

# Abstracting Linear Programs with Arrays into Linear Programs

—  
AI-Lab Technical Report  
University of Genova  
April 2005  
—

Alessandro Armando<sup>1</sup>, Massimo Benerecetti<sup>2</sup>, and Jacopo Mantovani<sup>1</sup>

<sup>1</sup> AI-Lab, DIST, Università di Genova, Italy

<sup>2</sup> Dip. di Scienze Fisiche, Università di Napoli “Federico II”, Italy

**Abstract.** We introduce a counterexample guided abstraction and refinement procedure for linear programs with arrays. Our procedure starts by abstracting away all array elements from the initial program and then it incrementally refines the abstract program by including array elements as suggested by the refinement process. While, in the worst case, the procedure may lead to the generation and analysis of the abstract program obtained by considering all the array elements of the concrete program, in many cases of interests only a few array elements suffice to successfully conclude the analysis. This is unlike other approaches in which the complexity of the analysis always increases with the size of the arrays involved in the input program. Moreover our analysis of arrays is precise, unlike other approaches that trade precision for efficiency and therefore may return false negatives.

## 1 Introduction

Counterexample-guided abstraction refinement is one of the leading approaches to software model checking (e.g. [1–3]). Generally speaking it consists of the following loop. Let  $P$  be the program to be model checked and let  $k = 0$ :

1. *Abstraction:* an initial abstract version  $\hat{P}_k$  of  $P$  is automatically built. By construction,  $\hat{P}_k$  has at least the same execution traces as  $P$ . Additionally,  $\hat{P}_k$  belongs to a family of programs for which an effective model checking procedure is available.
2. *Model Checking:*  $\hat{P}_k$  is model checked. If no error is detected, then the correctness of  $P$  is reported and the procedure halts. Otherwise (i.e. if an error is found in  $\hat{P}_k$ ), the existence of an error trace in  $P$  cannot be concluded as  $\hat{P}_k$  may contain traces that are not executable in  $P$ .
3. *Counterexample-guided Refinement:* The error trace detected by model checking  $\hat{P}_k$  is checked for feasibility in  $P$ . If the trace is found to be executable in  $P$ , then it is reported and the procedure halts. Otherwise a new program  $\hat{P}_{k+1}$  is computed such that it is at the same time (i) an abstraction of  $P$  and (ii) a refinement of  $\hat{P}_k$  in which the previous trace is no longer executable;  $k := k + 1$ , and go to 2.

In the SLAM project [4], abstract programs are Boolean Programs, i.e. imperative programs having the same control flow graph as the corresponding concrete programs they abstract and whose variables are restricted to range over boolean values only. The advantage of the approach is that effective model checking procedures for boolean programs exist (e.g. [5] and [6]). While the approach has proven very effective on specific application areas such as device drivers programming [2, 3], its effectiveness on other, more mundane classes of programs has to be ascertained. Notice that since the detection of a spurious execution trace leads to a new iteration of the check-and-refine loop, the efficiency of the approach depends in a critical way on the number of spurious execution traces allowed by the abstract program. Of course, the closer is the abstraction to the original program the smaller is the number of spurious execution traces that may be necessary to analyse.

In a previous paper [7] we proposed Linear Programs as a finer grained model for imperative programs and showed how the model checking procedure used in SLAM can be generalized to a model checking procedure for Linear Programs. Linear Programs differ from boolean programs in that program variables can range over a numeric domain  $\mathcal{D}$  (e.g.  $\mathbb{Z}$  or  $\mathbb{R}$ ); moreover, all conditions and assignments to variables involve linear expressions, i.e. expressions of the form  $c_0 + c_1 * x_1 + \dots + c_n * x_n$ , where  $c_0, \dots, c_n$  are numeric constants in  $\mathbb{Z}$  and  $x_1, \dots, x_n$  are program variables ranging over  $\mathcal{D}$ . Linear Programs are considerably more expressive than Boolean Programs and can explicitly encode complex correlations between data and control that must necessarily be abstracted away when using boolean programs.

In this paper we introduce a model checking procedure for linear programs with arrays (i.e. linear programs augmented with references to array elements of numeric type) based on the counterexample-guided abstraction refinement paradigm. Our procedure starts the analysis by abstracting away all array elements from the initial program and then it incrementally refines the abstract program by including array elements as suggested by the refinement process. While, in the worst case, the procedure may lead to the generation and analysis of the abstract program obtained by considering all the array elements of the concrete program, in many cases of interests only few array elements suffice to successfully conclude the analysis. This is unlike other approaches (as, e.g., in CBMC [8]) in which the complexity of the analysis always increases with the size of the arrays involved in the input program. Moreover our analysis of arrays is precise, unlike other approaches that trade precision for efficiency (as, e.g., SLAM and Blast [3]) and therefore may return false negatives.

The usage of our model checking procedure for linear programs in the context of the model checking procedure for linear programs with arrays presented in this paper requires two important extensions to linear programs: (i) the addition of a symbol for undefined values and (ii) support for conditional expressions. In this paper we provide a formal semantics (Section 3), we show in great detail the abstraction process (Section 4), we adapt the procedure for interprocedural data-flow analysis inspired by Bebop to Linear Programs, (Section 5), and we provide a detailed account of a symbolic model checking procedure which is amenable of mechanization (Section 6). We conclude (in Section 7) by discussing the implementation of our ideas in the EUREKA [7] tool and by presenting preliminary experimental results that confirm their effectiveness.

## 2 Counterexample-guided Abstraction-Refinement of Linear Programs with Arrays

We illustrate our counterexample-guided abstraction refinement procedure by showing its application to the linear program with arrays  $P$  of Figure 1.<sup>3</sup> Program  $P$  is abstracted into program  $\hat{P}_0$  by replacing

<sup>3</sup> The program is slightly simplified for readability.

$P$	$\hat{P}_0$	$\hat{P}_1$
<pre> void main() {   int i,a[3]; [1] a[1] = 1; [2] i = 0; [3] while(a[i]!=1&amp;&amp; i&lt;3) [4]   { a[i] = 2*i; [5]     i = i+1; } [6] if(i &gt; 1) [7]   ERROR: ; } </pre>	<pre> void main() {   int i; [1] ; [2] i = 0; [3] while(u!=1&amp;&amp; i&lt;3) [4]   { ; [5]     i = i+1; } [6] if(i &gt; 1) [7]   ERROR: ; } </pre>	<pre> void main() {   int i,a<sub>1</sub>; [1] a<sub>1</sub> = (1==1)?1:u; [2] i = 0; [3] while((i==1)?a<sub>1</sub>:u)!=1 &amp;&amp; i&lt;3) [4]   { a<sub>1</sub> = (i==1)?2*i:a<sub>1</sub>; [5]     i = i+1; } [6] if(i &gt; 1) [7]   ERROR: ; } </pre>

**Fig. 1.** A simple program ( $P$ ), the initial abstraction ( $\hat{P}_0$ ), and its refinement ( $\hat{P}_1$ ).

every occurrence of array expressions with the symbol  $u$  (denoting an arbitrary value of numeric type) and by replacing every assignment to array elements with a skip statement ( $;$ ).

A model checker for linear programs is then applied to  $\hat{P}_0$  to determine the reachability of the line labelled with **ERROR** label. The trace 1, 2, 3, 4, 5, 3, 4, 5, 3, 4, 5, 3, 6, 7 is detected by model checker. (How this is done is explained in the following sections.) This trace corresponds to executing three iterations of the **while** loop (lines [3]–[5]) which leaves variable  $i$  with the value 3. Therefore the condition of the **if** statement at line [6] evaluates to true and the line labelled **ERROR** is finally reached.

The feasibility check of the above trace w.r.t.  $P$  is done by generating a set of formulae whose satisfying valuations correspond to all the possible executions of the sequence of statements of  $P$  corresponding to the trace under consideration. This is done by first renaming the variables occurring in the statements

**Table 1.** Checking the trace for feasibility.

Step	Line	Original Statement	Renamed Statement	Formula
1	[1]	$a[1] = 1;$	$a_1[1] = 1;$	$a_1 = \text{store}(a_0, 1, 1)$
2	[2]	$i = 0;$	$i_2 = 0;$	$i_2 = 0$
3	[3]	$\text{assume}(a[i] \neq 1 \ \&\& \ i < 3);$	$\text{assume}(a_1[i_2] \neq 1 \ \&\& \ i_2 < 3);$	$a_1[i_2] \neq 1 \wedge i_2 < 3$
4	[4]	$a[i] = 2 * i;$	$a_4[i_2] = 2 * i_2;$	$a_4 = \text{store}(a_1, i_2, 2 * i_2)$
5	[5]	$i = i + 1;$	$i_5 = i_2 + 1;$	$i_5 = i_2 + 1$
6	[3]	$\text{assume}(a[i] \neq 1 \ \&\& \ i < 3);$	$\text{assume}(a_4[i_5] \neq 1 \ \&\& \ i_5 < 3);$	$a_4[i_5] \neq 1 \wedge i_5 < 3$
7	[4]	$a[i] = 2 * i;$	$a_7[i_5] = 2 * i_5;$	$a_7 = \text{store}(a_4, i_5, 2 * i_5)$
8	[5]	$i = i + 1;$	$i_8 = i_5 + 1;$	$i_8 = i_5 + 1$
9	[3]	$\text{assume}(a[i] \neq 1 \ \&\& \ i < 3);$	$\text{assume}(a_7[i_8] \neq 1 \ \&\& \ i_8 < 3);$	$a_7[i_8] \neq 1 \wedge i_8 < 3$
10	[4]	$a[i] = 2 * i;$	$a_{10}[i_8] = 2 * i_8;$	$a_{10} = \text{store}(a_7, i_8, 2 * i_8)$
11	[5]	$i = i + 1;$	$i_{11} = i_8 + 1;$	$i_{11} = i_8 + 1$
12	[6]	$\text{assume}(!(a[i] \neq 1 \ \&\& \ i < 3));$	$\text{assume}!(a_{10}[i_{11}] \neq 1 \ \&\& \ i_{11} < 3);$	$\neg(a_{10}[i_{11}] \neq 1 \wedge i_{11} < 3)$
13	[6]	$\text{assume}(i > 1);$	$\text{assume}(i_{11} > 1);$	$i_{11} > 1$

