays from different programs. Likewise, we will assume no a priori relation between any pair of loops in the two programs.

The example in Fig. 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 will 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 + 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 will 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 will not

be able to prove that it is equal to the value computed by the (different) function g in Line 7 of Program 2. Similarly, we will 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 will lead us to the input again and is easily seen to hold. The proof obligation through g , however, will lead 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 will *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 will proceed 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 will apply *induction* and assume that equivalence has already been proven for this pair of computations. This assumption will be validated at a later stage, 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 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 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 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 will frequently make use of subsets of \mathbb{Z}^d , with d some non-negative integer, and of relations over pairs of such sets. The integer points in these sets and relations will typically represent iterations of a loop or elements of an array. We require that all sets and relations can be described using affine (linear + constant) constraints. This requirement ensures that all operations we need to perform on our sets and relations can be performed reasonably efficiently.

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 Fig. 1 contain a parameter N that determines the size of the 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 variables. For example, the set of iterations (i.e., values of the i -iterator) for which the statement in Line 8 is executed, can be

represented as

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

3.2 Input Programs

Our equivalence checker takes two static affine programs 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 8 of Fig. 1(b). Since the constraints that describe an iteration domain are derived from the loops bounds and the conditions in the program, all these constructs need to be quasi-affine.

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 has access relation

$$\{ (i) \rightarrow (i - 1) \}.$$

Similarly, the access to the A array in Line 12 has access relation

$$\{ () \rightarrow (N - 1) \}.$$

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 “()”. 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 the 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 will try 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. If they are not, then we take the output arrays to be those arrays with 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. All other arrays are treated as temporary arrays. If an element of an array is read before or without being written, then it is taken as an element from an input array with the same name. This allows us to split an array that is initially used as an input array, but later reused to store intermediate results, into two arrays, one input array and one temporary (or output) array.

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.

In future work, we plan to extend our work to use *fuzzy* dataflow analysis (Barthou et al., 1997) instead.

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 Fig. 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

$$\{ () \rightarrow () \mid N = 1 \}. \quad (2)$$

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

$$\{ () \rightarrow (N - 1) \mid N \geq 2 \}. \quad (3)$$

3.4 Dependence Graphs

The equivalence checking will be performed on abstractions of the input programs called *dependence graphs*, which represent the results obtained from dataflow analysis. 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. We first present the definition of a dependence graph and then explain how such a dependence graph can be constructed from dataflow analysis.

Definition 1 (Dependence Graph) *A dependence graph is a connected directed graph $G = \langle V, E \rangle$ 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 $\langle d_v, D_v, f_v, r_v, l_v \rangle$, with*

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

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

- p_e a positive 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.

1. The domains of the dependence relations of the edges emanating from a vertex for a given argument position partition the domain of the vertex, i.e.,

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

The unique edge corresponding to a point \mathbf{x} and an argument position p is denoted $e_G(\mathbf{x}, p)$.

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, all iteration domains are finite.

The vertices in the dependence graph 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.

If there is more than one output array, then in principle it would be possible to construct separate dependence graphs for each output array and perform an equivalence check on each pair of dependence graphs separately. However, this could result in duplicate work. Instead, in our implementation, we create a virtual output computation that reads all elements from all output arrays. From here on we will assume that there is a single output computation.

The dimension of the output computation v_0 is equal to the dimension of the output array. The “iteration domain” of v_0 is equal to the set of indices of the elements of the array. The operation is set to the name of the output array and the arity is one. The location of the output computation is undefined. Input computations are very similar. The main difference is that they have arity zero. There is one special input computation \mathbb{Z} that corresponds to the set of integers. It is a one-dimensional computation with the whole of \mathbb{Z} as its iteration domain. The value produced by a given iteration of this virtual computation is equal to the iteration itself. The \mathbb{Z} -computation is useful for modeling quasi-affine expressions of iterators and parameters that occur outside of index expressions.

The computations in the dependence graph that are not input or output computations, each represent an operation inside a statement or a copy statement, i.e., a statement that merely copies a value from one array element to another. The dimension of such a computation is equal to the number of loops enclosing the statement. The iteration domain of the computation is equal to that of the statement in which the operation appears. If any of the enclosing loops does not have an upper bound, we can bound it by a fresh parameter, ensuring the third constraint is satisfied. The operation of the computation is equal to the operation performed or the special “id” for computations derived from copy statements. The arity is set to the number of arguments of the operation (1 for copy statements). The location of the computation identifies the location of the operation inside the program text.

Example 2 *The dependence graph of the program in Fig. 1(a) is shown in Fig. 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. The input computation is one-dimensional because `In` is a one-dimensional array. The output computation is zero-dimensional because `Out` is a scalar. The edges in the graph are explained in Example 3. Fig. 3 similarly shows the dependence graph for the program in Fig. 1(b).*

The edges in the dependence graph indicate which iterations of which computations require values computed in which iterations of which (other) computations. Edges arise in three different ways,

1. from a computation with operation f to a computation with an operation that appears as one of the arguments of f in the same statement of the program text
2. from a computation that reads a value from an array to the computation that wrote the value to the array
3. from the output computation to a computation that last wrote an element of the output array.

The first kind is the simplest. Both computations involved, say u and v , originate from the same statement and therefore have the same iteration domain $D_u = D_v$. The argument position $p_{(u,v)}$

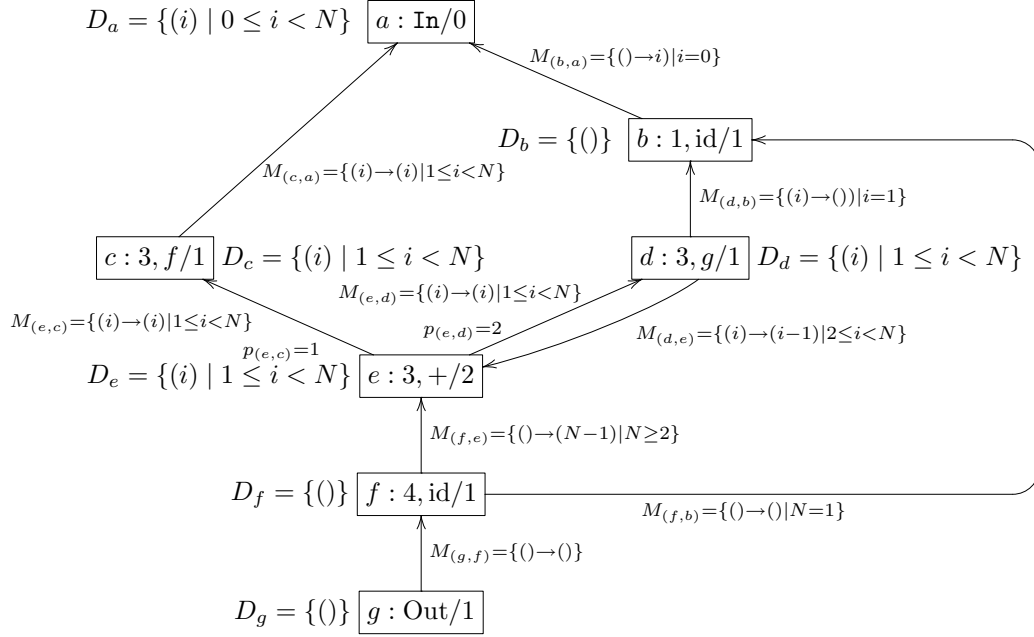


Figure 2: Dependence graph G_1 of the program in Fig. 1(a). Computations are named from a to g with a the input and g the output computation. Each computation v is represented as “ $v : l_v, f_v/r_v$ ” (l_v is absent for input and output computations). To avoid clutter, the dimension is omitted, while the domain is shown next to the box with the node; the argument position p_e is only indicated on edges emanating from node e (it is 1 on all other edges).

