Construction of Exact Polyhedral Model for Affine Programs with Data Dependent Conditions

Arkady V. Klimov

Institute of Design Problems in Microelectronics, Russian Academy of Sciences, Moscow, Russia
arkady.klimov@gmail.com

Abstract. Parallelizing compilers usually build polyhedral model (PM) for program parts which are referred to as static control parts (SCoP) and normally include regular nested loops with bounds and array indices being affine expressions of surrounding loop variables and parameters. Usually, PM has the form of a parametric (depending on integer parameters) graph that connects all array reads with respective array writes. Sometimes certain extensions of program class are allowed. We present our graph representation language and our original way to build it. Our buider allows for some dynamic control elements, e.g. arbitrary structured `if` with data dependent condition. Data dependent array indices can be incorporated in similar way.
Our PM representation can be treated as a program, equivalent to the original one, in a programming language with specific semantics. Two different semantic definitions are proposed. The ordinary one is SRE, System of Recursive Equations, and another, our original one, is DF, a dataflow semantics which is in some sense inverse to SRE. In particular, this means that our model is exact, whereas existing approaches yield generally, for programs with dynamic control elements, only approximate, or fuzzy models (in the sense that they appoint to some read not a specific write, but a set of possible writes).
As the PM carries the total semantics of the program, it can be used with various purposes for analysis, transformation, equivalence testing, etc. instead of original programs.

Keywords: polyhedral model, affine loop nests, data dependent conditionals, recurrence equations, dataflow semantics, program analysis, program transformation, equivalence testing

1 Introduction

The concept of polyhedral model (PM) appeared in the domain of automated parallelization. Most modern parallelizing compilers use the polyhedral model as an important source of information about the source program on behalf of the ability of reordering or parallelizing statements. Unfortunately, the class of

programs for which the model can be built is strongly restricted. Normally, it embraces affine loop nests with assignments in between, in which loop bounds and array element indices are affine expressions of surrounding loop variables and fixed structure parameters (array sizes etc.). `If`-statements with affine conditions are also allowed. Methods of building exact polyhedral model are well developed [3–6, 18] for this class of programs.

The polyhedral model provides statement instance-wise and array element-wise information on the dependences between all array element reads and array element writes. The compiler usually wants to know whether there is a dependence between given two statements under certain conditions. However, for the use in program parallelization, this model generally does not need to be precise: the exact information flow is irrelevant and false positive dependences are admissible.

In contrast, our aim is to totally convert the source program into the dataflow computation model such that it could be executed in a suitable machine. Thus we need the exact flow dependence information, and any kind of approximation is unacceptable. But as we know exactly all flow (true) dependences, we may ignore all other kinds of dependences, such as input, output, or anti-dependences.

When the source program is purely affine, the usual polyhedral model is exact and sufficient for our purpose. In such a model for each instance of read (load) operation there is an indication of the unique instance of write (store) operation that has written the value being read. This indication is usually represented in the form of a function (the so-called source function) that takes iteration vector of the read (and structure parameters) and produces the name and the iteration vector of a write or symbol $\perp$ indicating that such writes do not exist and the original state of memory is read.

However when the source program contains also one or several `if`-statements with non-affine (e.g. data dependent) conditions the known methods suggest only approximate model which identifies a set of possible writes for each read. Authors usually refer such models as fuzzy [6]. In some specific cases their model may provide a source function that uses as its input also values of predicates associated with non-affine conditionals in order to produce the unique source. As a rule these cases are those in which the number of such predicate values is finite (uniformly bounded).

In contrast, we built for arbitrary affine program with non-affine conditionals an exact and complete polyhedral model. Our model representation language is extended with predicate symbols corresponding to non-affine conditions of the source code. From such a model the exact source function for each read can be easily extracted. The resulting source function depends generally on iteration vector of the read and structure parameters as well as on an unlimited number of predicate values.

But the source function is not our aim. Rather it is the complete dataflow model, which can be treated, independently of the source program, as another program in a dataflow computation model. From the parallelization perspective this program carries implicitly the maximum amount of parallelism that is

reachable for the source program. A more detailed motivation and presentation of our approach can be found in [12, 13].

In this paper we describe our original way of building dataflow model for affine programs and then show how it can be expanded to programs with non-affine conditionals. The affine class and the concept of affine solution tree are defined in Section 2. Sections 3–5 describe our algorithm of construction of the PM. It comprises of several passes. The first pass building effects is described in Section 3. In Section 3.1 we introduce the concept of statement effect and define the process of its construction along the AST. Simultaneously we build a resulting graph skeleton which is described in Section 3.2. Section 3.3 (together with 5.2) describes our method of dealing with non-affine conditionals. Section 4 is about the second pass building states. In Section 4.1 we introduce the concept of state and define the process of its computation along the AST. In Section 4.2 we use states to build all source functions comprising the source graph (S-graph). In Section 5 we describe the third pass in which S-graph is inverted (Section 5.1) into a use graph (U-graph) which allows for direct execution in the dataflow computation model. Section 5.2 explains additional processing required for non-affine conditionals. Several examples are presented in Section 6. Section 7 is devoted to possible applications of polyhedral model. Section 8 compares and bridges our approach and achievements with those described in the literature.

2 Some Formalism

Consider a Fortran program fragment P, subroutine for simplicity. First of all, we are interested in memory accesses that have the form of array element access and are divided into reads and writes. Usually, in an assignment statement, there are zero or several read accesses and a single write access. For simplicity and without loss of generality we allow accesses only to an individual array element, not to a whole array or subarray. Scalars are treated as 0-dimension arrays.

We define a computation graph by simply running the program P with some input data. The graph consists of two kinds of nodes: reads and writes, corresponding respectively to executions of read or write memory accesses. There is a link from a write instance w to a read instance r if r reads the value written by w. In other words, r uses the same memory cell as w and w is the last write to this cell before r.

Now that our purpose is to obtain a compact parametric description of all such graphs for a given program P, we consider a limited class of programs, for which such a description is feasible. Such programs must fit the so-called affine class, which can be formally defined by the set of constructors presented in Fig.1. The right hand side e of an assignment may contain array element access $\mathsf{A}(i_1, \ldots, i_k)$, $k \geq 0$. All index expressions as well as bounds e_1 and e_2 of `do`-loops must be affine in surrounding loop variables and structure parameters. *Affine* expressions are those built from variables and integer constants with addition, subtraction and multiplication by literal integer. Also, in an affine expression,

Λ	(empty statement)
$\mathtt{A}(i_1, \ldots, i_k) = e$	(assignment, $k \geq 0$)
$S_1; S_2$	(sequence)
`if` c `then` S_1`; else` S_2`; endif`	(conditional)
`do` $v = e_1, e_2$`;` S`; enddo`	(`do`-loop)

Fig. 1. Affine program constructors

we allow whole division by literal integer. Condition c also must be affine, i.e. equivalent to $e = 0$ or $e > 0$ where e is affine.

Programs that satisfy these limitations are often called static control programs (SCoP) [6,7]. Their computation graph depends only on symbolic parameters and does not depend on dynamic data values. Further we remove the restriction that conditional expression c must be affine. Programs of the extended class are usually called dynamic control programs (DCoP). We wont consider programs with while-loops or non-affine array indices leaving it to future investigation.

A point in the computation trace of an affine program may be identified as (s, I_s), where s is a (name of a) point in the program and I_s is the iteration vector, that is a vector of integer values of all enclosing loop variables of point s. The list of these variables will be denoted as $\boldsymbol{I}_s$, which allows to depict the point s itself as $(s, \boldsymbol{I}_s)$. (Here and below boldface symbols denote variables or list of variables as syntactic objects, while normal italic symbols denote some values as usual).

Thus, denoting an arbitrary read or write instance as (r, I_r) or (w, I_w) respectively, we represent the whole computation graph as a mapping:

$$\mathrm{F}_P : (r, I_r) \mapsto (w, I_w) \tag{1}$$

which for any read node (r, I_r) yields the write node (w, I_w) that has written the value being read, or yields $\perp$ if no such write exist and thus the original contents of the cell is read. In other words, it yields a source for each read. Thus this form of graph is called a source graph, or S-graph.