in such a way that no variable is assigned twice during execution (as done in [8, 9]) and then by generating formulae encoding the behaviour of the renamed statements. Table 1 shows the sequence of original statements, the renamed statements, and the associated formulae for the trace above. Notice that, since a trace does not allow branching, **while** (as well as **if**) statements are replaced by assumptions (**assume**) of the positive or negated corresponding guard, depending on whether the trace proceeds along the *true* or the *false* branch of the conditional statement. The resulting set of formulae is then fed to a theorem prover. If

it is found unsatisfiable then the trace is not executable in  $P$ , whereas if it is found satisfiable then we can conclude that the trace is also executable in  $P$ . In our example the set of formulae (see rightmost column in Table 1) is found to be unsatisfiable. The formulae that contributed to the proof of unsatisfiability are those associated with steps 1, 2, 4, 5, and 6. Notice the term  $a_4[i_5]$  (with  $i_5 = 1$  given by the context) occurs in the formula associated with step 6. This means that in order to rule out the above trace we must refine  $\hat{P}_0$  by including the element of  $\mathbf{a}$  at position 1 thereby obtaining  $\hat{P}_1$ . In the general case if  $k_1, \dots, k_n$  are the indexes for array  $a$  to be included in the refined program, then the refinement step amounts to replacing every expression of the form  $a[e]$  with the conditional expression  $(e == k_1 ? a_{k_1} : (e == k_2 ? a_{k_2} : \dots (e == k_n ? a_{k_n} : \mathbf{u}) \dots))$ , and then every assignment of the form  $a[e_1] = e_2$ ; with the (parallel) assignment

$$a_{k_1}, \dots, a_{k_n} = (e_1 == k_1 ? e_2 : a_{k_1}), \dots, (e_1 == k_n ? e_2 : a_{k_n});$$

where  $a_{k_1}, \dots, a_{k_n}$  are new variables of numeric type. (If  $n = 0$ , the assignment above reduces to a skip (;) statement.). For more details see section 4. The application of the model checking procedure for linear programs to  $\hat{P}_1$  reveals that the error statement cannot be reached in  $\hat{P}_1$  and from that we can conclude that the error statement is not reachable in  $P$ .

### 3 Linear Programs and Linear Programs with Arrays

Let  $W$  be a set of program variables. Let  $V = \{v_1, \dots, v_l\} \subseteq W$  be a set of program variables ranging over  $\mathcal{D}$  and let  $A = \{a_1, \dots, a_m\} \subseteq W$  be a set of array variables. An *array variable*  $a \in A$  is a program variable equipped with a positive integer  $\dim(a)$ , the *size of the array*. The sets  $A$  and  $V$  form a partition of  $W$ , i.e.  $W = V \cup A$  and  $V \cap A = \emptyset$ . Moreover, let  $\mathbf{u}$  be a distinct symbol standing for undefined. A *generalized linear expression with arrays* (*linear expression with arrays*, for short) is inductively defined to be:

1. a constant symbol  $\mathbf{u}$ ;
2. a numeric constant  $c \in \mathbb{Z}$ ;
3. a numeric program variable  $v_j \in V$  ( $1 \leq j \leq l$ );
4. an expression of the form  $c * e$  where  $c \in \mathbb{Z}$  and  $e$  is a generalized linear expression with arrays;
5. an expression of the form  $e_1 + e_2$  with where  $e_1$  and  $e_2$  are generalized linear expressions with arrays;
6. an expression of the form  $(b ? e_1 : e_2)$ , called *conditional expression*, where  $b$  is a generalized boolean or linear expression with arrays, and  $e_1$  and  $e_2$  are generalized linear expressions with arrays.
7. an expression of the form  $a[e]$  where  $a \in A$  and  $e$  is a generalized linear expression with arrays;

A *generalized boolean linear expression with arrays* is an expression of the form  $(e_1 \text{ op } e_2)$ , where  $e_1, e_2$  are generalized linear expressions with arrays and  $\text{op} \in \{>=, =, <, >, ==, !=\}$ . The definition of *linear expression without arrays* (*linear expression*, for short) is subsumed by the above. In the following,  $E_W$  will denote the set of generalized linear expressions with arrays over  $W$ . A *linear program (with arrays)* is a program with the usual control-flow constructs (**if**, **while**, **assert**) and procedural abstraction with call-by-value parameter passing and recursion. Variables range over  $\mathcal{D}$ ; moreover, all conditions and assignments to variables involve linear expressions (with arrays, resp.). We denote the set of numeric variables and array variables of  $P$  with  $V_P$  and  $A_P$ , respectively.

We assume that every occurrence of  $(! b_1)$ ,  $(b_1 \ \&\& \ b_2)$ , and  $(b_1 \ || \ b_2)$  in a program is replaced by the equivalent expression  $(b_1 ? 0 : 1)$ ,  $(b_1 ? b_2 : 0)$ , and  $(b_1 ? 1 : b_2)$  respectively. Moreover, every boolean

linear expression  $b$  occurring outside the guard of a conditional expression, is replaced by the following equivalent linear expression  $(b ? 1 : 0)$ . Therefore, after the rewriting, boolean expressions will only occur within the guards of conditional expressions. We also assume that a skip statement  $(;)$  is added immediately after each procedure call in the original program. In this way the return point of a procedure call is unique, explicit, and labelled within the control flow graph of the program. Finally, without loss of generality, we will assume that all variable names are globally unique.

The *control flow graph* of a program  $P$  is a directed graph  $G_P = (N_P, \text{Succ}_P)$ , where  $N_P = \{0, 1, \dots, n, n+1, n+p\}$  is the set of vertices<sup>4</sup> and  $\text{Succ}_P : N_P \rightarrow 2^{N_P}$  is the usual successor function that maps each vertex, say  $i$ , in the set of its successors  $\text{Succ}_P(i)$  (see [10]). For every vertex  $i$  such that  $1 \leq i \leq n$ , let  $s_i$  denote the program statement that corresponds to vertex  $i$  of  $G_P$ . If  $s_i$  is **if**( $e$ ), **while**( $e$ ), or **assert**( $e$ ) then  $\text{Succ}_P(i) = \{\text{Tsucc}_P(i), \text{Fsucc}_P(i)\}$ , where  $\text{Tsucc}_P(i)$  ( $\text{Fsucc}_P(i)$ ) denotes the successor of  $i$  when  $e$  evaluates to true (false, resp.). If  $s_i$  is **assert**( $e$ ), then  $\text{Fsucc}_P(i) = 0$ . If  $pr$  is a procedure in  $P$ , then  $\text{First}_P(pr)$  is the vertex corresponding to the first statement in  $pr$ , and  $\text{Exit}_P(pr)$  is the exit vertex of  $pr$ . If  $s_i$  is a procedure call  $pr(e)$ ; then  $\text{Succ}_P(i) = \{\text{First}_P(pr)\}$  and  $\text{RetPt}_P(i)$  is the vertex of the  $;$  statement immediately following the procedure call. Moreover, if  $s_i$  is a statement occurring in the body of a procedure  $pr$ , then  $\text{ProcOf}_P(i) = pr$ . If  $s_i$  is a return statement in a procedure  $pr$ , then  $\text{Succ}_P(i) = \{\text{Exit}_P(pr)\}$ , and  $\text{Succ}_P(\text{Exit}_P(pr)) = \{j \in N_P : s_j = pr(e); \text{ and } j = \text{RetPt}_P(i)\}$ . Finally, if  $\text{Succ}_P(i_1) = \{i_2\}$ , we define  $\text{sSucc}_P(i_1) = i_2$ . We also assume the existence of a procedure called **main**: it is the first procedure to be executed. Moreover, without loss of generality, we assume that  $\text{First}_P(\text{main}) = 1$ .

We define  $\text{Globals}_P$  to be the set of global variables. Let  $i \in N_P$ , then by  $\text{Formals}_P(i)$  and  $\text{Locals}_P(i)$  we indicate the set of formal parameters of the procedure containing  $i$  and the set of the local variables in scope at vertex  $i$ . Therefore  $\text{Formals}_P(i) \subseteq \text{Locals}_P(i)$  for all  $i \in N_P$ . Moreover we define  $\text{InScope}_P(i) = \text{Globals}_P \cup \text{Locals}_P(i)$ .