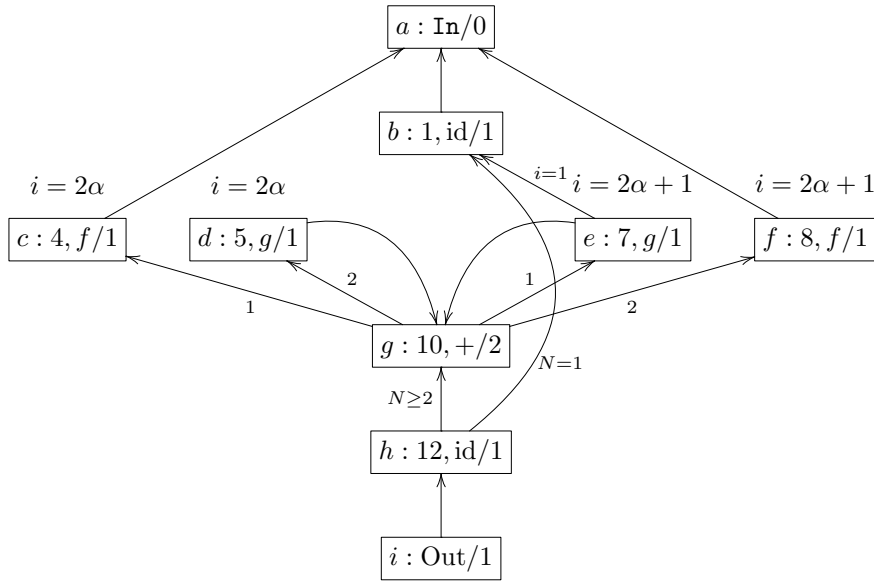


Figure 3: Dependence graph G_2 of the program in Fig. 1(b) (some details omitted).

is equal to the argument position of f_u in which f_v appears. The dependence relation associated to the edge is the identity relation on D_u , $M_{(u,v)} = \{(\mathbf{i}) \rightarrow (\mathbf{i}) \mid \mathbf{i} \in D_u\}$. For the second kind of edges, the argument position is that in which the array access appears. The dependence relation is the one computed by dataflow analysis. For the third kind, the argument position is always 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 dependence graph that results from the above construction also satisfies the remaining two constraints of Definition 1. To see that the first constraint is satisfied, first note that the edges that emanate from a given computation with a given argument position are all of the same kind. Either the argument is another operation, in which case there is a single intra-statement dependence edge leaving the computation for that argument and the constraint is trivially satisfied, or the argument is a read from an array, in which case dependence analysis ensures that the constraint is satisfied. The edges leaving the output computation are all of the third kind and the constraint is again satisfied because each element of the output array is written for the last time exactly once. The second constraint is satisfied because a statement iteration can only depend on a *previous* iteration in the execution order of the input program.

It should be noted that the effect of a dependence relation $M_{(u,v)}$ associated to an edge (u, v) is two-fold. The dependence relation first selects part of the iteration domain D_u , in particular, $D_u \cap \text{dom } M_{(u,v)}$, and then maps those elements from the D_u space to the D_v space.

Example 3 Consider once more the dependence graph in Fig. 2 of the program in Fig. 1(a). The computations in this graph have already been discussed in Example 2. Examples of intra-statement dependences in Fig. 2 are the edges from the addition (computation e) to respectively the computations c and d . As to edges that result from dataflow analysis, the dependence relations $M_{(f,b)}$ and $M_{(f,e)}$ are those from Equations (2) and (3) in Section 3.3, respectively. The output array `Out` is only written in Line 4, with access relation $\{() \rightarrow ()\}$. The dependence relation on the edge (g, f) is therefore the inverse of this access relation.

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. 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,
- $(v, w) \in E$, and,
- 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 such pair can be found, then we are done. Otherwise, part of our dependence graph looks like the fragment shown in Fig. 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

$$\begin{aligned} D_{v_1} &= D_v \setminus \text{dom } M_{v,w} \\ D_{v_2} &= D_v \cap \text{dom } M_{v,w}. \end{aligned}$$

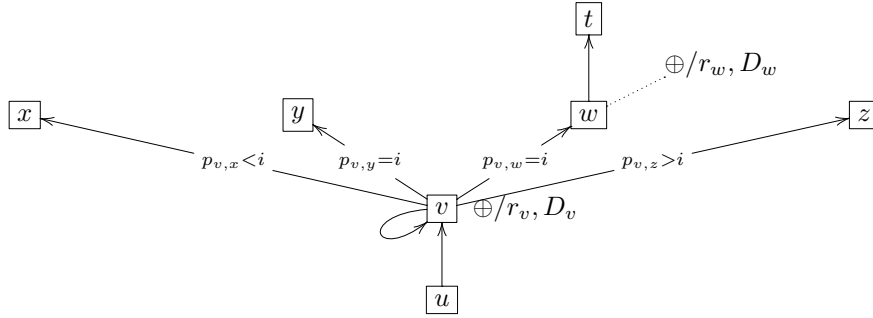


Figure 4: Part of a dependence graph before applying the associativity transformation.

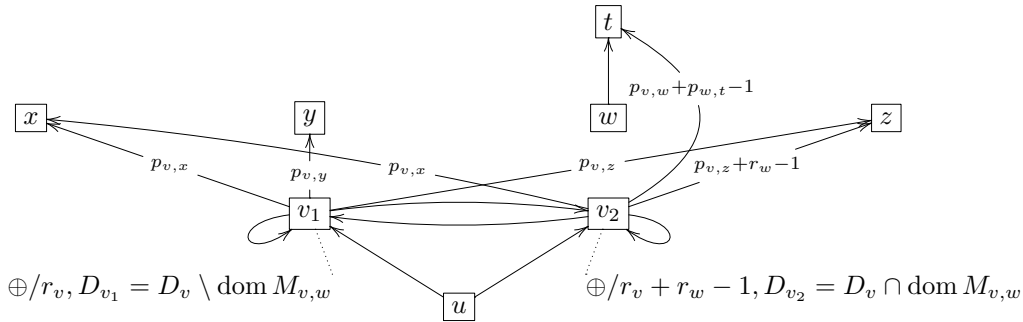


Figure 5: The same dependence graph fragment of Fig. 4, but after applying the associativity transformation.