However, for translation to our dataflow computation model we need the reverse: for each write node to find all read nodes (and there may exist several or none of them) which read the very value written. So, we need the multi-valued mapping

$$\mathrm{G}_P : (w, I_w) \mapsto \{(r, I_r)\} \tag{2}$$

which for each write node (w, I_w) yields a set of all read nodes $\{(r, I_r)\}$ that read that very value written. We will refer to this form of computation graph as a use graph, or U-graph.

A subgraph of S-graph (U-graph) associated with a given read r (write w) will be referred to as an r-component (w-component).

For each program statement (or point) s we define the domain $\mathsf{Dom}(s)$ as a set of values of iteration vector I_s, such that (s, I_s) occurs in the compu-

tation. The following proposition summarizes the well-established property of affine programs [4–7, 15, 18, 23] (which is also justified by our algorithm).

Proposition 1. *For any statement* $(s, \boldsymbol{I}_s)$ *in affine program* P *its domain* $\mathsf{Dom}(s)$ *can be represented as finite disjoint union* $\bigcup_i D_i$*, such that each subdomain* D_i *can be specified as a conjunction of affine conditions of variables* $\boldsymbol{I}_s$ *and structure parameters, and, when the statement is a read* $(r, \boldsymbol{I}_r)$*, there exist such* D_i *that the mapping* F_P *on each subdomain* D_i *can be represented as either* $\perp$ *or* $(w, (e_1, \ldots, e_m))$ *for some write* w*, where each* e_i *is an affine expression of variables* $\boldsymbol{I}_r$ *and structure parameters.*

This property suggests the idea to represent each r-component of F_P as a solution tree with affine conditions at branching vertices and terms of the form $S\{e_1, \ldots, e_m\}$ or $\perp$ at leaves. A similar concept of quasi-affine solution tree, *quast*, was suggested by P. Feautrier [5].

A single-valued solution tree (S-tree) is a structure used to represent r-components of a S-graph. Its syntax is shown in Fig.2. It uses just linear expressions (L-expr) in conditions and term arguments, so a special vertex type was introduced in order to implement integer division.

S-tree	::=	$\perp$	
		| *term*	
		| (*L-cond* $\rightarrow$ *S-tree*$_t$: *S-tree*$_f$)	(branching)
		| (*L-expr* =: *num var* + *var* $\rightarrow$ *S-tree*)	(integer division)
term	::=	*name*$\{$*L-expr*$_1, \ldots,$ *L-expr*$_k\}$	($k \geq 0$)
var	::=	*name*	
num	::=	$\ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid 3 \mid \ldots$	
L-cond	::=	*L-expr* $= 0$ | *L-expr* > 0	(affine condition)
L-expr	::=	*num* | *numvar* + *L-expr*	(affine expression)
atom	::=	$\perp$ | *name*$\{$*num*$_1, \ldots,$ *num*$_k\}$	(ground term, $k \geq 0$)

Fig. 2. Syntax for single-valued solution tree

Given concrete integer values of all free variables of the S-tree it is possible to evaluate the tree with a ground term as a result value. Here are evaluation rules, which must be applied iteratively while it is possible.

A branching like $(c \rightarrow T_1 : T_2)$ evaluates to T_1 if conditional expression c evaluates to true, otherwise to T_2.

A division $(e =: m\boldsymbol{q} + \boldsymbol{r} \rightarrow T)$ introduces two new variables $(\boldsymbol{q}, \boldsymbol{r})$ that take respectively the quotient and the remainder of integer division of integer value of e by positive constant integer m. The tree evaluates as T with parameter list extended with values of these two new variables. Note that the whole tree does not depend on variables $\boldsymbol{q}$ and $\boldsymbol{r}$ because they are bound variables.

It follows from Proposition 1 that for an affine program P the r-component of the S-graph F_P for each read $(r, \boldsymbol{I}_r)$ can be represented in the form of S-tree T depending on variables $\boldsymbol{I}_r$ and structure parameters.

However the concept of S-tree is not sufficient for representing w-components of U-graph, because those must be multi-valued functions in general. So, we extend the definition of S-tree to the definition of multi-valued tree, M-tree, by two auxiliary rules shown on Fig 3.

$$
\begin{array}{lll}
\textit{M-tree} ::= \ldots \text{the same as for } S\text{-tree} \ldots & & \\
\quad \mid (\&\textit{M-tree}_1 \ldots \textit{M-tree}_n) & \text{(finite union, } n \geq 2) \\
\quad \mid (@\, \textit{var} \rightarrow \textit{M-tree}) & \text{(infinite union)}
\end{array}
$$

Fig. 3. Syntax for multi-valued tree

The semantics also changes. The result of evaluating M-tree is a set of atoms. Symbol $\perp$ now represents the empty set, and the term $N\{\ldots\}$ represents a singleton.

To evaluate $(\&T_1, \ldots, T_n)$ one must evaluate sub-trees T_i and take the union of all results. The result of evaluating $(@\boldsymbol{v} \rightarrow T)$ is mathematically defined as the union of infinite number of results of evaluating T with each integer value v of variable $\boldsymbol{v}$. In practice the result of evaluating T is non-empty only within some bound interval of values v. In both cases the united subsets are supposed to be disjoint.

Below we present the scheme of our algorithm of building a S-graph (Sections 3 and 4) and then a U-graph (Section 5).

3 Building Statement Effect

3.1 Statement Effect and its Evaluation

Consider a program statement S, which is a part of an affine program P, and some k-dimensional array A. Let $(w\mathrm{A}, \boldsymbol{I}_{w\mathrm{A}})$ denote an arbitrary write operation on an element of array A within a certain execution of statement S, or the totality of all such operations. Suppose that the body of S depends affine-wise on free parameters $p_1, \ldots, p_l$ (in particular, they may include variables of loops surrounding S in P). We define the effect of S with respect to array A as a function

$$\mathsf{E}_{\mathrm{A}}[S] : (p_1, \ldots, p_l; q_1, \ldots, q_k) \mapsto (w\mathrm{A}, \boldsymbol{I}_{w\mathrm{A}}) + \perp$$

that, for each tuple of parameters $p_1, \ldots, p_l$ and indices $q_1, \ldots, q_k$ of an element of array $\mathtt{A}$, yields an atom $(w\mathrm{A}, \boldsymbol{I}_{w\mathrm{A}})$ or $\perp$. The atom indicates that the write operation $(w\mathrm{A}, \boldsymbol{I}_{w\mathrm{A}})$ is the last among those that write to element $\mathrm{A}(q_1, \ldots, q_k)$ during execution of S with affine parameters $p_1, \ldots, p_l$ and $\perp$ means that there are no such operations.

The following statement is another form of Proposition 1: *the effect can be represented as an S-tree with program statement labels as term names.* We shall call them simply *effect trees.*

Building effect is the core of our approach. Using S-trees as data objects we implemented some operations on them that are used in the algorithm presented on Fig.4. A good mathematical foundation of similar operations for similar trees has been presented in [8].

The algorithm proceeds upwards along the AST from primitives like empty and assignment statements. Operation Seq computes the effect of a statement sequence from the effects of component statements. Operation Fold builds the effect of a do-loop given the effect of the loop body. For conditional statement with affine condition the effect is built just by putting the effects of branches into the new conditional node.

$\mathsf{E}_\mathrm{A}[\Lambda] = \bot$ (empty statement)

$\mathsf{E}_\mathrm{A}[S_1; S_2] = \mathsf{Seq}(\mathsf{E}_\mathrm{A}[S_1], \mathsf{E}_\mathrm{A}[S_2])$ (sequence)

$\mathsf{E}_\mathrm{A}[LA:\ \mathrm{A}(e_1, \ldots, e_k) = e] =$ (assignments to A)
$(\boldsymbol{q}_1 = e_1 \to \ldots (\boldsymbol{q}_k = e_k \to LA\{\boldsymbol{I}\} : \bot) \ldots : \bot)$
where $\boldsymbol{I}$ is a list of all outer loop variables

$\mathsf{E}_\mathrm{A}[LB:\ \mathrm{B}(\ldots) = e] = \bot$ (other assignments)

$\mathsf{E}_\mathrm{A}[\texttt{if } c \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ endif}] = (c \to \mathsf{E}_\mathrm{A}[S_1] : \mathsf{E}_\mathrm{A}[S_2])$ (conditional)

$\mathsf{E}_\mathrm{A}[\texttt{do } \boldsymbol{v} = e_1, e_2\texttt{; } S\texttt{; enddo}] = \mathsf{Fold}(\boldsymbol{v}, e1, e2, \mathsf{E}_\mathrm{A}[S])$ (do-loop)

Fig. 4. The rules for computing effect tree wrt k-dimensional array A

The implementation of function Seq is straight. To compute $\mathsf{Seq}(T_1,T_2)$ we simply replace all $\bot$ leaves in T_2 with a copy of T_1. The result is then be simplified by a function Prune which prunes unreachable branches by checking the feasibility of affine conjunctions (the check is known as Omega-test [18]).