Let  $W$  be the set of numeric variables and array variables, a *valuation*  $\omega$  over  $W$  is a total function mapping the numeric variables in  $V \subseteq W$  into  $\mathcal{D}$  and each array variable  $a \in A \subseteq W$  into a finite mapping from  $\{0, \dots, \text{dim}(a) - 1\}$  into  $\mathcal{D}$ . A *state of a linear program with arrays*  $P$  is a pair  $\langle i, \omega \rangle$ , where  $i$  is a vertex of the control flow graph of  $P$  and  $\omega$  is a valuation over  $W \cap \text{InScope}_P(i)$ . Thus,  $\omega$  is a total function over  $\text{InScope}_P(i)$ . Again, the definition of *state of a linear program* is subsumed by the above. We lift  $\omega$  to a total function  $\bar{\omega} : E_W \rightarrow 2^{\mathcal{D}}$  over linear expressions with arrays defined as follows:

$$\bar{\omega}(e) = \begin{cases} \{e\} & \text{if } e \in \mathbb{Z} \\ \{\omega(e)\} & \text{if } e \in V \\ \{d \in \omega(a)(d_1) : d_1 \in \bar{\omega}(e_1)\} & \text{if } e = a[e_1] \text{ and } \bar{\omega}(e_1) \subseteq \{0, \dots, \text{dim}(a) - 1\} \\ \{c \cdot d_1 : d_1 \in \bar{\omega}(e_1)\} & \text{if } e = c * e_1 \\ \{d_1 \text{ op } d_2 : d_1 \in \bar{\omega}(e_1) \\ \text{and } d_2 \in \bar{\omega}(e_2)\} & \text{if } e = e_1 \text{ op } e_2 \text{ with } \text{op} \in \{>=, <=, <, >, \\ & \text{==, !=, +}\} \\ \bar{\omega}(e_1) \cup \bar{\omega}(e_2) & \text{if } e = (bc ? e_1 : e_2) \text{ and } \{0, d\} \subseteq \bar{\omega}(b) \text{ for} \\ & \text{some } d \neq 0 \\ \bar{\omega}(e_1) & \text{if } e = (b ? e_1 : e_2) \text{ and } 0 \notin \bar{\omega}(b) \\ \bar{\omega}(e_2) & \text{if } e = (b ? e_1 : e_2) \text{ and } \bar{\omega}(b) = \{0\} \\ \mathcal{D} & \text{otherwise} \end{cases}$$

<sup>4</sup> Vertices from 1 to  $n$  are associated with program statements, vertex 0 models the failure of **assert** statements, and vertices from  $n+1$  to  $n+p$  are the “exit” vertices of procedures, where  $p$  is the number of procedures defined in  $P$ .

The intuition is that  $\bar{\omega}(e)$  collects the possible values for a linear expression (with arrays)  $e$  compatible with a valuation  $\omega$ , accounting for the nondeterminism due to the undefined values. In other words, the occurrence of a symbol  $u$ , and similarly the occurrence of an out-of-range access to an array within an expression is modeled by nondeterministically assigning to the corresponding subexpression an arbitrary element in the domain  $\mathcal{D}$ .  $\bar{\omega}$  can be extended to  $k$ -uples  $\mathbf{e}$  of expressions in the obvious way.

State transitions in  $P$  are denoted by  $\langle i_1, \omega_1 \rangle \xrightarrow{\sigma} \langle i_2, \omega_2 \rangle$  where  $\sigma$  is either the symbol  $\epsilon$  or an expression of the form  $\text{CALL}(i, \omega)$  or  $\text{RET}(i, \omega)$  for  $i \in N_P$  such that there exists a procedure call  $s_j$  with  $i = \text{RetPt}(j)$  and  $\omega : \text{Locals}_P(j) \rightarrow \mathcal{D}$  (Terminals of the form  $\text{CALL}(i, \omega)$  and  $\text{RET}(i, \omega)$  represent the entry and exit points of the procedure invoked by  $s_j$  respectively). In the following we will use bold letters such as  $\mathbf{x}$  to denote a vector of variables, elements or expressions. We also allow for parallel assignment, denoted by  $\mathbf{x} = \mathbf{e}$ ; . Moreover, let  $\mathbf{c} = \langle c_1, c_2, \dots, c_n \rangle$  and  $\mathbf{d} = \langle d_1, d_2, \dots, d_n \rangle$  be  $n$ -uples of values in a set  $X$  and in  $\mathcal{D}$  respectively; for any function  $f : X \rightarrow \mathcal{D}$  we denote  $f[\mathbf{d}/\mathbf{c}]$  the function  $f'$  such that  $f'(c_k) = d_k$  for all  $k = 1, 2, \dots, n$ , and  $f'(c) = f(c)$  for all  $c \neq c_k$  and  $k = 1, 2, \dots, n$ .

State transitions are defined as follows:

- if  $i_1$  corresponds to a skip ( $;$ ), or a **return**, then  $\langle i_1, \omega_1 \rangle \xrightarrow{\epsilon} \langle \text{sSucc}_P(i_1), \omega_1 \rangle$ ;
- if  $i_1$  corresponds to the parallel assignment  $\mathbf{y} = \mathbf{e}$ ; then  $\langle i_1, \omega_1 \rangle \xrightarrow{\epsilon} \langle \text{sSucc}_P(i_1), \omega_1[\mathbf{d}/\mathbf{y}] \rangle$  for  $\mathbf{d} \in \bar{\omega}_1(\mathbf{e})$ ;
- if  $i_1$  corresponds to an assignment  $a[e_1] = e_2$ ; then  $\langle i_1, \omega_1 \rangle \xrightarrow{\epsilon} \langle \text{sSucc}_P(i_1), \omega_1[(\omega(a)[d_2/d_1])/a] \rangle$  for  $d_1 \in \bar{\omega}_1(e_1)$  and  $d_2 \in \bar{\omega}_1(e_2)$ ;
- if  $i_1$  corresponds to **if**( $e$ ), **while**( $e$ ), or **assert**( $e$ ), then  $\langle i_1, \omega_1 \rangle \xrightarrow{\epsilon} \langle i_2, \omega_1 \rangle$ , where
  - $i_2 = \text{Fsucc}_P(i_1)$  if  $0 \in \bar{\omega}_1(e)$  and
  - $i_2 = \text{Tsucc}_P(i_1)$  if  $d \in \bar{\omega}_1(e)$  for some  $d \neq 0$ ;
- if  $i_1$  corresponds to  $pr(\mathbf{e})$ ; then  $\langle i_1, \omega_1 \rangle \xrightarrow{\text{CALL}(\text{RetPt}_P(i_1), \omega)} \langle \text{First}_P(pr), \omega_2 \rangle$ , where  $\omega : \text{Locals}_P(i_1) \rightarrow \mathcal{D}$  is such that  $\omega(\mathbf{x}) = \omega_1(\mathbf{x})$ ,  $\omega_2(\mathbf{y}) \in \bar{\omega}_1(\mathbf{e})$ , and  $\omega_2(\mathbf{g}) = \omega_1(\mathbf{g})$  where  $\mathbf{x} = \text{Locals}_P(i_1)$ ,  $\mathbf{y} = \text{Formals}_P(i_2)$ , and  $\mathbf{g} = \text{Globals}_P$ ;
- if  $i_1$  is  $\text{Exit}_{pr}$  then  $\langle i_1, \omega_1 \rangle \xrightarrow{\text{RET}(i_2, \omega)} \langle i_2, \omega_2 \rangle$ , where  $i_2 \in \text{Succ}_P(i_1)$ ,  $\omega_2(\mathbf{g}) = \omega_1(\mathbf{g})$  for all  $\mathbf{g} = \text{Globals}_P$  and  $\omega_2(\mathbf{x}) = \omega(\mathbf{x})$  for all  $\mathbf{x} = \text{Locals}_P(i_2)$ .

A *path* is a sequence  $\langle i_0, \omega_0 \rangle \xrightarrow{\sigma_1} \langle i_1, \omega_1 \rangle \xrightarrow{\sigma_2} \dots \xrightarrow{\sigma_n} \langle i_n, \omega_n \rangle$  such that  $\langle i_k, \omega_k \rangle \xrightarrow{\sigma_{k+1}} \langle i_{k+1}, \omega_{k+1} \rangle$  for  $k = 0, \dots, n-1$ .<sup>5</sup> Notice that not all paths represent potential execution paths: in a transition like  $\langle i_1, \omega_1 \rangle \xrightarrow{\text{RET}(i_2, \omega)} \langle i_2, \omega_2 \rangle$  where  $i_1 = \text{Exit}_{pr}$ , the valuation  $\omega$  can be chosen arbitrarily and therefore  $\omega_2$  is not guaranteed to coincide with  $\omega_1$  on the locals of the caller, as required by the semantics of procedure calls. To rectify this, the notion of same-level valid path is introduced. A *valid path from*  $\langle i_0, \omega_0 \rangle$  *to*  $\langle i_n, \omega_n \rangle$  describes the transmission of effects from  $\langle i_0, \omega_0 \rangle$  to  $\langle i_n, \omega_n \rangle$  via a sequence of execution steps which may end with some number of activation records on the call stack; these correspond to “unmatched” terminals of the form  $\text{CALL}(i, \omega)$  in the string associated with the path. A *same-level valid path from*  $\langle i_0, \omega_0 \rangle$  *to*  $\langle i_n, \omega_n \rangle$  describes the transmission of effects from  $\langle i_0, \omega_0 \rangle$  to  $\langle i_n, \omega_n \rangle$ —where  $\langle i_0, \omega_0 \rangle$  and  $\langle i_n, \omega_n \rangle$  are in the same procedure—via a sequence of execution steps during which the call stack may temporarily grow deeper (because of procedure calls) but never shallower than its original depth, before eventually returning to its original depth.<sup>6</sup>

<sup>5</sup> In the sequel we will abbreviate  $\langle i_0, \omega_0 \rangle \xrightarrow{\sigma_1} \langle i_1, \omega_1 \rangle \xrightarrow{\sigma_2} \dots \xrightarrow{\sigma_n} \langle i_n, \omega_n \rangle$  with  $\langle i_0, \omega_0 \rangle \xrightarrow{\Sigma_0^n} \langle i_n, \omega_n \rangle$ , where  $\Sigma_0^n = \sigma_1 \sigma_2 \dots \sigma_n$ .