Now, v_2 can safely be replaced by the combination of the 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

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

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

$$\begin{aligned} M_{(u,v_1)} &= M_{(u,v)} \cap \mathbb{Z}^{d_u} \times D_{v_1} \\ M_{(u,v_2)} &= M_{(u,v)} \cap \mathbb{Z}^{d_u} \times D_{v_2}. \end{aligned}$$

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

$$\begin{aligned} M_{(v_1,s)} &= M_{(v,s)} \cap D_{v_2} \times \mathbb{Z}^{d_s} \\ M_{(v_2,s)} &= M_{(v,s)} \cap D_{v_1} \times \mathbb{Z}^{d_s}, \end{aligned}$$

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

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

```

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

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

```

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

Note that the fact that we do not flatten recurrences means that we cannot detect the equivalence of the two programs in Fig. 6 for computing $\sum_{0 \leq i < N} a[i]$. Shashidhar (2008) would completely unroll 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 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 4 (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 $\mathbf{x}_1 = \mathbf{x}_2$,
- $f_{v_1} = \text{id}$ and iteration $M_{e_{G_1}(\mathbf{x}_1,1)}(\mathbf{x}_1)$ of v_1' with $e_{G_1(\mathbf{x}_1,1)} = (v_1, v_1')$ is equivalent to iteration \mathbf{x}_2 of v_2 ,
- $f_{v_2} = \text{id}$ and iteration $M_{e_{G_2}(\mathbf{x}_2,1)}(\mathbf{x}_2)$ of v_2' with $e_{G_2(\mathbf{x}_2,1)} = (v_2, v_2')$ 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 $M_{e_{G_1}(\mathbf{x}_1,p)}(\mathbf{x}_1)$ of v_1' with $e_{G_1(\mathbf{x}_1,p)} = (v_1, v_1')$ is equivalent to iteration $M_{e_{G_2}(\mathbf{x}_2,p)}(\mathbf{x}_2)$ of v_2' with $e_{G_2(\mathbf{x}_2,p)} = (v_2, v_2')$, 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 $M_{e_{G_1}(\mathbf{x}_1,p)}(\mathbf{x}_1)$ of v_1' with $e_{G_1(\mathbf{x}_1,p)} = (v_1, v_1')$ is equivalent to iteration $M_{e_{G_2}(\mathbf{x}_2,\pi(p))}(\mathbf{x}_2)$ of v_2' with $e_{G_2(\mathbf{x}_2,\pi(p))} = (v_2, v_2')$,

Definition 5 (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.*

4 Equivalence Checking

In order to prove equivalence of two dependence graphs we basically follow Definition 4 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 of Shashidhar et al. (2005); Shashidhar (2008); Barthou et al. (2002) and Alias and Barthou (2003), who 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 recurrences and commutativity. We first describe the equivalence tree. The main steps in our algorithm are then described as rewriting rules on this equivalence tree. We conclude with an overview of the complete algorithm and a correctness proof.

4.1 The Equivalence Tree

The propagation from output to input constructs an equivalence tree. Each node n in the equivalence tree expresses 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 proven to be equivalent (as in Definition 4) to certain iterations of $v_{n,2}$. The 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, $R_n^{\text{want}} \subseteq D_{v_{n,1}} \times D_{v_{n,2}}$. The above annotations of an equivalence node remain unaltered during the execution of the equivalence checker. The other annotations, described below, may change when the equivalence node is processed. To describe the steps in our algorithm as tree rewriting rules, we therefore need to keep track of the “state” of a node. When a new equivalence node is created, it is in the “init” state. While the node is being processed it is in the “open” state, after which it will move to the “semi-closed” state. The node may then be “reopened” and after all processing it moves to the “closed” state.

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 proven 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. (2009a) 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. (2009b). The R^{lost} relation is initially undefined and is (re)set when the equivalence node moves to the “semi-closed” or “closed” state.

The initial equivalence tree consists of a single root node n_0 that models the equivalence to be proven between the output arrays of both programs. For this root, we have $R_{n_0}^{\text{want}} = \{(\mathbf{i}) \leftrightarrow (\mathbf{i}) \mid \mathbf{i} \in D\}$ 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. 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 6 *Fig. 7 shows a partial equivalence tree for the dependence graphs in Fig. 2 and Fig. 3, which correspond to the programs in Fig. 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 a of the tree expresses the desired equivalence of the output arrays. Since the output arrays are scalars in this example, we have $R_a^{\text{want}} = \{() \leftrightarrow ()\}$.*

4.2 Trivial Cases

We first discuss a couple of trivial equivalence tree rewriting rules, shown in Fig. 8. If there is a node n in the init state with an empty R_n^{want} , then there is nothing to prove and then also no correspondence that cannot be proven. The node can therefore be closed with empty R_n^{lost} . The opposite conclusion holds if the computations of the node have different operations, neither of which is the copy operation id . According to Definition 4, iterations of 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 4, we can only prove identical iterations (array indices) to be equivalent.

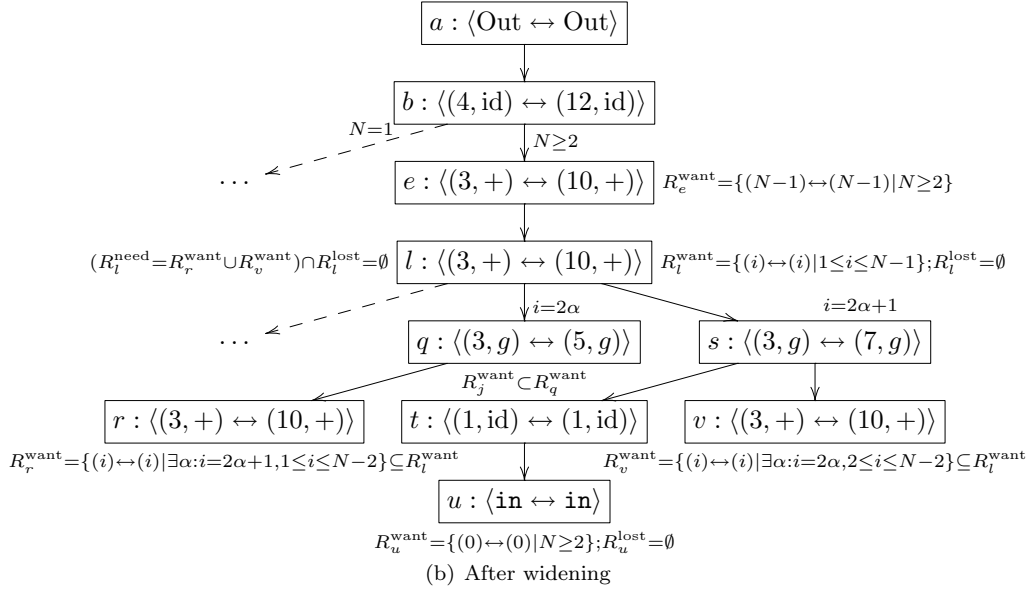
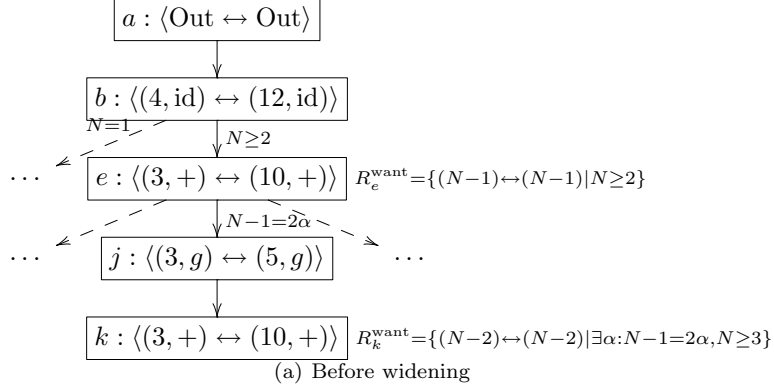


Figure 7: Partial equivalence tree for the dependence graphs in Fig. 2 and Fig. 3.