The operation $\mathsf{Fold}(\boldsymbol{v}, e_1, e_2, T)$, where $\boldsymbol{v}$ is a variable and e_1 and e_2 are affine expressions, produces another S-tree T' that does not depend on $\boldsymbol{v}$ and represents the following function. Depending on all other parameters of e_1, e_2 and T we find the maximum value v of variable $\boldsymbol{v}$ in between values of e_1 and e_2, for which T evaluates to a term t (not $\bot$), and yield the term t for that value v as the result. Building this T' usually involves the solution of parametric integer programming problems (1-dimensional) and combining the results.

3.2 Graph Node Structure

In parallel with building the effect of each statement we also compose a graph skeleton, which is a set of nodes with placeholders for future links. For each assignment a separate node is created. At this stage the graph nodes are associated with AST nodes, or statements, in which they were created, for the purpose that will be explained below in Section 4. The syntax (structure) of a graph node description is presented in Fig.5.

Non-terminals ending with s usually denote an arbitrary number of its base word (a repetition), e.g. *ports* signifies *list of ports*. A node consists of a header

node ::= (*node* (*name context*)
(*dom conditions*)
(*ports ports*)
(*body computations*)
)
context ::= *names*
condition ::= *L-cond* | *TF-tree*
port ::= (*name type source*)
computation ::= (*eval name type expression destination*)
source ::= *S-tree* | IN
destination ::= *M-tree* | OUT

Fig. 5. Syntax for graph node description

with name and context, domain description, list of ports that describe inputs and a body that describes output result. The context here is just a list of loop variables surrounding the current AST node. The domain specifies a condition on these variables for which the graph node instance exists. Besides context variables it may depend on structure parameters. Ports and body describe inputs and outputs. The source in a port initially is usually an atom (or, generally, an *S*-tree) depicting an array access (array name and index expressions), which must be eventually resolved into a *S*-tree referencing other graph nodes as the sources of the value (see Section 4.2). A computation consists of a local name and type of an output value, an expression to be evaluated, and a destination placeholder $\perp$ which must be replaced eventually by a M-tree that specifies output links (see Section 5). The tag IN or OUT declares the node as input or output respectively.

Consider for example a statement `S=S+A(i)` of the summation program in Fig.8a. The initial view of the corresponding graph node is shown in Fig.6. Note that the expression in **eval** clause is built from the right hand side by replacing all occurrences of scalar or array element references with their local names (that became port names as well). A graph node for assignment usually has a single **eval** clause that represents the generator of values written by the assignment. Thereby a term of effect tree may be considered as a reference to a graph node output.

(**node** (S1 i)
(**dom** ($i \geq 1$)($i \leq n$))
(**ports** (s1 double S{}) (a1 double A{i}))
(**body** (**eval** S double ($s1 + a1$) $\perp$))
)

Fig. 6. An initial view of graph node for statement `S=S+A(i)`

3.3 Processing Non-affine Conditionals

When the source program contains a non-affine conditional statement S, special processing is needed. We add a new kind of condition, a predicate function call, or simply predicate, depicted as

$$name^{bool\text{-}const}\{L\text{-}exprs\} \tag{3}$$

that may be used everywhere in the graph where a normal affine expression can. It contains a name, sign T or F (affirmation or negation) and a list of affine arguments.

However, not all operations can deal with such conditions in argument trees. In particular, the Fold cannot. Thus, in order that Fold can work later we perform the elimination of predicates immediately after they appear in the effect tree of a non-affine conditional statement.

First, we drag the predicate p, which is initially on the top of the effect tree $\mathsf{E}_\mathrm{A}[S] = (p \to T_1 : T_2)$, downward to leaves. The rather straightforward process is accomplished with pruning. In the result tree, T_S, all copies of predicate p occur only in downmost positions of the form $(p \to A_1 : A_2)$, where each A_i is either term or $\bot$. We call such conditional sub-trees *atomic*. In the worst case the result tree will have a number of atomic sub-trees being a multiplied number of atoms in sub-trees T_1 and T_2.

Second, each atomic sub-tree can now be regarded as an indivisible composite value source. When one of A_i is $\bot$, this symbol depicts an implicit rewrite of an old value into the target array cell $\mathtt{A}(q_1, \ldots, q_k)$ rather than just no write. With this idea in mind we now replace each atomic sub-tree U with a new term $\boldsymbol{U}_\mathrm{new}\{i_1, \ldots, i_n\}$ where argument list is just a list of variables occurring in the sub-tree U. Simultaneously, we add the definition of $\boldsymbol{U}_\mathrm{new}$ in the form of a graph node (associated with the conditional statement S as a whole) which is shown in Fig.7. This kind of nodes will be referred to as blenders as they blend two input sources into a single one. The domain of the new node is that of statement

$$\begin{array}{l}(\textbf{node}\ (\boldsymbol{U}_\mathrm{new}\ i_1 \ldots i_n)\\ \qquad (\textbf{dom}\ \mathsf{Dom}(S) + \text{path-to-}U\text{-in-}T_S)\\ \qquad (\textbf{ports}\ (\mathrm{a}\ t\ (p \to \mathsf{RW}(A_1) : \mathsf{RW}(A_2))))\\ \qquad (\textbf{body}\ (\textbf{eval}\ a\ t\ a\ \bot)\ )\\ \qquad)\end{array}$$

Fig. 7. Initial contents of the blender node for atomic subtree U in $\mathsf{E}_\mathrm{A}[S] = T_S$

S restricted by conditions on the path to the sub-tree U in the whole effect tree T_S. The result is defined as just copying the input value a (of type t). The most intriguing is the source tree of the sole port a. It is obtained from the atomic sub-tree $U = (p \to A_1 : A_2)$. Each A_i is replaced (by operator RW) as follows. When A_i is a term it remains unchanged. Otherwise, when A_i is $\bot$, it is replaced with

explicit reference to the array element being rewritten, $\mathtt{A}(q_1, \ldots, q_k)$. However, an issue arises: variables $q_1, \ldots, q_k$ are undefined in this context. The only variables allowed here are $i_1, \ldots, i_n$ (and fixed structure parameters). Thus we need to express indices $q_1, \ldots, q_k$ through known values $i_1, \ldots, i_n$.

To resolve this issue consider the list of (affine) conditions L on the path to the subtree U in the whole effect tree T_S as a set of equations connecting variables $q_1, \ldots, q_k$ and $i_1, \ldots, i_n$.

Proposition 2. *Conditions L specify a unique solution for values $q_1, \ldots, q_k$ depending on $i_1, \ldots, i_n$.*

Proof. Consider another branch A_j of subtree U, which must be a term. We prove a stronger statement, namely, that given exact values of all free variables occurring in A_j, $\mathsf{Vars}(A_j)$, all q-s are uniquely defined. The term A_j denotes the source for array element $\mathtt{A}(q_1, \ldots, q_k)$ within some branch of the conditional statement S. Note, however, that this concrete source is a write on a single array element only. Hence, array element indices $q_1, \ldots, q_k$ are defined uniquely by $\mathsf{Vars}(A_j)$. Now recall that all these variables are present in the list $i_1, \ldots, i_n$ (by definition of this list). □

Now that the unique solution does exist, it can be easily found by our affine machinery. See Section 5 in which the machinery used for graph inversion is described.