<sup>6</sup> Precise definitions of valid path and same-level valid path can be found in [11, 5].

An *initialized path* is a path whose initial state is  $\langle 1, \omega_0 \rangle$  for some  $\omega_0$ . A state  $\langle i, \omega \rangle$  is *reachable* if and only if there exists an initialized valid path to  $\langle i, \omega \rangle$ . A vertex  $i \in N_P$  is *reachable* if and only if there exists a valuation  $\omega$  such that  $\langle i, \omega \rangle$  is reachable.

## 4 Abstracting Linear Programs with Arrays into Linear Programs

Let  $P$  be a linear program with arrays and let  $\{R(a)\}_{A_P}$  be a family of sets of indexes such that  $R(a) \subseteq \{0, \dots, \dim(a) - 1\}$  for all  $a \in A_P$ . Let  $\widehat{V} \subseteq V_P$  be a subset of the variables of  $P$ . The *abstraction of  $P$  w.r.t.  $\{R(a)\}_{A_P}$  and  $\widehat{V}$*  is the linear program  $\widehat{P}$  defined as follows. The set of program variables of  $\widehat{P}$  is  $V_{\widehat{P}} = \widehat{V} \cup \{a_k : a \in A, k \in R(a)\}$ . Intuitively,  $a_k$  is a new variable representing in  $\widehat{P}$  the  $(k+1)$ -th element of the array  $a$ , and  $\widehat{V}$  is a subset of the variables of  $P$ . Let  $R(a) = \{k_1, \dots, k_n\}$  with  $k_i < k_{i+1}$  for  $i = 1, \dots, n-1$ . Given any linear expression  $e$  in  $P$ , its abstract version  $\widehat{e}$  is obtained from  $e$  by replacing every occurrence of a variable not belonging to  $\widehat{V}$  with the undefined symbol  $\mathbf{u}$ , and every occurrence of expressions of the form  $a[e]$  with  $\text{abs}(a[e], [k_1, \dots, k_n])$ , where:

$$\begin{aligned} \text{abs}(a[e], []) &= \mathbf{u} \\ \text{abs}(a[e], [k_1, k_2, \dots, k_n]) &= (\widehat{e} == k_1 ? a_{k_1} : \text{abs}(a[e], [k_2, \dots, k_n])). \end{aligned}$$

The linear program  $\widehat{P}$  is obtained from  $P$  by replacing all the expressions  $e$  occurring in  $P$  with  $\widehat{e}$ , and then by replacing each assignment of the form  $x = e$ ; with the skip statement  $(;)$  if  $x \notin \widehat{V}$ , with  $x = \widehat{e}$ ; otherwise, and by replacing each assignment of the form  $a[e_1] = e_2$ ; with the (parallel) assignment

$$a_{k_1}, \dots, a_{k_n} = (\widehat{e}_1 == k_1 ? \widehat{e}_2 : a_{k_1}), \dots, (\widehat{e}_1 == k_n ? \widehat{e}_2 : a_{k_n});$$

If  $n = 0$  (i.e. if  $R(a) = \emptyset$ ), the assignment above reduces to a skip  $(;)$  statement.

Let  $S_P$  be the set of states of program  $P$ . The *abstraction of  $\omega$  w.r.t.  $\{R(a)\}_{A_P}$*  is the valuation  $\widehat{\omega}$  over  $V_{\widehat{P}} = \widehat{V} \cup \{a_k : a \in A_P, k \in R(a)\}$ , such that  $\widehat{\omega}(v_i) = \omega(v_i)$ , for all  $v_i \in \widehat{V}$  and  $\widehat{\omega}(a_j) = d \in \overline{\omega}(a[j])$  for all  $a \in A_P$  and  $j \in \{R(a)\}_{A_P}$ . The abstraction of valuations is then lifted to abstraction of states by means of the function  $h : S_P \rightarrow S_{\widehat{P}}$  such that  $h(\langle i, \omega \rangle) = \langle i, \widehat{\omega} \rangle$  for all  $\langle i, \omega \rangle \in S_P$ . Notice that  $h$  is both total and surjective.

Let  $S_1 \subseteq S_P$ . We define the abstraction  $\alpha(S_1) = \{h(s) \mid s \in S_1\}$ ; conversely, for all  $\widehat{S}_1 \subseteq S_{\widehat{P}}$ , the concretization is defined by  $\gamma(\widehat{S}_1) = \{s \in S_P \mid h(s) \in \widehat{S}_1\}$ .

**Lemma 1.** *The pair  $\langle \alpha, \gamma \rangle$  forms a Galois connection  $\langle 2^{S_P}, \subseteq \rangle \xleftrightarrow[\alpha]{\gamma} \langle 2^{S_{\widehat{P}}}, \subseteq \rangle$ .*

*Proof.* We need to prove that  $\alpha, \gamma$  satisfy the fundamental property that  $\forall X \in S, \forall Y \in \widehat{S}, \alpha(X) \subseteq Y \Leftrightarrow X \subseteq \gamma(Y)$ .

- Given  $X \subseteq S, Y \subseteq \widehat{S}$ , we want to prove that  $\alpha(X) = Y \Leftrightarrow X = \gamma(Y)$ .
  - If  $\alpha(X) = \{h(s) : s \in X\} = Y$ , then for every  $h(s)$  in  $Y$ ,  $s$  is in  $X$ . Hence, by definition of  $\gamma$ ,  $\gamma(Y) = X$ .
  - If  $\gamma(Y) = \{s \in S : h(s) \in Y\} = X$ , then every  $s$  in  $S$  is also in  $X$ . Hence, by definition of  $\alpha$ ,  $\alpha(X) = Y$ .
- Now we only need to prove that  $\alpha(X) \subset Y \Leftrightarrow X \subset \gamma(Y)$ .  $Y$  contains at least one element  $b : b = h(a)$  with  $a \in S, b \in \widehat{S}$  and  $a \notin X$  iff (by definition of  $\gamma$ ) the elements of  $\gamma(Y)$  are the ones of  $X$  plus at least one element,  $a \in S \setminus X$ .

Hence the thesis.

The following result is preliminary to the proof of soundness, and states the relation between abstract and concrete valuations.

**Lemma 2.** *Let  $e$  be a linear expression with arrays built out of the numeric variables  $V$  and array variables  $A$ . Let  $\{R(a)\}_A$  be a family of sets of indexes such that  $R(a) \subseteq \{0, \dots, \dim(a) - 1\}$  for all  $a \in A$  and  $\widehat{V} \subseteq V_P$ . Let  $\widehat{e}$  be the linear expression obtained from  $e$  by replacing all the variables not in  $\widehat{V}$  with  $\mathbf{u}$  and all the expressions of the form  $a[i]$  with  $\text{abs}(a[i], [k_1, \dots, k_n])$ , where  $R(a) = \{k_1, \dots, k_n\}$ . Then,  $\overline{\omega}(e) \subseteq \overline{\omega}(\widehat{e})$ , for all valuations  $\omega$  over  $V$  and  $A$ .*

*Proof.* The proof proceeds by induction on the structure of the concrete expression  $e$ .

**Base step**  $e \in V_P \cup \mathcal{D}$ . If, in addition,  $e \in \widehat{V} \cup \mathcal{D}$ , then  $\widehat{e} = e$ , while if  $e \in V_P \setminus \widehat{V}$ , then  $\widehat{e} = \mathbf{u}$ . Moreover, if  $e \in \widehat{V}$ , from the definition of  $\widehat{\omega}$ , we have  $\widehat{\omega}(e) = \omega(e) \in \mathcal{D}$ , and  $\overline{\omega}(e) = \{\omega(e)\} = \overline{\omega}(\widehat{e})$ . If  $e \in \mathcal{D}$ ,  $\overline{\omega}(e) = \{e\} = \overline{\omega}(\widehat{e})$ . Finally, if  $e \in V_P \setminus \widehat{V}$ ,  $\overline{\omega}(e) \subseteq \mathcal{D} = \overline{\omega}(\widehat{e})$ . The thesis immediately follows;

**Inductive step** assume the thesis holds for every sub-expression  $e'$  occurring in  $e$ . Then we have the following cases:

1.  $e = a[i]$ . Then  $\widehat{e} = \text{abs}(a[i], [k_1, \dots, k_n])$ . We consider the following two cases:
  - (a)  $\overline{\omega}(\widehat{i}) \subseteq \{k_1, \dots, k_n\}$ . By inductive hypothesis,  $\overline{\omega}(i) \subseteq \overline{\omega}(\widehat{i})$ . By the definitions of  $\overline{\omega}$ ,  $\widehat{\omega}$  and  $\text{abs}(a[i], [k_1, \dots, k_n])$ , we have that

$$\overline{\omega}(a[i]) = \{\omega(a)(k) \mid k \in \overline{\omega}(i)\}$$

and

$$\widehat{\omega}(\text{abs}(a[i], [k_1, \dots, k_n])) = \{\widehat{\omega}(a_k) \mid k \in \overline{\omega}(\widehat{i})\}$$