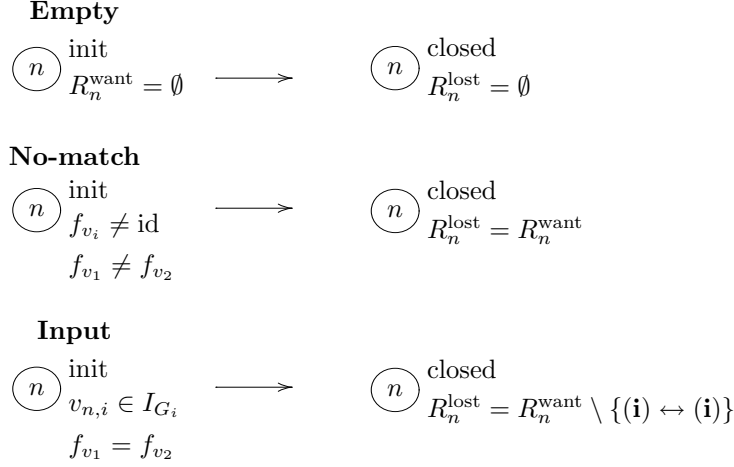


Figure 8: Trivial tree rewriting rules. In any tree rewriting rule, the left hand side describes the conditions that need to hold for the rule to apply. The right hand side describes the effect of the rule, including a change of state, an addition or update of an annotation, the creation of child nodes or the removal of child nodes. Any annotation not specified on the right hand side is left untouched.

Example 7 To avoid clutter in the equivalence tree in Fig. 7, we do not show any node with an empty R_n^{want} or with conflicting operations. Fig. 7(b) does show one node, u , where the input rule applies. R_u^{want} is a subset of the identity mapping and so we are able to prove it completely.

4.3 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 corresponding tree rewriting rules are shown in Fig. 9. For now, the reader may assume that the “successful induction” condition always applies and that therefore the “semi-closed” state is immediately rewritten into a “closed” state. The induction condition 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.,

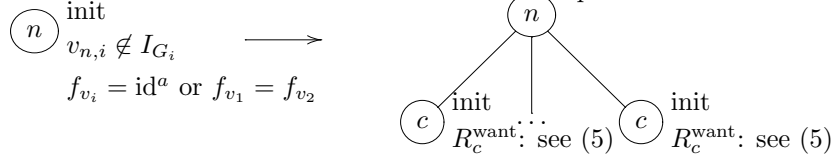
$$T_{n,p_1,p_2} = \{ (e_1, e_2) \in E_{G_1} \times E_{G_2} \mid \exists (u_1, u_2) \in V_{G_1} \times V_{G_2} : e_1 = (v_{n,1}, u_1), e_2 = (v_{n,2}, u_2), p_{e_1} = p_1, p_{e_2} = p_2 \},$$

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

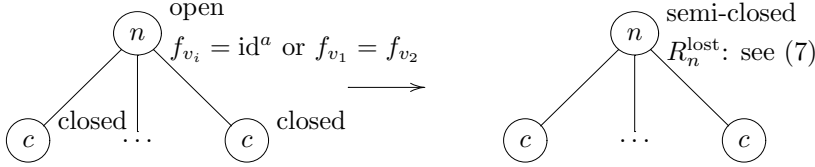
$$T_n = \bigcup_{1 \leq p \leq r_{v_{n,1}}} T_{n,p,p}, \quad (4)$$

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

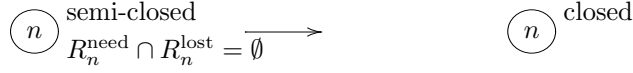
Forward (Copy) Propagation



Backpropagation



Successful Induction



^a*i* is the smallest index satisfying this constraint

Figure 9: Propagation tree rewriting rules. The Forward (Copy) Propagation rule only applies if neither Induction or Widening from Fig. 10 apply.

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 mappings of both edges, i.e.,

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

where the \oplus operator combines two mappings of type $D_{v_1} \rightarrow D_{u_1}$ and $D_{v_2} \rightarrow D_{u_2}$ into one of type $D_{v_1} \times D_{v_2} \rightarrow D_{u_1} \times D_{u_2}$:

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

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.

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

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

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. 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 8 Consider once more the partial equivalence tree in Fig. 7(a). The root node a expresses a correspondence between output computation g in dependence graph G_1 (Fig. 2) and output computation i in dependence graph G_2 (Fig. 3). 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 $\langle (4, \text{id}/1) \leftrightarrow (12, \text{id}/1) \rangle$ with $R_b^{\text{want}} = R_a^{\text{want}} = \{() \leftrightarrow ()\}$, 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, $\langle (3, +/2) \leftrightarrow (10, +/2) \rangle$, has $R^{\text{want}} = \{(N-1) \leftrightarrow (N-1) \mid N \geq 2\}$ 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 $N \geq 2$).

4.4 Commutative Operations

Propagation over commutative operations requires a bit more work. According to Definition 4, 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 and T_n^π the sets of pairs of edges with arguments permuted according to π , i.e.,