Thus, we obtain, for conditional statement S, the effect tree that does not contain predicate conditions. All predicates got hidden within new graph nodes. Hence we can continue the process of building effects using the same operations on trees as we did in the purely affine case. Also for each predicate condition a node must be created that evaluates the predicate value.

We shall return back to processing non-affine conditionals in Section 5.2.

4 Evaluation and Usage of States

4.1 Computing States

A state before statement $(s, \boldsymbol{I}_s)$ in affine program fragment P with respect to array element $\mathtt{A}(q_1, \ldots, q_k)$ is a function that takes as arguments the iteration vector $I_s = (i_1, \ldots, i_n)$, array indices $(q_1, \ldots, q_k)$ and values of structure parameters and yields the write (w, I_w) in the computation of P that is the last among those that write to array element $\mathtt{A}(q_1, \ldots, q_k)$ before (s, I_s).

In other words this function presents an effect of executing the program from the beginning up to the point just before (s, I_s) wrt array $\mathtt{A}$. It can be expressed as an S-tree, which may be called a *state tree* at program point before statement s for array $\mathtt{A}$.

To compute state trees for each program point we use the following method.

So far for each statement B in an affine program fragment P we have computed the S-tree $\mathsf{E}_{\mathrm{A}}[B]$ representing the effect of B wrt array $\mathtt{A}$. Now we are to

compute for each statement B the S-tree $\Sigma_{\mathrm{A}}[B]$ representing the state before B wrt array `A`.

For the starting point of program P we set

$$\Sigma_{\mathrm{A}}[P] = (\boldsymbol{q}_1 \geq l_1 \to (\boldsymbol{q}_1 \leq u_1 \to \ldots \mathrm{A\_init}\{q_1, \ldots\} \ldots : \bot) : \bot) \tag{4}$$

where term $\mathrm{A\_init}\{q_1, \ldots, q_k\}$ signifies an untouched value of array element $\mathtt{A}(q_1, \ldots, q_k)$ and l_i, u_i are lower and upper bounds of array dimensions (which are only allowed to be affine functions of fixed parameters). Thus, (4) signifies that all `A`'s elements are untouched before the whole program P.

The further computation of Σ_{A} is described by the following production rules:

1. Let $\Sigma_{\mathrm{A}}[B_1; B_2] = T$. Then $\Sigma_{\mathrm{A}}[B_1] = T$. *The state before any starting part of B is the same as that before B.*
2. Let $\Sigma_{\mathrm{A}}[B_1; B_2] = T$. Then $\Sigma_{\mathrm{A}}[B_2] = \mathsf{Seq}(T, \mathsf{E}_{\mathrm{A}}[B_1])$. *The state after the statement B_1 is that before B_1 combined by* Seq *with the effect of B_1.*
3. Let $\Sigma_{\mathrm{A}}[\mathtt{if}\ c\ \mathtt{then}\ B_1\ \mathtt{else}\ B_2\ \mathtt{endif}] = T$. Then $\Sigma_{\mathrm{A}}[B_1] = \Sigma_{\mathrm{A}}[B_2] = T$. *The state before any branch of* `if`*-statement is the same as before the whole* `if`*-statement.*
4. Let $\Sigma_{\mathrm{A}}[\mathtt{do}\ \boldsymbol{v} = e_1, e_2;\ B;\ \mathtt{enddo}] = T$. Then

$$\Sigma_{\mathrm{A}}[B] = \mathsf{Seq}(T, \mathsf{Fold}(\boldsymbol{v}, e_1, \boldsymbol{v}{-}1, \mathsf{E}_{\mathrm{A}}[B])) \tag{5}$$

The state before the loop body B with the current value of loop variable $\boldsymbol{v}$ is that before the loop combined by Seq *with the effect of all preceding iterations of B.*

The last form (5) needs some comments. It is the case in which the upper limit in the Fold clause depends on $\boldsymbol{v}$. To be formally correct, we must replace all other occurrences of $\boldsymbol{v}$ in the clause with a fresh variable, say $\boldsymbol{v}'$. Thus, the resulting tree will (generally) contain $\boldsymbol{v}$, as it expresses the effect of all iterations of the loop before the v-th iteration. The situation is much like that of

$$\int_0^x f(x)dx.$$

Using the rules 1-4 one can achieve (and compute the state in) any internal point of the program P (a point may be identified by a statement following it). For speed, we do not compute the state wrt array `X` at some point if there are no accesses to `X` within the current block after that point. Also, we compute only once the result of Fold with a variable as the upper limit and then use the result both for the effect of the whole loop and for the state at the beginning of the body.

The following proposition limits the usage, within a state tree T, of terms whose associated statements are enclosed in a conditional statement with non-affine condition. It will be used further in Section 5.2.

Proposition 3. *Let a conditional statement S with non-affine condition be at a loop depth m within a dynamic control program P. Consider a state tree $ST_p =$*

$\Sigma_{\mathrm{A}}[p]$ in a point p within P w.r.t. an array `A`*. Let $A\{i_1, \ldots, i_k\}$ be a term in ST_p, whose associated statement, also A, is inside a branch of S. Then the following claims are all true:*

- *$m \leq k$,*
- *p is inside the same branch of S and*
- *indices $i_1, \ldots, i_m$ are just variables of loops enclosing S.*

Proof. Let A be a term name, whose associated statement A is inside a branch b of a conditional statement S with non-affine condition. It is either assignment to an array, say `A`, or a blender node emerged from some inner conditional (performing a "conditional assignment" to `A`). From our way of hiding predicate conditions described in Section 3.3 it follows that the effect tree of S, $\mathsf{E}_{\mathrm{A}}[S]$, as well as of any other statement containing S, will not contain a term with name A. Hence, due to our way of building states from effects described above, this is also true for the state tree of any point outside S, including the state ST_S before the S itself. Now, consider the state ST_p of a point p within a branch b_1 of S. (Below we'll see that $b_1 = b$). We have

$$ST_p = \mathsf{Seq}(ST_S, ST_{S-p}), \tag{6}$$

where ST_{S-p} is the effect of executing the code from the beginning of the branch b_1 to p (recall that the state before the branch b_1, ST_{b_1}, is the same as ST_S according to Rule 3 above). Consider a term $A\{i_1, \ldots, i_k\}$ in ST_p. As it is not from ST_S, it must be in ST_{S-p}. Obviously, ST_{S-p} contains only terms associated with statements of the same branch with p. Thus, $b_1 = b$. And these terms are only such that their initial m indices are just variables of m loops surrounding S. Thus, given that the operation Seq does not change term indices, we have the conclusion of Proposition 3. □

4.2 Resolving Array Accesses

Now we shall use states before each statement to accomplish building the source graph F_P. Consider a graph node example shown on Fig.6. Initially, source trees in **ports** clause contain terms denoting references to array element, like $\mathtt{A}\{\mathtt{e_1}, \ldots, \mathtt{e_k}\}$. We want to resolve each such array access term into subtree with only normal source terms. Recall that each graph node is associated with a certain point p in the AST (that is, a statement, or the program point before the statement) and that we already have a state $\Sigma_{\mathrm{A}}[p]$. Now we apply $\Sigma_{\mathrm{A}}[p]$ as a function to indices $(e_1, \ldots, e_k)$ and use the result as a replacement for the term $\mathtt{A}\{\mathtt{e_1}, \ldots, \mathtt{e_k}\}$. As we do the application symbolically (just as a substitution with subsequent pruning), the result will be a S-tree. Having expanded each such array access term we get the port source tree in which all terms refer only to other graph nodes. And the set of all port source trees in all graph nodes comprises the source graph F_P.

Recall that each graph node X has a domain $\mathsf{Dom}(X)$ which is a set of possible context vectors. It is specified by a list of conditions, which are collected

from surrounding loop bounds and `if` conditions. The list may contain also predicate conditions. We write $D \Rightarrow p$ to indicate that the condition p is valid in D (or follows from D). In case of a predicate condition $p = \boldsymbol{p}^b\{e_1, \ldots, e_k\}$ it signifies that the list D just contains p (or contains some other predicate $p = \boldsymbol{p}^b\{f_1, \ldots, f_k\}$ such that $D \Rightarrow (e_i = f_i)$ for all $i = 1, \ldots, k$). For a S-graph built so far the following proposition limits the usage of atoms $A\{\ldots\}$ for which $\mathsf{Dom}(A)$ has predicate condition.