Since  $\omega(a)(k) = \widehat{\omega}(a_k)$ , for any  $k \in \{k_1, \dots, k_n\}$ , and  $\overline{\omega}(i) \subseteq \overline{\omega}(\widehat{i}) \subseteq \{k_1, \dots, k_n\}$ , it immediately follows that  $\overline{\omega}(a[i]) \subseteq \widehat{\omega}(\text{abs}(a[i], [k_1, \dots, k_n]))$ . Hence the thesis;

- (b)  $\overline{\omega}(\widehat{i}) \not\subseteq \{k_1, \dots, k_n\}$ . Thus, there must be a  $k \notin \{k_1, \dots, k_n\}$ , with  $k \in \overline{\omega}(\widehat{i})$ . Therefore,  $0 \in \overline{\omega}(\widehat{i})$  (for all  $k' \in \{k_1, \dots, k_n\}$ ). By the definitions of  $\text{abs}(a[i], [k_1, \dots, k_n])$  and  $\widehat{\omega}(\cdot)$ , we have  $\widehat{\omega}(\mathbf{u}) \subseteq \widehat{\omega}(\text{abs}(a[i], [k_1, \dots, k_n])) = \widehat{\omega}(\widehat{e}) \subseteq \mathcal{D}$ . In other words,  $\overline{\omega}(\widehat{e}) = \mathcal{D}$ . Since  $\overline{\omega}(a[i]) \subseteq \mathcal{D}$ , the thesis follows immediately.
2.  $e = e_1 \text{ op } e_2$  (where  $\text{op} \in \{>=, =, <, >, ==, !=, *, +\}$ ). Then,  $\widehat{e} = \widehat{e}_1 \text{ op } \widehat{e}_2$ . The thesis immediately follows from the inductive hypothesis and the definitions of  $\overline{\omega}(\cdot)$  and  $\widehat{\omega}(\cdot)$ .
3.  $e = (b ? e_1 : e_2)$ , then  $\widehat{e} = (\widehat{b} ? \widehat{e}_1 : \widehat{e}_2)$ . By inductive hypothesis, we have that  $\overline{\omega}(b) \subseteq \overline{\omega}(\widehat{b})$ ,  $\overline{\omega}(e_1) \subseteq \overline{\omega}(\widehat{e}_1)$ , and  $\overline{\omega}(e_2) \subseteq \overline{\omega}(\widehat{e}_2)$ . One of the following cases occurs:
  - (a)  $\overline{\omega}(b) = \{0\}$ . Therefore,  $\overline{\omega}(e) = \overline{\omega}(e_2)$ . Moreover,  $\{0\} \subseteq \overline{\omega}(\widehat{b})$  and  $\overline{\omega}(\widehat{e}_2) \subseteq \overline{\omega}(\widehat{e})$ . Hence, by the definitions of  $\overline{\omega}$  and  $\widehat{\omega}$ ,  $\overline{\omega}(e) = \overline{\omega}(e_2) \subseteq \overline{\omega}(\widehat{e}_2) \subseteq \overline{\omega}(\widehat{e})$ .
  - (b)  $\overline{\omega}(b) \subseteq \mathcal{D} \setminus \{0\}$  and  $\overline{\omega}(e) = \overline{\omega}(e_1)$ . Therefore, either  $\overline{\omega}(\widehat{b}) \subseteq \mathcal{D} \setminus \{0\}$  and  $\overline{\omega}(\widehat{e}) = \overline{\omega}(\widehat{e}_1)$ , or  $\overline{\omega}(\widehat{b}) \cap \mathcal{D} \setminus \{0\} \neq \emptyset$  and  $\overline{\omega}(\widehat{e}) = \overline{\omega}(\widehat{e}_1) \cup \overline{\omega}(\widehat{e}_2)$ . Hence,  $\overline{\omega}(e) = \overline{\omega}(e_1) \subseteq \overline{\omega}(\widehat{e}_1) \subseteq \overline{\omega}(\widehat{e})$ .
  - (c)  $\mathcal{D} \setminus \{0\} \cap \overline{\omega}(b) \neq \emptyset$ . Therefore  $\mathcal{D} \setminus \{0\} \cap \overline{\omega}(\widehat{b}) \neq \emptyset$ . Hence, by the definitions of  $\overline{\omega}$  and  $\widehat{\omega}$ , the induction hypothesis, and monotonicity of set union, we have  $\overline{\omega}(e) = \overline{\omega}(e_1) \cup \overline{\omega}(e_2) \subseteq \overline{\omega}(\widehat{e}_1) \cup \overline{\omega}(\widehat{e}_2) = \overline{\omega}(\widehat{e})$ .

Given a set of states  $S$  of a program  $P$ , let  $\text{post}_P(S) = \{s' : s \xrightarrow{\sigma} s' \text{ and } s \in S\}$ , the set of states reachable from  $S$  by a single transition. We can now state the soundness of the abstraction:

**Theorem 1 (Soundness).** *Let  $\widehat{P}$  be an abstraction of  $P$  w.r.t. the family of sets of indexes  $\{R(a)\}_{A_P}$  and  $\widehat{V} \subseteq V_P$ . Let  $h$ ,  $\alpha$  and  $\gamma$  be defined as above, then the abstract transition relation  $\xrightarrow{\sigma}_{\widehat{P}}$  is an upper approximation of the concrete transition relation  $\xrightarrow{\sigma}_P$ , in symbols  $\text{post}_P \subseteq (\gamma \circ \text{post}_{\widehat{P}} \circ \alpha)$ .*

*Proof.* Let  $s = \langle i, \omega \rangle \in S_P$  be any concrete state. The proof proceeds by cases, considering the possible statements  $s_i$ , labelling state  $s$ .<sup>7</sup>

- $s_i$  is `;` (or `return ;`) and so is  $\widehat{s}_i$ . Given the definition of transition relation, the thesis immediately follows from the properties of the Galois connection  $\langle \alpha, \gamma \rangle$ ;
- $s_i$  is  $x = e$ ; where  $x \in V_P$  is a variable.<sup>8</sup> Then, in the concrete program,  $\langle i, \omega \rangle \xrightarrow{e}_P \langle s\text{Succ}_P(i), \omega' \rangle$ , where  $\omega' \in \Omega' = \{\omega[d/x] : d \in \overline{\omega}(e)\}$ . Then we have two cases.  
If  $x \in \widehat{V}$ , then  $\widehat{s}_i$  is  $x = \widehat{e}$ ; . By Lemma 2,  $\overline{\omega}(e) \subseteq \overline{\widehat{\omega}}(\widehat{e})$ . Moreover, by the definition of  $\xrightarrow{\sigma}_{\widehat{P}}$ ,  $\langle i, \widehat{\omega} \rangle \xrightarrow{e}_{\widehat{P}} \langle s\text{Succ}_P(i), \omega_1 \rangle$ , where  $\omega_1 \in \Omega_1 = \{\omega = \widehat{\omega}[d/x] : d \in \overline{\widehat{\omega}}(\widehat{e})\}$ .  
For every  $\omega' \in \Omega'$ , we have that  $h(\langle s\text{Succ}_P(i), \omega' \rangle) \in \{\langle s\text{Succ}_P(i), \omega \rangle : \omega \in \Omega_1\}$  (by the definition of the functions  $\widehat{\cdot}$  and  $h(\cdot)$ ). Therefore, since  $\langle \alpha, \gamma \rangle$  forms a Galois connection, we conclude that:

$$\{\langle s\text{Succ}_P(i), \omega' \rangle : \omega' \in \Omega'\} \subseteq \gamma(\{\langle s\text{Succ}_P(i), \omega_1 \rangle : \omega_1 \in \Omega_1\})$$

In the second case,  $x \notin \widehat{V}$ . Then,  $\widehat{s}_i$  is a skip statement `;`. The by the definition of  $\xrightarrow{\sigma}_{\widehat{P}}$ , we have  $\langle i, \widehat{\omega} \rangle \xrightarrow{e}_{\widehat{P}} \langle s\text{Succ}_P(i), \widehat{\omega} \rangle$ . Since  $\omega$  and  $\omega'$  only differ on the value of a variable not belonging to  $V_{\widehat{P}}$ , by the definition of the function  $\widehat{\cdot}$ ,  $\widehat{\omega} = \widehat{\omega}'$ . Therefore, by the definition of  $\gamma$ , we conclude that  $\langle s\text{Succ}_P(i), \omega' \rangle \in \gamma(\{\langle s\text{Succ}_P(i), \widehat{\omega}' \rangle\})$ .

- $s_i$  is  $a[j] = e$ ; (with  $e$  a liner expression with arrays), and assume  $R(a) = \{k_1, \dots, k_n\}$ . Then,  $\widehat{s}_i$  is of the form

$$a_{k_1}, \dots, a_{k_n} = (\widehat{j} == k_1) ? \widehat{e} : a_{k_1}, \dots, (\widehat{j} == k_n) ? \widehat{e} : a_{k_n}.$$

In the concrete program,  $\langle i, \omega \rangle \xrightarrow{e}_P \langle s\text{Succ}_P(i), \omega' \rangle$ , where

$$\omega' \in \Omega' = \{\omega'' = \omega[(\omega(a)[k/d])/a] : k \in \overline{\omega}(j) \text{ and } d \in \overline{\omega}(e)\}$$

In the abstract program,  $\langle i, \widehat{\omega} \rangle \xrightarrow{e}_{\widehat{P}} \langle s\text{Succ}_P(i), \omega_1 \rangle$ , where  $\omega_1 \in \Omega_1$  and

$$\Omega_1 = \{\omega'_1 = \widehat{\omega}[d_1/a_{k_1}, \dots, d_n/a_{k_n}] : d_i \in \overline{\widehat{\omega}}((\widehat{j} == k_i) ? \widehat{e} : a_{k_i}) \text{ for } 1 \leq i \leq n\}$$

Moreover, by Lemma 2,  $\overline{\omega}(j) \subseteq \overline{\widehat{\omega}}(\widehat{j})$ ,  $\overline{\omega}(e) \subseteq \overline{\widehat{\omega}}(\widehat{e})$ , and, for any  $k \in \{k_1, \dots, k_n\}$ ,  $\overline{\omega}(a[k]) = \overline{\widehat{\omega}}(a_k)$ .