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

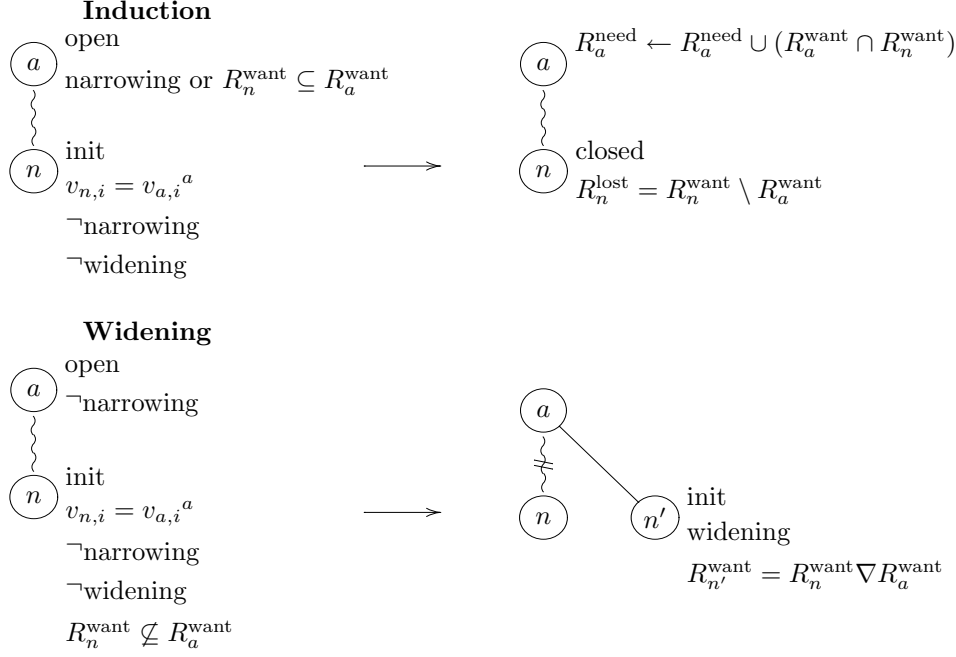
Obviously, we cannot just apply (6) during backpropagation because a given pair of iterations will in most cases only be proven equivalent for one particular permutation of the arguments. The pairs of iterations that we cannot prove equivalent are then those that cannot be proven equivalent using any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

$$R_n^{\text{lost}} = \bigcap_{\pi \in \Pi} \left(\bigcup_{(e_1, e_2) \in T_n^\pi} (M_{e_1}^{-1} \oplus M_{e_2}^{-1}) R_{c_{e_1, e_2}}^{\text{lost}} \right) \cap R_n^{\text{want}}. \quad (7)$$

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.

It should be noted that our approach is 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 examples 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.

Example 9 In our running example, the $+/2$ computation of Program 1 (node e of Fig. 2) has one outgoing edge for each argument, while node g in Fig. 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



^a a is the closest ancestor satisfying this constraint

Figure 10: Widening tree rewriting rules. A squiggly line (\sim) denotes a path in the tree and a crossed squiggly line (\sim/\sim) emanating from a node means that the tree rooted at that node has been removed.

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 e in Fig. 7. Once R^{lost} is available in each of the children, backpropagation can update R^{lost} in this node e .

4.5 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.

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^{need} . The induction rule is shown in Fig. 10 in a slightly more general form. Since $R_n^{\text{want}} \subseteq R_a^{\text{want}}$ in the case we are discussing here, we have $R_n^{\text{lost}} = R_n^{\text{want}} \setminus R_a^{\text{want}} = \emptyset$. The conditions \neg narrowing and \neg widening on node n ensure that n is a genuine new node resulting from a Forward 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 by the “successful induction” rule in Fig. 9. 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 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.

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 $D_{v_{n,1}} \times D_{v_{n,2}}$ 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 . The corresponding tree rewriting rule is shown in Fig. 10. Backpropagation uses the normal rule of Fig. 9. 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 $D_1 \times D_2$ 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 $D_1 \times D_2$) and the widening operator will then increase the dimension of the affine subspace. So, after a finite number of widening steps, $R^{\text{want}} = D_1 \times D_2$, 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.

Finally, we need to consider what happens when it turns out we have been over-optimistic in our induction hypothesis, i.e., when $R_n^{\text{need}} \cap R_n^{\text{lost}} \neq \emptyset$. 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 proven, which is definitely the case when it becomes empty. Our narrowing operator is fairly simple. The first time it is applied, we replace the tree rooted at n by a new child node n' and set $R_{n'}^{\text{want}} = R_n^{\text{want}} \setminus R_n^{\text{lost}}$. Then, in any descendant d with the same pair of computations, we unconditionally apply induction, setting $R_d^{\text{lost}} = R_d^{\text{want}} \setminus R_{n'}^{\text{want}}$ (see Fig. 10). 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 the narrowing node becomes semi-closed and the resulting R^{lost} still intersects with its R^{need} , then we perform a second and final narrowing step and reduce R^{want} to the empty set. We need not create a child this time, we simply set $R_n^{\text{lost}} = R_n^{\text{want}}$. The corresponding tree rewriting rules are shown in Fig. 11. Note that we cannot use the Backpropagation rule of Fig. 9 to compute R_n^{lost} . Instead, we have to use the Finish Narrowing rule as the elements of R_n^{want} not passed to node n' have to be preserved in the final R_n^{lost} . This is also the motivation for introducing the state “reopened”.

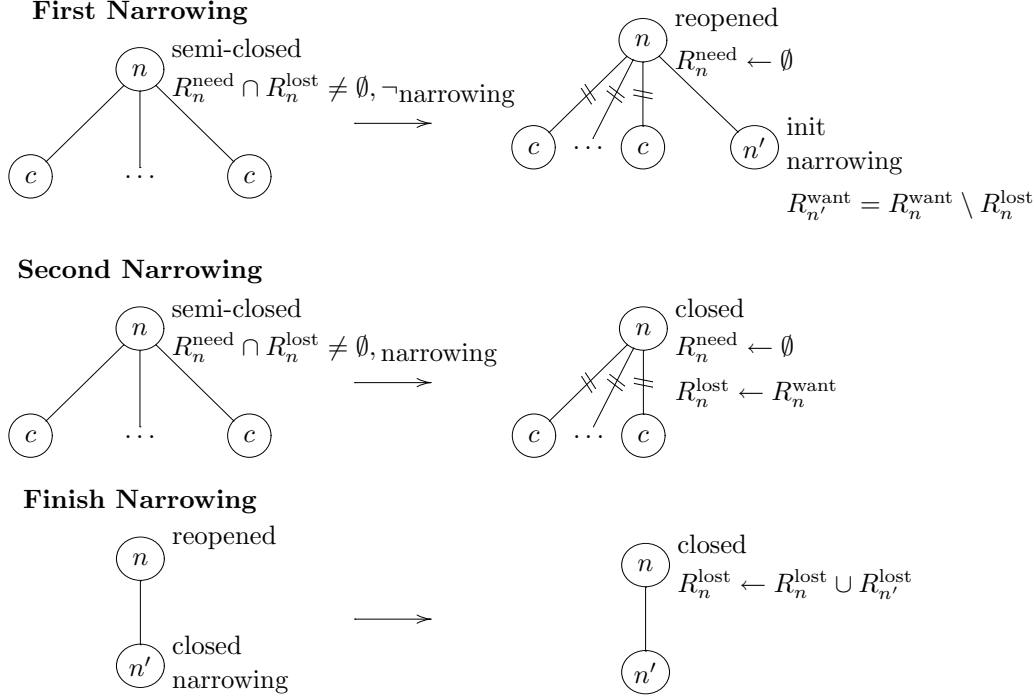


Figure 11: Narrowing tree rewriting rules.

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 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) 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 $B[i]$.