Proposition 4. *Suppose that B is a regular node (not a blender) whose source tree T contains a term $A\{i_1, \ldots, i_k\}$ (corresponding to an assignment to an array* `A`*). Let $\mathsf{Dom}(A) \Rightarrow p$, where $p = \boldsymbol{p}^b\{j_1, \ldots, j_m\}$ is a predicate condition. Then:*

- $m \le k$,
- $j_1 = i_1, \ldots, j_m = i_m$, *and all these are just variables of loops enclosing the conditional statement with predicate p,*
- $\mathsf{Dom}(B) \Rightarrow p$.

Proof. As $\mathsf{Dom}(A) \Rightarrow \boldsymbol{p}^b\{j_1, \ldots, j_m\}$, the predicate p denotes the condition of a conditional statement S enclosed by m loops with variables $j_1, \ldots, j_m$, and this S contains the statement A in the branch b (by construction of Dom). The source tree T was obtained by a substitution into the state tree before B, $ST_B = \Sigma_{\mathrm{A}}[B]$, which must contain a term $A\{i'_1, \ldots, i'_k\}$. It follows, by Proposition 3, that statement B is inside the same branch b (hence, $\mathsf{Dom}(B) \Rightarrow p$), $m \le k$ and $i'_1, \ldots, i'_m$ are just variables $j_1, \ldots, j_m$. However the substitution replaces only formal array indices and does not touches enclosing loop variables, here $j_1, \ldots, j_m$. Hence $i'_1 = i_1, \ldots, i'_m = i_m$. □

When B is a blender the assertion of the Proposition 4 is also valid but $\mathsf{Dom}(B)$ should be extended with conditions on the path from the root of the source tree to the term $A\{\ldots\}$. The details are left to the reader.

5 Building the Dataflow Model

5.1 Building *U*-graph by Inverting *S*-graph: Affine Case

In dataflow computation model the data flow from source nodes to use nodes. Thus, the generating node must know exactly which other nodes (and by which port) need the generated value and sends the data element to all such node-port pairs. This information, known also as use graph G_P, is to be represented in the form of destination M-trees in the eval clauses of graph nodes. Initially they are all set to $\perp$ as just placeholders.

Suppose the source program fragment P is purely affine. Having the source graph F_P in the form of affine S-trees in node ports, it is not difficult to produce the inversion resulting in M-trees.

The graph is inverted path-wise: first, we split each tree into paths. Each path starts with header term $R\{i_1, \ldots, i_n\}$, ends with term $W\{e_1, \ldots, e_m\}$ and

has a list of affine conditions extended with quotient/remainder definitions like ($e =: k\boldsymbol{q}+\boldsymbol{r}$) in between ($k$ is a literal integer here). Only variables $i_1, \ldots, i_n$ and structure parameters can be used in affine expressions $e, ei, \ldots$. New variables $\boldsymbol{q}$ and $\boldsymbol{r}$ may be used only to the right of their definition. The InversePath operation produces the inverted path that starts with header term $W\{j_1, \ldots, j_m\}$ with new formal variables $j_1, \ldots, j_m$, ends with term $R\{f_1, \ldots, f_n\}$ and has a list of affine conditions and divisions in between. Also, universally quantified variables can be introduced by clause ($\mathtt{@}\boldsymbol{v}$). All affine expressions f_i are built with variables $j_1, \ldots, j_m$, structure parameters and $\boldsymbol{q}/\boldsymbol{r}/\mathtt{@}\boldsymbol{v}$-variables defined earlier in the list. The inversion involves solving the system of linear Diophantine equations. In essence, it can be viewed as a projection or variable elimination process. When a variable cannot be eliminated it is simply introduced with @-clause.

In general, one or several paths can be produced. All produced paths are grouped by new headers, each group being an M-tree for respective graph node, in the form (& T_1 $T_2 \ldots$) where each T_i is a 1-path tree. Further, the M-tree can be simplified by the operation SimplifyTree. This operation also involves finding bounds for @-variables, which are then included into @-vertices in the form:

$$(\mathtt{@}\boldsymbol{v}(l_1 u_1)(l_2 u_2) \ldots T)$$

where l_i, u_i are affine lower and upper bounds of i-th interval, and v must belong to one of the intervals.

5.2 Inverting *S*-graph for Programs with Non-affine Conditionals

When program P has non-affine conditionals the above inversion process will probably yield some M-trees with predicate conditions. Hence, a node with such M-tree needs the value of the predicate as its input. However, this value may not necessarily be needed always, and thus it may induce redundant dependences. So, when a predicate vertex in M-tree does not dominate all term leaves, we should cut the vertex off and create another node with the sub-tree as a destination tree. Otherwise, we just add to the node a Boolean port connected to a predicate evaluating node. We must do so repetitively until all predicates in M-trees refer to Boolean-valued ports.

In either case some nodes need an additional port for the value of predicate. We call such nodes *filters*. In the simplest case a filter has just two ports, one for the main value and one for the value of the predicate, and sends the main value to the destination when the predicate value is true (or false) and does nothing otherwise.

Generally, the domain of each token and each node may have several functional predicates in the condition list. Normally, a token has the same list of predicates as its source and target nodes. However, sometimes these lists may differ by one item. Namely, a filter node generates tokens with a longer predicate list whereas the blender node makes the predicate list one item shorter compared to that of incoming token. In the examples below arrows are green (dotted) or red (dashed) depending on the predicate value.

However, our aim is to produce not only U-graph, but both S-graph and U-graph which must be both complete and mutually inverse. To simplify our task we update the S-graph before inversion such that inversion does not produce predicates in M-trees. To achieve this we check for each port whether its source node has enough predicates in its domain condition list. When we see the difference, namely that the source node has less predicates, then we insert a filter node before that port. And the opposite case, that the source has more predicates, is impossible, as it follows immediately from Proposition 4.

6 Examples

A set of simple examples of a source program (subroutine) with the two resulting graphs S-graph and U-graph are shown in Figs. 8,9,11. All graphs has been built automatically in textual form and then redrawn manually in graphical view. Nodes are rectangles or other shapes and data dependences are arrows between them. Usually a node has several input and one output ports. A port is usually identified as a point on node boundary. The domain is usually shown once for a group of nodes with the same domain (at the up side in curly braces). Those groups are separated by a dotted line. Each node should be considered as a collection of instance nodes of the same type that differ in domain parameters from each other. Arrows between nodes may fork depending on some condition (usually it is affine condition of domain parameters), which is then written near the start of the arrow immediately after the fork. When arrow enters a node it carries a new context (if it has changed) written there in curly braces. The simplest and purely affine example in Fig.8 explains the notations. Arrows in the S-graph are directed from a node port to its source (node output). The S-graph

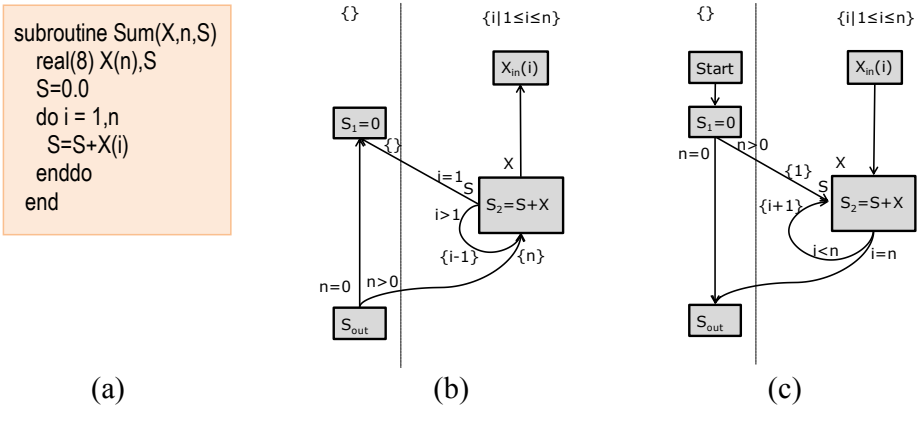


Fig. 8. Fortran program Sum (a), its S-Graph (b) and U-graph (c)

arrows can be interpreted as the flow of requests for input values. More exact semantics will be described in Section 7.2.