We need to show that for any  $\omega'' \in \Omega'$ ,  $\langle s\text{Succ}_P(i), \omega'' \rangle \in \gamma(\{\langle s\text{Succ}_P(i), \omega \rangle : \omega \in \Omega_1\})$ .

Let us consider an arbitrary  $\omega'' \in \Omega'$ . Then,  $\omega'' = \omega[(\omega(a)[d/k])/a]$  for some  $k \in \overline{\omega}(j)$  and  $d \in \overline{\omega}(e)$ .

We may have two cases:

<sup>7</sup> Throughout this proof we will extensively use some fundamental properties of Galois connections. Among them we recall  $\forall x \in S_P, x \subseteq \gamma(\alpha(x))$  and  $\forall y \in S_{\widehat{P}}, \alpha(\gamma(y)) \subseteq y$ . Moreover, given that the control flow graphs of  $P$  and  $\widehat{P}$  are identical, we have that  $s\text{Succ}_P(i) = s\text{Succ}_{\widehat{P}}(i)$  and we will use  $s\text{Succ}_P(i)$  for the successor of the abstract program as well. Same goes for the set  $\text{Succ}_P(i)$ .

<sup>8</sup> Here we only prove the single, non-parallel, assignment. The proof can be easily extended to the parallel case.

1.  $k \in \{k_1, \dots, k_n\}$ . Since  $\bar{\omega}(j) \subseteq \widehat{\omega}(j)$ , we also have that  $k \in \widehat{\omega}(j)$ . Therefore,  $\bar{\omega}(j == k) \cap \mathcal{D} \setminus \{0\} \neq \emptyset$ . Moreover,  $0 \in \bar{\omega}(j == k')$ , for  $k \neq k' \in \{k_1, \dots, k_n\}$ . As a consequence,  $\Omega_2 = \{\widehat{\omega}[d/a_k] : d \in \bar{\omega}(\widehat{e})\} \subseteq \Omega_1$ . Clearly, since  $\bar{\omega}(e) \subseteq \widehat{\omega}(\widehat{e})$ , we also have  $d \in \bar{\omega}(\widehat{e})$ . Thus,  $\langle sSucc_P(i), \omega' \rangle \in \gamma(\{\langle sSucc_P(i), \omega \rangle : \omega \in \Omega_2\})$ , by definition of  $\gamma$ . Hence the conclusion.
  2.  $k \notin \{k_1, \dots, k_n\}$ . Therefore,  $\omega''(a)(k') = \omega(a)(k')$ , for any  $k' \in \{k_1, \dots, k_n\}$ . Since,  $\bar{\omega}(j) \subseteq \widehat{\omega}(j)$ , we also have that  $k \in \widehat{\omega}(j)$ . Hence,  $0 \in \widehat{\omega}(j == k')$ , for  $k' \in \{k_1, \dots, k_n\}$ . As a consequence,  $\widehat{\omega} \in \Omega_1$ . Since  $\omega$  and  $\omega''$  only differ on the value of some array element not belonging to  $\{k_1, \dots, k_n\}$ , by the definition of the function  $\widehat{\cdot}$ ,  $\widehat{\omega} = \widehat{\omega}'$ . Therefore, also  $\widehat{\omega}' \in \Omega_1$ . By the definition of  $\gamma$ , we conclude that  $\langle sSucc_P(i), \omega' \rangle \in \gamma(\{\langle sSucc_P(i), \omega \rangle : \omega \in \Omega_1\})$ .
- $s_i$  is **if**( $e$ ) (**while**( $e$ ) or **assume**( $e$ );), where  $e$  is an (boolean) linear expression with arrays. Then  $\widehat{s}_i$  is **if**( $\widehat{e}$ ) (**while**( $\widehat{e}$ ) or **assume**( $\widehat{e}$ );), and, by Lemma 2,  $\bar{\omega}(e) \subseteq \widehat{\omega}(\widehat{e})$ . According to the definition of the transition relation (see Section 3), there are three cases:
    1. if  $0 \in \bar{\omega}(b)$  and  $\bar{\omega}(b) \cap \mathcal{D} \setminus \{0\} \neq \emptyset$ , then  $\langle i, \omega \rangle \xrightarrow{e}_P \langle i', \omega \rangle$ , where  $i' \in Succ_P(i)$ . Since  $\bar{\omega}(e) \subseteq \widehat{\omega}(\widehat{e})$ , then  $0 \in \widehat{\omega}(b)$  and  $\widehat{\omega}(b) \cap \mathcal{D} \setminus \{0\} \neq \emptyset$ . Therefore,  $\langle i, \widehat{\omega} \rangle \xrightarrow{e}_P \langle i', \widehat{\omega} \rangle$ , where  $i' \in Succ_P(i)$ , and the thesis follows from the properties of Galois connection;
    2.  $\bar{\omega}(e) = \{0\}$  and  $\langle i, \omega \rangle \xrightarrow{e}_P \langle Tsucc_P(i), \omega \rangle$ . Since  $\bar{\omega}(e) \subseteq \widehat{\omega}(\widehat{e})$ , then either  $\bar{\omega}(\widehat{e}) = \{0\}$ , or  $0 \in \bar{\omega}(\widehat{b})$  and  $\widehat{\omega}(\widehat{b}) \cap \mathcal{D} \setminus \{0\} \neq \emptyset$ . Therefore,  $\langle i, \widehat{\omega} \rangle \xrightarrow{e}_P \langle i', \widehat{\omega} \rangle$ , where, depending on the case, either  $i' = Tsucc_P(i)$  or  $i' \in Succ_P(i)$ . The thesis immediately follows;
    3.  $\bar{\omega}(b) \subseteq \mathcal{D} \setminus \{0\}$  and  $\langle i, \omega \rangle \xrightarrow{e}_P \langle Fsucc_P(i), \omega \rangle$ . The proof is similar to the proof of the previous case.
  - $s_i$  is **assert**( $e$ ); where  $e$  is an (boolean) linear expression with arrays. Then  $\widehat{s}_i$  is **assert**( $\widehat{e}$ );. The proof is similar to the previous case.
  - $i = \text{Exit}_P(pr)$ . Let  $\langle i, \omega \rangle \xrightarrow{\text{RET}(i_1, \omega_1)}_P \langle i_1, \omega_2 \rangle$  be any state transition of the concrete program. By definition of transition,  $i_1 \in Succ_P(i)$ ,  $\omega_2(g) = \omega(g)$  for all  $g \in \text{Globals}_P$  and  $\omega_2(x) = \omega_1(x)$  for all  $x \in \text{Locals}_P(i_1)$ . Since the control-flow graphs of the the two programs are isomorphic,  $Succ_P(i) = Succ_{\widehat{P}}(i)$ , we have  $i_1 \in Succ_{\widehat{P}}(i)$ . Moreover, it is immediate to check that  $\widehat{\omega}_2(g) = \widehat{\omega}(g)$  for all  $g \in \text{Globals}_{\widehat{P}}$  and  $\widehat{\omega}_2(x) = \widehat{\omega}_1(x)$  for all  $x \in \text{Locals}_{\widehat{P}}(i_1)$ . Therefore,  $\langle i, \widehat{\omega} \rangle \xrightarrow{\text{RET}(i_1, \widehat{\omega}_1)}_{\widehat{P}} \langle i_1, \widehat{\omega}_2 \rangle$  is a state transition of  $\widehat{P}$ . By the properties of Galois connections  $\{\langle i_1, \omega_2 \rangle\} \subseteq \gamma(\alpha(\{\langle i_1, \omega_2 \rangle\}))$  and, in addition,  $\alpha(\{\langle i_1, \omega_2 \rangle\}) = \{\widehat{\omega}_2\}$ . Hence, the thesis follows.
  - $s_i$  is **pr**( $\mathbf{e}$ ); where  $\mathbf{e}$  is a vector of linear expressions with arrays. Then, statement  $\widehat{s}_i$  is **pr**( $\widehat{\mathbf{e}}$ );. Let  $\langle i, \omega \rangle \xrightarrow{\text{CALL}(\text{RetPt}(i), \omega_1)}_P \langle \text{First}_P(pr), \omega_2 \rangle$  be any state transition from  $\langle i, \omega \rangle$ . Then,  $\omega_1(x) = \omega(x)$  for all  $x \in \text{Locals}_P(i)$ ,  $\omega_2(x_i) = d$  for  $d_i \in \bar{\omega}(e_i)$  for all  $x_i \in \text{Formals}_P(i_2)$ , and  $\omega_2(g) = \omega(g)$  for all  $g \in \text{Globals}_P$ . Both  $\text{First}_P(pr)$  and  $\text{RetPt}_P(i)$  also belong to the control-flow graph of  $\widehat{P}$ . Moreover, as in the previous case, it is immediate to check that  $\widehat{\omega}_2(g) = \widehat{\omega}(g)$  for all  $g \in \text{Globals}_{\widehat{P}}$  and  $\widehat{\omega}_1(x) = \widehat{\omega}(x)$  for all  $x \in \text{Locals}_{\widehat{P}}(i_1)$ . In addition, since  $\bar{\omega}(e_i) \subseteq \widehat{\omega}(\widehat{e}_i)$ , for all  $e_i$  occurring in  $\mathbf{e}$ , then we also have that  $d_i \in \bar{\omega}(\widehat{e}_i)$ . Therefore,  $\langle i, \widehat{\omega} \rangle \xrightarrow{\text{CALL}(\text{RetPt}(i), \widehat{\omega}_1)}_{\widehat{P}} \langle \text{First}_{\widehat{P}}(pr), \widehat{\omega}_2 \rangle$  is a state transition of  $\widehat{P}$ . Similarly to the previous case, the thesis follows.