Example 10 After applying propagation to node j of the equivalence tree in Fig. 7(a), we end up in node k which refers to the same pair of computations as node e . We have $R_k^{\text{want}} = \{(N-2) \leftrightarrow (N-2) \mid \exists \alpha : N-1 = 2\alpha, N \geq 3\}$, while $R_e^{\text{want}} = \{(N-1) \leftrightarrow (N-1) \mid N \geq 2\}$, i.e., $R_k^{\text{want}} \not\subseteq R_e^{\text{want}}$. We therefore apply widening and obtain $R_l^{\text{want}} = R_e^{\text{want}} \nabla R_k^{\text{want}} = \{(i) \leftrightarrow (i) \mid 1 \leq i \leq N-1\}$. The new node l replaces the entire tree rooted at e , as shown in Fig. 7(b). We now have $R_l^{\text{want}} \not\subseteq R_e^{\text{want}}$, but l is marked as widening and so the widening rule no longer applies and we perform propagation on l in much the same way as we did on e in Example 9. Another propagation step on the resulting node q yields node r , with the same pair of computations as node l . Since $R_r^{\text{want}} \subseteq R_l^{\text{want}}$, the induction rule applies, R_r^{lost} is set to \emptyset and R_l^{need} is updated. A very similar story happens in node v . After propagating the results back to node l , we see that $R_l^{\text{lost}} = \emptyset$, meaning that all the induction hypotheses have been validated and there is no need for a narrowing phase.

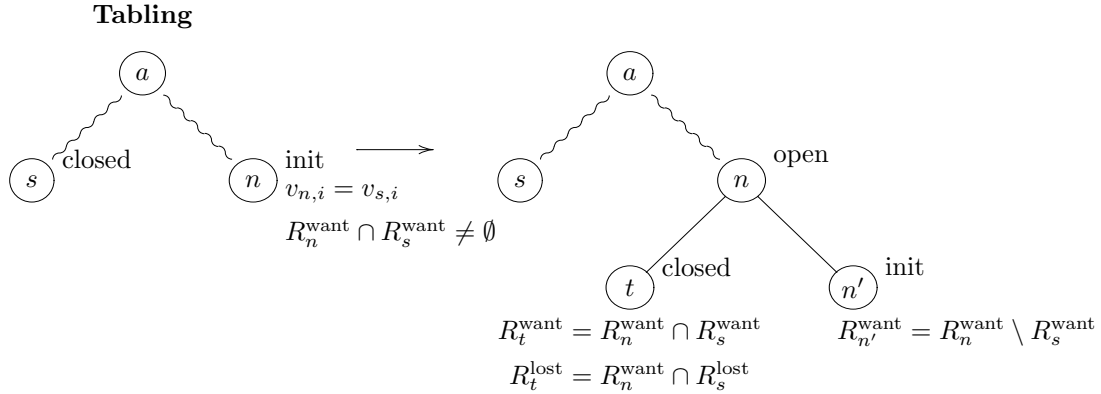


Figure 12: Tabling tree rewriting rule.

4.6 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) proven 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 (closed) 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. These results are stored in a child node t of n . Another child n' is created for handling the possibly empty remainder of R_n^{want} , i.e., $R_n^{\text{want}} \setminus R_s^{\text{want}}$. The corresponding tree rewriting rule is shown in Fig. 12.

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.

4.7 Algorithms

An algorithm for depth-first traversal of the equivalence tree is shown in Algorithm 1 and Algorithm 2. Algorithm 2 encodes the propagation (Fig. 9) and narrowing (Fig. 11) rules, while Algorithm 1 encodes all the other rules. The comments on the right contain the names of the rules that are being applied. The “Forward (Copy) Propagation” is split into a “Copy” and a “Propagation” block. Since the state of a node is implicitly defined by the algorithm, it is not recorded explicitly. The “Successful Induction” rule is then also implicitly applied when no narrowing is performed. Since the need for a widening is detected at the level of a descendant, while the actual widening happens at the level of the ancestor, the “Widening” rule is split into two parts. Communication between the two part occurs through an exception that is thrown when the need for widening is detected. The node t of the “Tabling” rule is not explicitly created.

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

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

```
1 Initialize  $R_n^{\text{need}} = \emptyset$ 
2 try
3   if  $R_n^{\text{want}} = \emptyset$  then                                /* Empty */
4     | Set  $R_n^{\text{lost}} = \emptyset$ 
5   else if both computations refer to the same input then  /* Input */
6     | Set  $R_n^{\text{lost}} = R_n^{\text{want}} \setminus \{(\mathbf{i}) \leftrightarrow (\mathbf{i})\}$ 
7   else if there is a non-ancestor node  $s$  with  $s_{v_i} = n_{v_i}$  and  $R_n^{\text{want}} \cap R_s^{\text{want}} \neq \emptyset$  then
      /* Tabling */
8     | Create a new child node  $n'$  with same computations and  $R_{n'}^{\text{want}} = R_n^{\text{want}} \setminus R_s^{\text{want}}$ 
9     | Try and handle  $n'$  (Algorithm 1)
10    | Set  $R_n^{\text{lost}} = (R_s^{\text{lost}} \cap R_n^{\text{want}}) \cup R_{n'}^{\text{lost}}$ 
11  else if  $n$  is not a narrowing or widening node and there is an ancestor node  $a$  with
       $a_{v_i} = n_{v_i}$  then
12    | let  $a$  be the closest ancestor of  $n$  with  $a_{v_i} = n_{v_i}$  in
13      | if  $a$  is a narrowing node or  $R_n^{\text{want}} \subseteq R_a^{\text{want}}$  then          /* Induction */
14        | Set  $R_n^{\text{lost}} = R_n^{\text{want}} \setminus R_a^{\text{want}}$ 
15        | Add  $R_a^{\text{want}} \cap R_n^{\text{want}}$  to  $R_a^{\text{need}}$ 
16      | else
17        | /*  $a$  is not a narrowing node and  $R_n^{\text{want}} \not\subseteq R_a^{\text{want}}$           /* Widening */
18        | throw  $\text{widen}(a, R_n^{\text{want}})$                                      */
19  else
20  | Handle propagation node  $n$  (Algorithm 2)
21 catch  $\text{widen}(n, R^{\text{want}})$                                            /* Widening */
22  | Remove all children of  $n$ 
23  | Create a new child node  $n'$  with same computations and  $R_{n'}^{\text{want}} = R_n^{\text{want}} \nabla R^{\text{want}}$ 
24  | Mark  $n'$  as widening
25  | Try and handle  $n'$  (Algorithm 1)
26  | Set  $R_n^{\text{lost}} = R_{n'}^{\text{lost}} \cap R_n^{\text{want}}$                                /* Backpropagation */
```

Algorithm 2: Handle propagation node n

Input: dependence graphs $G_i = \langle V_i, E_i \rangle$, $i = 1, 2$; node n

Output: relation R_n^{lost}

Modifies: node n

Throws: $\text{widen}(a, R^{\text{want}})$, 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
6   |   
$$\begin{cases} R_n^{\text{lost}} = \bigcap_j (M_{e_j}^{-1} \oplus 1) R_{c_j}^{\text{lost}} & \text{if } i = 1 \\ R_n^{\text{lost}} = \bigcap_j (1 \oplus M_{e_j}^{-1}) R_{c_j}^{\text{lost}} & \text{if } i = 2 \end{cases}$$