In the U-graph, vice versa, arrows are directed from node output to some node port. In contrast with the S-graph, they denote actual flow of data. U-graph execution obeys semantic of dataflow computation model described in Section 7.3.

In the U-graph there also appears the need to get rid of zero-port nodes which arise from assignments with no one read operation. We simply insert into such nodes a dummy port which receives a dummy value. It looks as if we use, in the right hand side of the assignment, a dummy scalar variable that is set at the start of the program P. Thus a node `Start`, which generates a token for node S1 (corresponding to the assignment `S=0.0`), appeared in the U-graph of our example.

A simplest example with non-affine conditions is shown on Fig.9. The textual view of graphs was generated automatically, whereas the graphical view was drawn by hand.

When a source program contains a non-affine conditional, in the S-graph there appears a new kind of node, the blender, depicted as a blue truncated triangle (see Fig. 9b). Formally, it has a single port, which receives data from two different sources depending on the value of the predicate. Thus, it has another implicit port for Boolean value (on top). The main port arrows go out from side edges; true and false arrows are dotted green and dashed red respectively. The S-graph semantics of the blender is:

1. Invoke the predicate and wait for the result.
2. Depending on the result execute true (green) or false (red) branch.

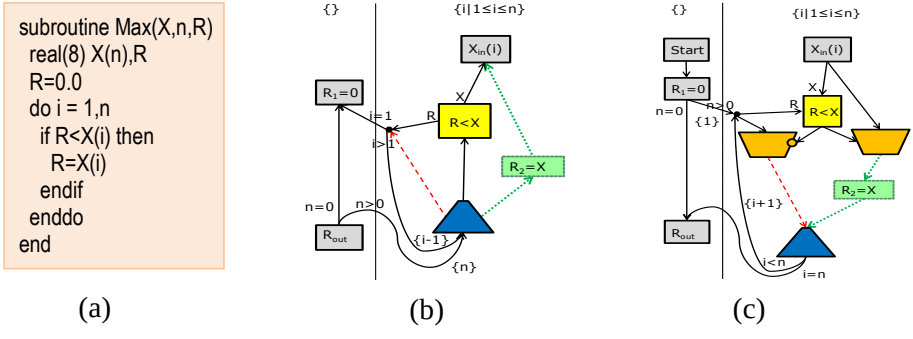


Fig. 9. Fortran program Max (a), its S-Graph (b) and U-graph (c)

In the U-graph the blender does not need a condition: in either case it receives a value token on its unique port without knowing which node has sent it and under which condition. However, when the source itself is not under the needed

condition, a filter node must be inserted in between the source node and the receiver port (it is shown in Fig.8c as an inverted yellow trapezoid). The predicate value coming into a side edge and a circle at the entry point indicate that the main value is passed when the condition is false.

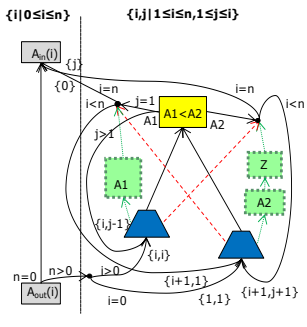
The textual view of the blender from Fig.9 is shown in Fig.10. Note the predicate FP1{i} on top of the S-tree of the unique port R. The S-tree contains a reference to BR{i-1} under conjunction $(\mathrm{FP1}^{\mathrm{F}}\{\mathrm{i}\})(i > 1)$. The backward data arrow of the U-graph (in the M-tree of **eval**clause) goes through the filter node FR1.R{i+1}.

(**node** (BR i)
(**dom** $(1 \leq \mathrm{i})(\mathrm{i} \leq \mathrm{n})$)
(**ports** (R double (FP1{i} $\rightarrow$ R2{i} : (i = 1 $\rightarrow$ R1{} : BR{i − 1}))))
(**body** (**eval** R double R(i = n $\rightarrow$ R_out{} : (&P1.R{i} FR1.R{i + 1})))

Fig. 10. The blender from example on Fig.9

A more complex and interesting example, a bubble sort program and its graphs, is shown on Fig.11. In contrast with previous ones, the U-graph exhibits high parallelism: its parallel time is $2n$ instead of $n(n + 1)/2$ for sequential execution.

```
subroutine Bubble(A,n)
  real(8) A(0:n),Z
  do i=n,1,-1
    do j=1,i
      if A(j-1)<A(j) then
        Z=A(j)
        A(j)=A(j-1)
        A(j-1)=Z
      endif
    enddo
  enddo
end
```



(a) (b) (c)

Fig. 11. Fortran program Bubble (a), its S-Graph (b) and U-graph (c)

7 Properties and Usage of Polyhedral Model

7.1 General Form of Dataflow Graph

We start from the purely static control graph, which is a set of nodes with syntax shown in Fig. 5. In this form both the S-graph and the U-graph are presented. Ignoring destination trees in **eval** clauses we get the S-graph, ignoring source trees in ports we get the U-graph. Both graphs represent the same dependence relation. It means that the respective set of trees must be mutually inverse. Recall that source trees representing the S-graph are S-trees (single-values), while destination trees forming the U-graph are M-trees (multi-valued).

Then we introduce a special kind of node called a *predicate* which produces a Boolean condition. This value is used directly by source tree of a *blender*, which is an identity node with a single port with the source tree of the form $(p \to T_1 : T_2)$, where predicate condition p has the form (3).

As we avoid predicate conditions appearing in destination trees, we introduce *filter* nodes, which are in some sense inverse to blenders. Conceptually, *filter* is an identity node with the usual input port and the destination tree of the form $(p \to T_{out} : \bot)$. But instead of predicate condition p of the form $\boldsymbol{p}^b\{e_1, \ldots, e_k\}$, we add a port named p with atom $P\{e_1, \ldots, e_k\}$ as a source tree and either $(p \to T_{out} : \bot)$ or $(p \to \bot : T_{out})$ as a destination tree. Thus filter is used as a gate which is open or closed depending on the value on port p: the gate is open, if the value is b, otherwise closed. Note that filters are necessary in U-graph, but not in S-graph.

The S-graph must satisfy the two following constraints. The first is a consistency restriction. Consider a node $\mathrm{X}\{\boldsymbol{I}\}$ with domain D_X and a source tree T. Let $I \in D_\mathrm{X}$. Then $T(I)$ is some atom $\mathrm{Y}\{J\}$ such that $J \in D_\mathrm{Y}$. The second constraint requires that the S-graph must be *well-founded*, which means that no one object node $\mathrm{X}\{I\}$ may transitively depend on itself.

7.2 Using the *S*-graph as a Program

The S-graph can be used to evaluate output values given all input values. Also, all structure parameters must be known. We assume that each node produces a single output value (otherwise atom names in source trees should refer to a node-output pair rather than just a node).

Following [6] we transform the S-graph into a System of Recurence Equations (SRE), which can be treated as a recursive functional program. Each node of S-graph is presented as definition of recursive function whose arguments are context variables. Its right hand side is composed of a body expression with ports as calls to additional functions, whose right hand sides in turn are obtained from their source trees with atoms and predicates as function calls. Input nodes are functions defined elsewhere. In Fig.12 is presented a simplified SRE for the S-graph from Fig.9b. Execution starts with invocation of the output node function. Evaluation step is to evaluate the right hand side calling other

$$
\begin{aligned}
&\mathrm{P(i)} = \mathrm{R(i)} < \mathrm{X(i)}\\
&\mathrm{B(i)} = \text{if P(i) then X(i) else R(i)}\\
&\mathrm{R(i)} = \text{if i} = 1 \text{ then R1() else if i} > 1 \text{ then B(i} - 1\text{) else } \bot\\
&\mathrm{R1()} = 0\\
&\mathrm{R_{out}} = \text{if n} = 0 \text{ then R1() else if N} > 0 \text{ then B(n) else } \bot
\end{aligned}
$$

Fig. 12. System of Recurrent Equations equivalent to S-graph on Fig.9b

invocations recursively. For efficiency it is worth doing tabulation so that neither function call is executed twice for the same argument list.