An immediate consequence of Theorem 1 is the following, stating the soundness of the abstraction defined in this section when applied to a linear, array-free program  $P$ :

**Corollary 1.** *Let  $P$  be a linear program without arrays (i.e. an array-free program) and  $\widehat{V} \subseteq V_P$ . Moreover, let  $\widehat{P}$  be the abstraction of  $P$  w.r.t.  $\widehat{V}$ , and  $h$ ,  $\alpha$  and  $\gamma$  be defined as above. Then,  $\text{post}_P \subseteq (\gamma \circ \text{post}_{\widehat{P}} \circ \alpha)$ .*

In our abstraction/refinement schema, the initial set of variables chosen for the abstraction is  $\widehat{V} = V_P$ , that is, the initial abstracted program contains all the numeric variables of  $P$  and does not contain any array expression.

## 5 Interprocedural Data-Flow Analysis

We now define a procedure for interprocedural data-flow analysis of linear programs. Our procedure extends the one described in [5], which is itself built on top of the tabulation algorithm defined by Reps, Horwitz, and Sagiv in [11]. For the sake of brevity here we give a simplified version of the procedure, in the sense that we do not describe *Summary Edges*. Summary Edges are data structures that cache the result of the analysis of procedures thereby avoiding redundant work in subsequent computations. A detailed description of Summary Edges may be found in [7] when they are applied to Linear Programs, and in [5] when applied to Boolean Programs.

Let  $i \in N_P$  and  $e = \text{First}_P(\text{ProcOf}_P(i))$ . A *path edge*  $\pi_i = \langle \omega_e, \omega_i \rangle$  *incident in*  $i$  is a pair of valuations such that there is a valid path  $\langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^e} \langle e, \omega_e \rangle$  and a same-level valid path  $\langle e, \omega_e \rangle \xrightarrow{\Sigma_e^i} \langle i, \omega_i \rangle$  for some valuation  $\omega_0$ . In other words, a path edge represents a suffix of a valid path from  $\langle 1, \omega_0 \rangle$  to  $\langle i, \omega_i \rangle$ . Let  $i \in N_P$ , by  $\Pi_i(P)$  we denote the set of path edges incident in  $i$  associated with program  $P$ . Vertex  $i$  is reachable if and only if  $\Pi_i(P) \neq \emptyset$ .

Let  $\mathcal{Q}(P)$  be the  $N_P$ -indexed family of sets  $\mathcal{Q}_i(P) = \{\langle \omega_1, \omega_2 \rangle : \omega_1, \omega_2 : \text{InScope}_P(i) \rightarrow \mathcal{D}\}$  for all  $i \in N_P$ . Let  $\Pi, \Pi^1$  be two  $N_P$ -indexed family of sets such that  $\Pi_i \subseteq \mathcal{Q}_i(P)$  and  $\Pi_i^1 \subseteq \mathcal{Q}_i(P)$ , for all  $i \in N_P$ . We define the binary relation  $\Pi \Rightarrow \Pi^1$  as follows. Let  $i \in N_P$ ,  $\Pi_j^1 = \Pi_j$  for all  $j \notin \text{Succ}_P(i)$  and

- if  $i$  corresponds to a skip (;) or **return**, then  $\Pi_{\text{sSucc}_P(i)}^1 = \Pi_{\text{sSucc}_P(i)} \cup \Pi_i$ ;
- if  $i$  corresponds to an assignment  $\mathbf{y}=\mathbf{e}$ , then

$$\Pi_{\text{sSucc}_P(i)}^1 = \Pi_{\text{sSucc}_P(i)} \cup \{\langle \omega_e, \omega_i[\mathbf{d}/\mathbf{y}] \rangle : \langle \omega_e, \omega_i \rangle \in \Pi_i, \mathbf{d} \in \bar{\omega}_i(\mathbf{e})\}$$

- if  $i$  corresponds to an **if**( $e$ ), a **while**( $e$ ), or an **assert**( $e$ ) statement, then

$$\begin{aligned} \Pi_{\text{Tsucc}_P(i)}^1 &= \Pi_{\text{Tsucc}_P(i)} \cup \{\langle \omega_e, \omega_i \rangle \in \Pi_i : d \in \bar{\omega}_i(e) \text{ for some } d \neq 0\} \\ \Pi_{\text{Fsucc}_P(i)}^1 &= \Pi_{\text{Fsucc}_P(i)} \cup \{\langle \omega_e, \omega_i \rangle \in \Pi_i : 0 \in \bar{\omega}_i(e)\} \end{aligned}$$

- if  $i$  corresponds to a procedure call  $pr(\mathbf{a})$ , then

$$\Pi_{\text{sSucc}_P(i)}^1 = \Pi_{\text{sSucc}_P(i)} \cup \{\langle \omega_j, \omega_j \rangle : \omega_j(\mathbf{g}) = \omega_i(\mathbf{g}), \omega_j(\mathbf{y}) \in \bar{\omega}_i(\mathbf{a}), \langle \omega_e, \omega_i \rangle \in \Pi_i, \mathbf{g} = \text{Globals}_P, \mathbf{y} = \text{Formals}_P(pr)\}$$

- if  $i = \text{Exit}_P(pr)$  then

$$\begin{aligned} \Pi_j^1 &= \Pi_j \cup \{\langle \omega_e, \omega_j \rangle : \omega_j(\mathbf{z}) = \omega_k(\mathbf{z}), \omega_j(\mathbf{g}) = \omega_i(\mathbf{g}), \\ &\quad \langle \omega_e, \omega_k \rangle \in \Pi_k, \langle \omega_h, \omega_i \rangle \in \Pi_i, \\ &\quad \omega_h(\mathbf{y}) \in \bar{\omega}_k(\mathbf{a}), \omega_k(\mathbf{g}) = \omega_h(\mathbf{g}), \\ &\quad \text{RetPt}(k) = j, \mathbf{y} = \text{Formals}_P(pr), \\ &\quad \mathbf{z} = \text{Locals}_P(k), \text{ and } \mathbf{g} = \text{Globals}_P\} \end{aligned}$$

if  $j \in \text{Succ}_P(i)$ . Notice that if  $j \in \text{Succ}_P(i)$  then  $j$  is the first statement following a procedure call  $s_k = pr(\mathbf{a})$  (condition  $\text{RetPt}(k) = j$ ) and  $\Pi_j^1$  is obtained from  $\Pi_j$  by adding all the pairs  $\langle \omega_e, \omega_j \rangle$

obtained from  $\langle \omega_e, \omega_k \rangle \in \Pi_k$  and  $\langle \omega_h, \omega_i \rangle \in \Pi_i$  such that the values associated at  $k$  (according to  $\omega_k$ ) to the globals and to actuals  $\mathbf{a}$  are equal to the values associated at  $h$  (according to  $\omega_h$ ) to the globals and to the formals of  $pr$  respectively (conditions  $\omega_h(\mathbf{y}) \in \bar{\omega}_k(\mathbf{a})$  and  $\omega_k(\mathbf{g}) = \omega_h(\mathbf{g})$ ) and  $\omega_j$  is defined to be equal to  $\omega_k$  over the locals of the caller (condition  $\omega_j(\mathbf{z}) = \omega_k(\mathbf{z})$ ) and to be equal to  $\omega_i$  over the globals (condition  $\omega_j(\mathbf{g}) = \omega_i(\mathbf{g})$ ), thereby inheriting the effects on the globals of the procedure call. Notice that  $e$  and  $h$  are the vertexes associated to the first statements in the procedure containing  $s_k$  (the procedure call) and vertex  $i$  (the exit vertex of the called procedure) respectively.