7   else if  $f_{v_1} = f_{v_2}$  then                                                    /* Propagation */
8   |   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
9   |    $v_2$ 
10  |   Try and handle child nodes  $c_{(e_1, e_2)}$  (Algorithm 1)
11  |   Set  $R_n^{\text{lost}}$  according to (6) or (7)                                          /* Backpropagation */
12 else
13 |   Set  $R_n^{\text{lost}} = R_n^{\text{want}}$                                                     /* No-match */
14 if  $R_n^{\text{need}} \cap R_n^{\text{lost}} \neq \emptyset$  then
15 |   Remove all children of  $n$ 
16 |   Reset  $R_n^{\text{need}} = \emptyset$ 
17 |   if  $n$  is not narrowing then                                              /* First Narrowing */
18 |   |   Create child node  $n'$  with same computations and  $R_{n'}^{\text{want}} = R_n^{\text{want}} \setminus R_n^{\text{lost}}$ 
19 |   |   Mark  $n'$  as narrowing
20 |   |   Try and handle  $n'$  (Algorithm 1)
21 |   |   Reset  $R_n^{\text{lost}} = R_n^{\text{lost}} \cup R_{n'}^{\text{lost}}$                                 /* Finish Narrowing */
22 |   else                                                                        /* Second Narrowing */
23 |   |   Reset  $R_n^{\text{lost}} = R_n^{\text{want}}$ 
```

4.8 Termination and soundness

We will now proceed to prove termination and soundness of our equivalence checking procedure. 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 rules that cut away parts of the equivalence tree as not actually removing any nodes, but just marking them as being 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 11 *Algorithm 1 terminates. That is, the equivalence tree is finite.*

Proof There are four rules that create new children, at most $|E_1||E_2|$ by Forward Propagation (Fig. 9; Line 3 and Line 7 of Algorithm 2), one by Widening (Fig. 10; Line 22 of Algorithm 1), one by Narrowing (Fig. 11; Line 16 of Algorithm 2) and one by Tabling (Fig. 12; Line 8 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 the Tabling rule. 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 11 of Algorithm 1). These narrowing and widening node 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. \square

Lemma 12 *For any undeleted and closed 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.,*

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

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

Proof First note that this property is stable, in the sense that once proven 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 closed and further steps can only add elements to the R_a^{need} and never remove elements from these relations. The only exception is Line 14 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 the nodes are closed, i.e., a depth-first postordering of the equivalence tree. We consider nine cases.

- Empty (Fig. 8; Line 3 of Algorithm 1)
 $R_n^{\text{want}} \setminus R_n^{\text{lost}} = \emptyset$, so (8) trivially holds.
- Input (Fig. 8; Line 5 of Algorithm 1)
 $R_n^{\text{want}} \setminus R_n^{\text{lost}} = R_n^{\text{want}} \cap \{(\mathbf{i}) \leftrightarrow (\mathbf{i})\}$ holds by Definition 4.

- Tabling (Fig. 12; Line 7 of Algorithm 1)

By induction, we have

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

and

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

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 closed 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 12 of Algorithm 2, or it is made to hold in Line 14.) 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 computation as n , n will never be used as a closest ancestor in the application of an Induction rule, 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 (Fig. 10; Line 13 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 (Fig. 10; Line 25 of Algorithm 1)

Since n' has the same pair of computation as n , n will never be used as a closest ancestor in the application of an Induction rule, 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 (Fig. 11; Line 15 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 (Fig. 11; Line 20 of Algorithm 2)

In this case, $R_n^{\text{want}} \setminus R_n^{\text{lost}} = \emptyset$ and so the lemma trivially holds.

- Propagation (Fig. 9; Line 6 of Algorithm 2)

If the condition in Line 12 holds, then we apply one of the narrowing rules 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 4 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 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 1), this process terminates and we can drop $(\mathbf{i}, \mathbf{i}')$ from the antecedent. 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 (Fig. 9; Line 1 of Algorithm 2)

This case is proved in the same way as Propagation and thereby completes the proof. □

Proposition 13 *The two programs are equivalent if $R_{n_0}^{\text{lost}} = \emptyset$, with n_0 the initial node in the equivalence tree.*

Proof After termination of the algorithm (Lemma 11), the initial node n_0 is closed. Since n_0 is the root node, it has no ancestors, i.e., $\mathcal{A}(n_0) = \emptyset$. By Lemma 12, 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 Section 4 has been implemented as part of our C++ `isa` prototype tool set (<http://www.kotnet.org/~skimo/loop/isa-0.09.tar.bz2>). The tool set contains a polyhedral extractor from C based on SUIF (Amarasinghe et al., 1995) 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., 1996a) 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).

```

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];
}

```

Figure 13: A pair of equivalent programs from Shashidhar et al. (2005).

```

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];
}

```

Figure 14: A pair of non-equivalent programs.

Instead, we use our own thread-safe C library called `isl` (Verdoolaege, 2009), which uses GMP to perform all its integer manipulations in exact integer arithmetic and is available from <http://freshmeat.net/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 parametric integer programming (Feautrier, 1988b), which is used in our dependence analysis tool. 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.

6 Examples and Experiments

Fig. 13 reproduces the motivating example of Shashidhar et al. (2005). The programs are only equivalent if the `+` operator is treated as both associative and commutative. Fig. 14 shows a pair of programs with output arrays `out` that are not equivalent (unless $N = 0$) because `b[0]` is

```

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];

```

Figure 15: A pair of equivalent programs with a non-uniform recurrence.