Note, that both the consistency and the well-foundedness conditions together provide the termination property of the S-graph program.

7.3 Computing the U-graph in the Dataflow Computation Model

The U-graph can be executed as program in the dataflow computation model. A node instance with concrete context values *fires* when all its ports get data element in the form of data token. Each fired instance is executed by computing all its **eval** clauses sequentially. All port and context values are used as data parameters in the execution. In each **eval** clause the expression is evaluated, the obtained value is assigned to a local variable and then sent out according to the destination M-tree. The tree is executed in an obvious way. In the conditional vertex, the left or right subtree is executed depending on the Boolean value of the condition. In &-vertices, all sub-trees are executed one after another. An -vertex acts as a `do`-loop with specified bounds. Each term of the form $R.x\{f_1, \ldots, f_n\}$ acts as a token send statement, that sends the computed value to the graph node R to port x with the context built of values of f_i. The process stops when all output nodes get the token or when all activity stops (quiescence condition). To initiate the process, tokens to all necessary input nodes should be sent from outside.

7.4 Extracting Source Functions from S-graph

There are two ways to extract the source function from the S-graph. First, we may use the S-graph itself as a program that computes the source for a given read when the iteration vector of the read as well as values of all predicates are available. We take the SRE and start evaluating the term $R(i_1, \ldots, i_n)$, where $i_1, \ldots, i_n$ are known integers, and stop as soon as some term of the form $W(j_1, \ldots, j_m)$ is encountered (where W is a node name corresponding to a write operation and $j_1, \ldots, j_m$ are some integers).

Also, there is a possibility to extract the general definition of the source function for a given read in a program. We may do it knowing nothing about predicate values. We start from the term $R\{\boldsymbol{i}_1, \ldots, \boldsymbol{i}_n\}$ where $\boldsymbol{i}_1, \ldots, \boldsymbol{i}_n$ are symbolic variables and proceed unfolding the S-graph symbolically into just the S-tree. Having encountered the predicate node we insert the branching with symbolic

predicate condition (without expanding it further). Having encountered a term $W\{e_1, \ldots, e_m\}$ we stop unfolding the branch. Proceeding this way we will generate a possibly infinite S-tree representing the source function in question. If were lucky the S-tree will be finite. It seems that, in previous works on building polyhedral models for programs with non-affine `if`-s [6, 7], the exact result is produced only when the above process stops with a finite S-tree as a result.

But we can also produce a good result even when the generated S-tree is infinite (note, that this is the case in examples `Max` and `Bubble`). Having encountered a node already visited we generalize, i.e. cut-off the earlier generated sub-tree from that node replacing it with invocation of another function and schedule the generation of a new function starting from that node. The process converges to a set of mutually recursive function definitions that implements the source function in the most unalloyed form.

The process we have just described is a particular case of the well known *supercompilation* [10,17,19,21]. In a more general setting this concept can provide a very elaborate and productive tool for transforming programs represented in the dataflow (or polyhedral) model. An interesting open issue is to invent a good *whistle* and *generalization strategy* for a supercompiler that deals with polyhedral *configurations*.

7.5 Analysis

The polyhedral model may be used for various purposes. Many useful properties of original programs can be detected. Here are some examples: array bounds checks, dead/unused code detection and feasible parallelism. Various questions can be studied by means of abstract interpretation of the polyhedral model instead of the original program.

7.6 Transformations

Initially our compiler generates a very fine grained graph in which a node corresponds to a single assignment, or condition expression, or it is a blender or a filter. Thus, it is a good idea to apply a *coagulation* transformation, that takes, say, a `Bubble` program U-graph shown in Fig.11(c), and produces a more coarse grained data flow graph as shown in Fig.13(b). Within coagulation, several small nodes are combined into a large single node. The body of a resulting node is a fusion of bodies of original nodes. Data transfer between original small nodes transforms into just variable *def* and *use* within the body of the resulting node. This code can be efficiently evaluated on a special multiprocessor system.

In Fig.13(c) is shown the computation graph for $n = 4$. Each node instance is marked with its context values. Here one can see clearly the possible parallelism.

A form of coagulation is vectorization. It involves gluing together several nodes of the same type. This transformation is the analogue of tiling for affine loop programs.

An inverse to coagulation, *atomization*, can also be useful. For example, it is beneficial before testing equivalence.

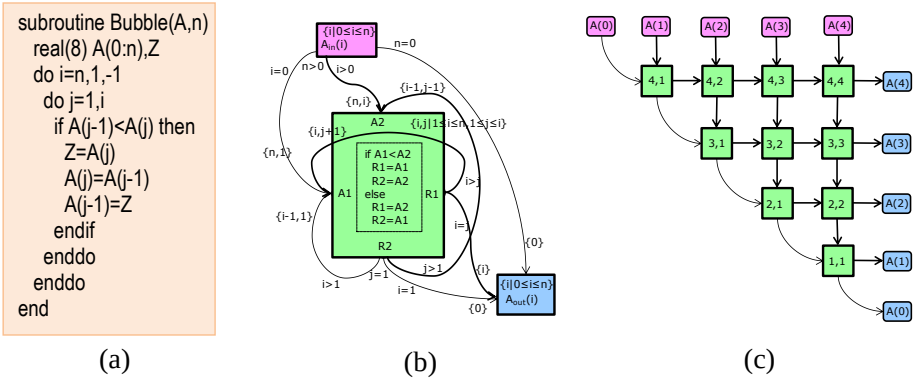


Fig. 13. Fortran program Bubble (a), its coagulated U-Graph (b) and the graph expansion for $n = 4$ (c)

7.7 Equivalence Testing

The S-graph form can be used for testing two affine programs for equivalence. Consider, for example, another version of bubble sort program, `Bubble2`, shown on Fig.14(a), its coagulated U-graph (b) and respective computation graph for $n = 4$ (c). It is easy to see that the computation graphs of both programs `Bubble` and `Bubble2` are essentially the same: they differ only in the way of numbering the nodes. To prove it one needs to find the affine mapping of contexts that would make the graphs equal.

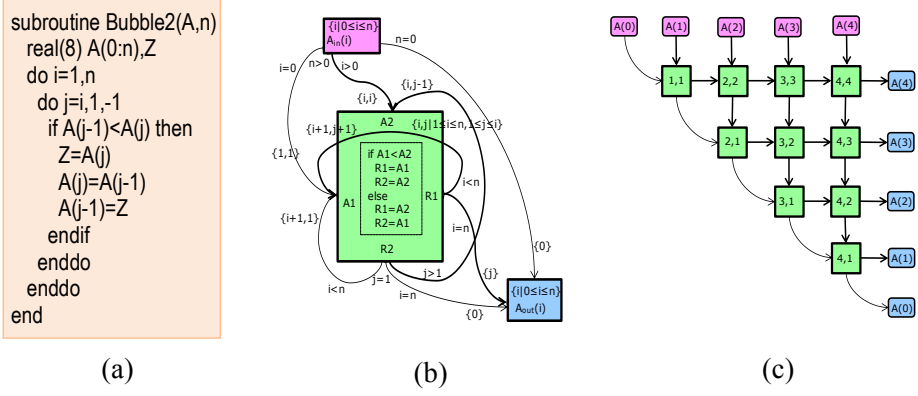


Fig. 14. Fortran program Bubble2 (a), its coagulated U-Graph (b) and the graph expansion for $n = 4$ (c)

Generally we need a regular procedure to establish equivalence of two polyhedral graphs. A good relevant procedure is presented in [22]. It allows for graphs which have a generalized form of static control dependence graph with affine dependences. Graph vertices are adorned by abstract operation symbols with single output. Thus, strictly speaking within their approach our example is not tractable: if we use coagulated form (as in Figs. 13-14) then the node function with two outputs R1 and R2 does not meet the requirement of a single output, and if we consider atomistic form (as in Fig.11) then the property of static control does not hold. Perhaps the first problem is just technical and the approach can be easily generalized. However, data dependent conditions generally cannot be easily eliminated. So, we want to generalize the approach of [22] to admit some restricted dynamic control.