```

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

```

Figure 16: Partial loop unrolling.

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. Fig. 15 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.

Fig. 16 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 (See Godlin and Strichman, 2008, Section 6). Our proof procedure has no problem proving the equivalence of these two programs. Of course, both programs are also equivalent to the program “ $out = n \geq 0 ? n * (n+1) / 2 : 0;$ ”, 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 1 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 in the next rows 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. The tool of Shashidhar

program 1	program 2	cases	stats	comps	d	da time	ec time	∇	Δ
Program 1	Program 2	1	10	16	1	0.008	0.010	1	0
Fig. 13 left	Fig. 13 right	1	8	18	1	0.003	0.006	0	0
Fig. 14 left	Fig. 14 right	1	10	14	1	0.004	0.006	1	3
Fig. 15 left	Fig. 15 right	1	6	12	1	0.003	0.004	1	0
Fig. 16(a)	Fig. 16(b)	1	6	8	1	0.005	0.006	1	0
USVD 1	USVD 1	1	620	628	2	0.664	2.087	5	0
USVD 2	USVD 2	1	134	146	3	0.355	0.116	10	0
USVD 1	USVD 2	1	377	387	3	0.509	0.409	5	0
CLooG-isl 1	CLooG-PL 1	1	133	135	3	0.237	0.271	51	0
CLooG-isl 2	CLooG-PL 2	1	1090	1092	3	9.746	6.212	116	0
CLooG-isl 3	CLooG-PL 3	1	26	28	5	0.378	0.294	24	0
CLooG-isl 4	CLooG-PL 4	1	48	52	2	6.908	27.778	233	0
CLooG-isl	CLooG-PL	105	4337	4629	5	40.660	51.444	1741	0

Table 1: 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.

(2008) was developed outside of our university and we were unable to obtain a working copy of the tool for performing a comparative experiment.

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 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. All 105 tests were proven to be equivalent. The number of statements in these tests ranges from 4 to 1090 with running times up to 28 seconds (all but six are well below 1 second) and 51 seconds in total. The number of widening steps performed ranges from 0 to 237, with a grand total of 1741 widening steps. The final row of the table summarizes these results, while the preceding rows show details of some individual test cases: `reservoir/QR`, `swim`, `thomasset` and `reservoir/liu-zhuge1`.

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 the number of statements or the dimension of the iteration domains. Also note that the only pair of programs in Table 1 that requires any narrowing is the pair that is 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 Fig. 15. We are investigating if delaying the widening by one step or the use of more advanced widening or narrowing operators can solve these problems.

7 Related Work

Many approaches have been proposed for checking the equivalence of two programs, but few of these approaches handle 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. 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 proven 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 Fig. 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, 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 Fig. 16. 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., 2009b). 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 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., 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, we want to extend our approach 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 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.

References

- Absar, M. J., Marchal, P., Catthoor, F., 2005. Data-access optimization of embedded systems through selective inlining transformation. In: Miranda, M., Ha, S. (Eds.), Proceedings of the 2005 3rd Workshop on Embedded Systems for Real-Time Multimedia, ESTImedia 2005, September 22-23, 2005, New York Metropolitan Area, USA. IEEE Computer Society, pp. 75–80.
- Alias, C., Barthou, D., Apr. 2003. On the recognition of algorithm templates. In: Int. Workshop on Compilers Optimization Meets Compiler Verification. Vol. 82 of ENTCS. Elsevier Science, Warsaw, pp. 395–409.
- Amarasinghe, S., Anderson, J., Lam, M. S., Tseng, C.-W., 1995. An overview of the SUIF compiler for scalable parallel machines. In: Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing. San Francisco, CA.
- Barrett, C., Tinelli, C., Jul. 2007. CVC3. In: Damm, W., Hermanns, H. (Eds.), Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07). Vol. 4590 of Lecture Notes in Computer Science. Springer-Verlag, pp. 298–302, berlin, Germany.
- Barthou, D., Collard, J.-F., Feautrier, P., 1997. Fuzzy array dataflow analysis. *J. Parallel Distrib. Comput.* 40 (2), 210–226.
- Barthou, D., Feautrier, P., Redon, X., Aug. 2002. On the equivalence of two systems of affine recurrence equations. In: Euro-Par Conference. Vol. 2400 of Lect. Notes in Computer Science. Springer-Verlag, Paderborn, pp. 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, pp. 7–16.
- Catthoor, F., Danckaert, K., Kulkarni, C., Brockmeyer, E., Kjeldsberg, P., Van Achteren, T., Omnes, T., 2002. Data access and storage management for embedded programmable processors. Kluwer Academic Publishers, Boston, USA.
- Chen, C., 2009. Omega+ library.
URL <http://www.cs.utah.edu/~chunchen/omega/>
- Cook, W., Rutherford, T., Scarf, H. E., Shallcross, D. F., Aug. 1991. An implementation of the generalized basis reduction algorithm for integer programming. Cowles Foundation Discussion Papers 990, Cowles Foundation, Yale University.
- Cousot, P., Cousot, R., 1992. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (Eds.), Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming. LNCS 631, Springer-Verlag, Leuven, Belgium, pp. 269–295.
- Feautrier, P., 1988a. Array expansion. In: ICS '88: Proceedings of the 2nd international conference on Supercomputing. ACM Press, pp. 429–441.
- Feautrier, P., 1988b. Parametric integer programming. *Operationnelle/Operations Research* 22 (3), 243–268.
- Feautrier, P., 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20 (1), 23–53.
- Franke, B., O'Boyle, M., May 2003. Array recovery and high-level transformations for DSP applications. *ACM Transactions on Embedded Computing Systems* 2 (2), 132–162.

- Fu, Q., Bruynooghe, M., Janssens, G., Catthoor, F., 2006. Requirements for constraint solvers in verification of data-intensive embedded system software. In: Blanc, B., Gotlieb, A., Michel, C. (Eds.), *Proceedings of the 1st Workshop on constraints in Software Testing, Verification and Analysis*. pp. 46–57.
- Godlin, B., Strichman, O., 2008. Inference rules for proving the equivalence of recursive procedures. *Acta Inf.* 45 (6), 403–439.
- Godlin, B., Strichman, O., Jul. 2009. Regression verification. In: *46th Design Automation Conference (DAC'09)*. pp. 466–471.
- Karr, M., 1976. Affine relationships among variables of a program. *Acta Informatica* 6, 133–151.
- Kaufmann, M., Moore, J. S., Manolios, P., 2000. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA.
- Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D., Nov. 1996a. The Omega library. Tech. rep., University of Maryland.
- Kelly, W., Pugh, W., Rosser, E., Shpeisman, T., 1996b. Transitive closure of infinite graphs and its applications. *Int. J. Parallel Program.* 24 (6), 579–598.
- Manjunathaiah, M., Megson, G. M., Rajopadhye, S. V., Risset, T., 2001. Uniformization of affine dependance programs for parallel embedded system design. In: Ni, L. M., Valero, M. (Eds.), *ICPP 2002, Proceedings*. IEEE Computer Society, pp. 205–213.
- Mateev, N., Menon, V., Pingali, K., 2001. Fractal symbolic analysis. In: *ICS '01: Proceedings of the 15th international conference on Supercomputing*. ACM, New York, NY, USA, pp. 38–49.
- Matsumoto, T., Seto, K., 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, pp. 43–48.
- Müller-Olm, M., 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*. pp. 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., Janssens, G., 2005. Verification of source code transformations by program equivalence checking. In: *CC 2005, Proceedings*. Vol. 3443 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, pp. 221–236.
- van Engelen, R. A., 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, pp. 80–89.
- Verdoolaege, S., Apr. 2009. An integer set library for program analysis, advances in the Theory of Integer Linear Optimization and its Extensions, AMS 2009 Spring Western Section Meeting, San Francisco, California, 25-26 April 2009.
- Verdoolaege, S., Bruynooghe, M., Jul. 2008. Algorithms for weighted counting over parametric polytopes: A survey and a practical comparison. In: Beck, M., Stoll, T. (Eds.), *The 2008 International Conference on Information Theory and Statistical Learning*.

- Verdoolaege, S., Janssens, G., Bruynooghe, M., Jun. 2009a. Equivalence checking of static affine programs using widening to handle recurrences. In: *Computer Aided Verification 21*. Springer, pp. 599–613.
- Verdoolaege, S., Palkovic, M., Bruynooghe, M., Janssens, G., Catthoor, F., 2009b. Experience with widening based equivalence checking in realistic multimedia systems. In: *High Level Design Validation and Test Workshop*. Accepted.
URL <https://lirias.kuleuven.be/handle/123456789/242220>
- Verma, M., Marwedel, P., 2007. *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*. Springer Publishing Company, Incorporated.