Since our work is not yet finished, we just describe here the variety of dynamic control graphs which we want to be allowed (Section 7.1 above).

In our terms, the equivalence testing procedure deals with a pair of S-graphs. Following [22], it starts from output nodes denoting values of output arrays, and tries to prove that the respective values are computed by essentially the same function compositions from values of input nodes. When a predicate condition, p, is encountered in a source tree, the predicate value is 'requested' and the result is used symbolically for the selection of the source. Thus, a split appears in the proof three due to the unknown predicate value. Now we expand the equivalence testing procedure from [22], so as to correctly deal with such splits.

8 Related Work

In this Section we compare our approach with other attempts of building polyhedral models for affine programs with non-affine conditionals.

The foundations of dependence (data flow) analysis for arrays have been well established in the 90-s by Feautrier [4–6], Pugh [18], Collard and Griebl [3, 7], Maslov [15] and others [9, 16]. Their methods use the Omega test and Integer Programming libraries and yield an exact solution for dependence between any pair of read and write references in affine program. Thus in the pure affine case our work adds almost nothing more (except that we use the resulting polyhedral model further to produce a program in the dataflow computation model). However in the case of affine programs extended with non-affine conditions (the so-called dynamic control programs), the state-of-the-art is to yield in the general case a fuzzy solution [6]. It is fuzzy in the sense that the source function produces a set of possible sources, not the unique and exact source. The authors claim that it is the best that can be done. But it seems that the claim proceeds from assumption that the result should be represented in the form of the finite quasi-affine solution tree (quast). And as we have seen in Section 7.4, generally the source function can be represented as a finite or infinite quast, but it can always be represented as a finite S-graph.

Our base affine machinery of building the exact S-graph also differs. Whereas it is a common practice to build the polyhedral model by considering each read-

write pair independently, our method of building a dataflow model first produces effects and then states using only writes, and then resolves all reads against states. It is interesting to notice a similarity between our effect/state building process and the process of backward traversing the control flow graph presented in [3,7] which fail, in general, to produce exact (not fuzzy) results. Both processes are moving along the same path but in opposite directions. The authors usually argue for moving backward noticing that the process can stop when the total source is found (cf. also [15]). It is a good idea, and it can be incorporated into our algorithm simply by porting it to a lazy language, e.g. to Haskell, or by somehow emulating the laziness. In the lazy setting, the tree T will not be built at all in applications like $\mathsf{Seq}(T, t)$, where t is a term (or a $\perp$-less tree).

Speaking of parallelization, one must not forget about the distribution of computations in space and time. On this subject, there are many works in which an optimal (in terms of communications volume and load balancing) mapping in multidimensional space and time is sought, and then on its basis an inverse translation into a loop nest program with parallel loops is made [1, 7]. In our dataflow computational model the knowledge of the distribution functions, although not mandatory, can significantly improve the efficiency of execution. The project of a real multiprocessor that can directly execute the dataflow model is being developed in our institute IDPM RAS [2, 14, 20].

9 Conclusion

Our aim was to build the converter of a program P that belongs to a specific class into the dataflow computation model. Thus we need not only to build the exact and complete data flow model (which is usually referred to as the polyhedral model and comprises of exact source functions for each read operation in the program P), but also to invert it and thus obtain the exact use function for each write operation. The latter representation can be used as an equivalent program in a specific dataflow computation model, in which the maximum parallelism inherent to program P is exhibited. At the present time, the described machinery is implemented in a prototype translator which is written totally in the functional language Refal, version 6 [11]. Currently, it admits as input an arbitrary affine program extended with non-affine conditions in `if`-statements (provided that unlimited computing resources are available).

However, the intermediate source graph also appears interesting. It can also be treated as an independent semantic representation of the input program, namely, the SRE. Partially evaluating the SRE one can use it as an exact source function definition, that is evaluate the write statement of input program that wrote the value being read by the given read statement. Or one can produce a more refined form of the source function for a given read operation. Note that all this is possible for arbitrary affine programs with non-affine conditionals.

The work was supported by Russian Academy of Sciences Presidium Program for Fundamental Research "Fundamental Problems of System Programming" in 2009–2013.

References

1. Uday Bondhugula, Muthu Manikandan Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In Laurie J. Hendren, editor, *CC*, volume 4959 of *Lecture Notes in Computer Science*, pages 132–146. Springer, 2008.
2. V.S. Burtsev. "Vybor novoj sistemy organizacii vypolneniya vysokoparallelnyh vychislitelnyh processov, primery vozmozhnyh arhitekturnyh reshenij postroeniya superEVM" (The choice of a new organization system of execution of highly-parallel computation processes and examples of possible supercomputer architecture solutions). In V.S. Burtsev, editor, *Parallelizm vychislitelnyh processov i razvitie arhitektury superEVM*, pages 41–78. IVVS RAS, Moscow, 1997.
3. Jean-Francois Collard and Martin Griebl. A precise fixpoint reaching definition analysis for arrays. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, pages 286–302, London, UK, UK, 2000. Springer-Verlag.
4. Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, 1988.
5. Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.
6. Paul Feautrier. Array dataflow analysis. In Santosh Pande and Dharma P. Agrawal, editors, *Compiler Optimizations for Scalable Parallel Systems*, pages 173–219. Springer-Verlag New York, Inc., New York, NY, USA, 2001.
7. Martin Griebl. *Automatic parallelization of loop programs for distributed memory architectures*. Habilitation thesis, Department of Informatics and Mathematics, University of Passau, 2004.
8. S.A. Guda. Operations on the tree representations of piecewise quasi-affine functions. *"Informatika i ee primeneniya" (Informatics and its applications)*, 7(1):58–69, 2013.
9. Gautam Gupta and Sanjay V. Rajopadhye. The Z-polyhedral model. In Katherine A. Yelick and John M. Mellor-Crummey, editors, *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007*, pages 237–248. ACM, 2007.
10. Andrei V. Klimov. An approach to supercompilation for object-oriented languages: the Java Supercompiler case study. In *First International Workshop on Metacomputation in Russia, Proceedings. Pereslavl-Zalessky, Russia, July 2–5, 2008*, pages 43–53. Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2008.
11. Ark.V Klimov. Refal-6. URL: http://refal.net/ arklimov/refal6/index.html, 2004.
12. Ark.V. Klimov. The use of selection trees for describing states in parallelizing compiler. In *Proceedings of All-Russian Scientific Conference Scientific service in Internet*, pages 238–240. Moscow, MSU Press, 2009. URL: http://agora.guru.ru/abrau2009/pdf/238_NSSI_2009_Abrau-2009.pdf.
13. Ark.V. Klimov. Transforming affine nested loop programs to dataflow computation model. In *Ershov Informatics Conference, PSI Series, 8-th edition, Preliminary Proceedings, June, 27 July, 1*, pages 274–285, Akademgorodok, Novosibirsk, Russia, 2011.
14. Ark.V. Klimov, N.N. Levchenko, S.A. Okunev, and A.L. Stempkovsky. Supercomputers, memory hierarchy and dataflow computation model. *Program systems: theory and applications*, 5(1):15–36, 2014.

15. Vadim Maslov. Lazy array data-flow dependence analysis. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 311–325. ACM Press, 1994.
16. Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In David S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 1–14. ACM, 1991.
17. Andrei P. Nemytykh, Victoria Pinchik, and Valentin Turchin. A self-applicable supercompiler. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, volume 1110 of *Lecture Notes in Computer Science*, pages 233–252. Springer-Verlag, 1996.
18. William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In Joanne L. Martin, editor, *SC*, pages 4–13. IEEE Computer Society / ACM, 1991.
19. Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
20. A.L. Stempkovsky, N.N. Levchenko, S.A. Okunev, and V.V. Tsvetkov. Parallel dataflow computing system the further development of architecture and the structural organization of the computing system with automatic distribution of resources. *Informatsionnye tekhnologii*, (10):2–7, 2008.
21. Valentin F. Turchin. Program transformation by supercompilation. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 257–281. Springer, 1985.
22. Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 599–613. Springer, 2009.
23. V.V. Voevodin and Vl.V. Voevodin. *"Parallel'nyje vychislenija" (Parallel computations)*. BKhV-Peterburg, St. Petersburg, 2004.