**Theorem 2 (Soundness).** *If  $\Pi \Rightarrow \Pi^1$ , then for all  $i \in N_P$ ,  $\Pi_i \subseteq \mathbf{\Pi}_i(P)$  implies  $\Pi_i^1 \subseteq \mathbf{\Pi}_i(P)$ .*

*Proof.* Let  $\Pi \subseteq \mathbf{\Pi}(P)$  and  $\Pi \Rightarrow \Pi^1$ . Since  $\Pi \Rightarrow \Pi^1$ , then  $\Pi^1$  is obtained from  $\Pi$  by one of the cases in the definition of  $\Rightarrow$  and let  $i \in N_P$  be vertex of the statement considered. If  $j \notin \text{Succ}_P(i)$  then  $\Pi_j^1 = \Pi_j$  and since  $\Pi \subseteq \mathbf{\Pi}(P)$ , it trivially follows that  $\Pi_j \subseteq \mathbf{\Pi}_j(P)$ . If  $j \in \text{Succ}_P(i)$  then  $\Pi_j^1 = \Pi_j \cup \Pi^*$  for some  $\Pi^*$  and we must prove that  $\Pi_j^1 \subseteq \mathbf{\Pi}_j(P)$ . This amounts to proving that both  $\Pi_j \subseteq \mathbf{\Pi}_j(P)$  and  $\Pi^* \subseteq \mathbf{\Pi}_j(P)$  hold. The former is an obvious consequence of the hypothesis  $\Pi \subseteq \mathbf{\Pi}(P)$ . For the latter we must show that the pairs of valuations in  $\Pi^*$  are path edges incident in  $j$  and we proceed by cases:

- If  $s_i$  is `;` (or `return ;`), then  $j = \text{sSucc}_P(i)$  and  $\Pi^* = \Pi_i$ . Let  $\langle \omega_e, \omega_i \rangle \in \Pi_i$ . By hypothesis  $\Pi_i \subseteq \mathbf{\Pi}_i(P)$ . This means that  $\langle \omega_e, \omega_i \rangle$  is a path edge incident in  $i$  and therefore that there exists a valid path  $\langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^e} \langle e, \omega_e \rangle \xrightarrow{\Sigma_e^i} \langle i, \omega_i \rangle$ . This path can be extended to the path

$$\langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^e} \langle e, \omega_e \rangle \xrightarrow{\Sigma_e^i} \langle i, \omega_i \rangle \xrightarrow{\epsilon} \langle j, \omega_j \rangle,$$

which is a valid path. Therefore  $\langle \omega_e, \omega_i \rangle$  is a path edge incident in  $j$ .

- If  $s_i$  is an assignment  $\mathbf{y} = \mathbf{e}$ , then  $j = \text{sSucc}_P(i)$  and  $\Pi^* = \{ \langle \omega_e, \omega_i[\mathbf{d}/\mathbf{y}] \rangle : \langle \omega_e, \omega_i \rangle \in \Pi_i, \mathbf{d} \in \bar{\omega}_i(\mathbf{e}) \}$ . Let  $\langle \omega_1, \omega_2 \rangle \in \Pi^*$ . By the definition of  $\Pi^*$  we know that  $\omega_1 = \omega_e$  and  $\omega_2 = \omega_i[\mathbf{d}/\mathbf{y}]$  for some  $\langle \omega_e, \omega_i \rangle \in \Pi_i$  and  $\mathbf{d} \in \bar{\omega}_i(\mathbf{e})$ . By hypothesis  $\Pi_i \subseteq \mathbf{\Pi}_i(P)$ . This means that  $\langle \omega_e, \omega_i \rangle$  is a path edge incident in  $i$  and therefore that there exists a valid path  $\langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^e} \langle e, \omega_e \rangle \xrightarrow{\Sigma_e^i} \langle i, \omega_i \rangle$ . This path can be extended to  $\langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^e} \langle e, \omega_e \rangle \xrightarrow{\Sigma_e^i} \langle i, \omega_i \rangle \xrightarrow{\epsilon} \langle j, \omega_i[\mathbf{d}/\mathbf{y}] \rangle$  which is a valid path. Therefore  $\langle \omega_1, \omega_2 \rangle$  is a path edge incident in  $j$ .
- If  $s_i$  is `if(e)`, `while(e)`, or `assert(e)` then  $j \in \{ \text{Tsucc}_P(i), \text{Fsucc}_P(i) \}$ . If  $j = \text{Tsucc}_P(i)$  then  $\Pi^* = \{ \langle \omega_e, \omega_i \rangle \in \Pi_i : d \in \bar{\omega}_i(e) \text{ for some } d \neq 0 \}$ . Let  $\langle \omega_1, \omega_2 \rangle \in \Pi^*$ . By the definition of  $\Pi^*$  we know that  $\omega_1 = \omega_e$  and  $\omega_2 = \omega_i$  for some  $\langle \omega_e, \omega_i \rangle \in \Pi_i$  such that  $d \in \bar{\omega}_i(e)$  for some  $d \neq 0$ . By hypothesis  $\Pi_i \subseteq \mathbf{\Pi}_i(P)$ . This means that  $\langle \omega_e, \omega_i \rangle$  is a path edge incident in  $i$  and therefore that there exists a valid path  $\langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^e} \langle e, \omega_e \rangle \xrightarrow{\Sigma_e^i} \langle i, \omega_i \rangle$ . This path can be extended to  $\langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^e} \langle e, \omega_e \rangle \xrightarrow{\Sigma_e^i} \langle i, \omega_i \rangle \xrightarrow{\epsilon} \langle j, \omega_i \rangle$  which is a valid path. Therefore  $\langle \omega_1, \omega_2 \rangle$  is a path edge incident in  $j$ . The proof for  $j = \text{Fsucc}_P(i)$  is analogous and therefore omitted.
- If  $s_i$  is a procedure call  $pr(\mathbf{e})$  then  $j = \text{sSucc}_P(i)$  and  $\Pi^* = \{ \langle \omega_j, \omega_j \rangle : \omega_j(\mathbf{g}) = \omega_i(\mathbf{g}), \omega_j(\mathbf{y}) \in \bar{\omega}_i(\mathbf{e}), \langle \omega_e, \omega_i \rangle \in \Pi_i, \mathbf{g} = \text{Globals}_P, \mathbf{y} = \text{Formals}_P(pr) \}$ . Let  $\langle \omega_1, \omega_2 \rangle \in \Pi^*$ . By the definition of  $\Pi^*$  we know that  $\omega_1 = \omega_j$  and  $\omega_2 = \omega_j$  where  $\omega_j$  is a valuation such that  $\omega_j(\mathbf{g}) = \omega_i(\mathbf{g})$  and  $\omega_j(\mathbf{y}) \in \bar{\omega}_i(\mathbf{e})$  for some  $\langle \omega_e, \omega_i \rangle \in \Pi_i$  with  $\mathbf{g} = \text{Globals}_P$  and  $\mathbf{y} = \text{Formals}_P(pr)$ . By hypothesis  $\Pi_i \subseteq \mathbf{\Pi}_i(P)$ . This means that  $\langle \omega_e, \omega_i \rangle$  is a path edge incident in  $i$  and therefore that there exists a valid path  $\langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^e} \langle e, \omega_e \rangle \xrightarrow{\Sigma_e^i} \langle i, \omega_i \rangle$ . This path can be extended to  $\langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^e} \langle e, \omega_e \rangle$

- $\xrightarrow{\Sigma_e^i} \langle i, \omega_i \rangle \xrightarrow{\text{CALL}(\text{RetPt}_P(i), \omega)} \langle j, \omega_j \rangle$  where  $\omega : \text{Locals}_P(i) \rightarrow \mathcal{D}$  is such that  $\omega(\mathbf{x}) = \omega_i(\mathbf{x})$ , with  $\mathbf{x} \in \text{Locals}_P(i)$ , which is a valid path. Therefore  $\langle \omega_1, \omega_2 \rangle$  is a path edge incident in  $j$ .
- If  $i = \text{Exit}_P(pr)$ , then

$$\begin{aligned} \Pi^* = \{ \langle \omega_e, \omega_j \rangle : & \omega_j(\mathbf{z}) = \omega_k(\mathbf{z}), \omega_j(\mathbf{g}) = \omega_i(\mathbf{g}), \\ & \langle \omega_e, \omega_k \rangle \in \Pi_k, \langle \omega_h, \omega_i \rangle \in \Pi_i, \\ & \omega_k(\mathbf{e}) = \omega_h(\mathbf{y}), \omega_k(\mathbf{g}) = \omega_h(\mathbf{g}), \\ & \text{RetPt}(k) = j, \mathbf{y} = \text{Formals}_P(pr), \\ & \mathbf{z} = \text{Locals}_P(k), \text{ and } \mathbf{g} = \text{Globals}_P \} \end{aligned}$$

for  $j \in \text{Succ}_P(i)$ . Let  $\langle \omega_1, \omega_2 \rangle \in \Pi^*$ . By the definition of  $\Pi^*$ ,  $\omega_1 = \omega_e$  and  $\omega_2 = \omega_j$  where  $\omega_j(\mathbf{z}) = \omega_k(\mathbf{z})$  and  $\omega_j(\mathbf{g}) = \omega_i(\mathbf{g})$  for  $\langle \omega_e, \omega_k \rangle \in \Pi_k$  and  $\langle \omega_h, \omega_i \rangle \in \Pi_i$  such that  $\omega_k(\mathbf{e}) = \omega_h(\mathbf{y})$  and  $\omega_k(\mathbf{g}) = \omega_h(\mathbf{g})$  with  $\text{RetPt}(k) = j$ ,  $\mathbf{y} = \text{Formals}_P(pr)$ ,  $\mathbf{z} = \text{Locals}_P(k)$ , and  $\mathbf{g} = \text{Globals}_P$ . By hypothesis  $\Pi_i \subseteq \Pi_i(P)$  and  $\Pi_k \subseteq \Pi_k(P)$ . This means that  $\langle \omega_e, \omega_k \rangle$  and  $\langle \omega_h, \omega_i \rangle$  are path edges incident in  $k$  and  $i$  respectively and therefore that there exist valid paths  $\langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^e} \langle e, \omega_e \rangle \xrightarrow{\Sigma_e^k} \langle k, \omega_k \rangle$  and  $\langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^h} \langle h, \omega_h \rangle \xrightarrow{\Sigma_h^i} \langle i, \omega_i \rangle$ . Consider the path

$$\langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^e} \langle e, \omega_e \rangle \xrightarrow{\Sigma_e^k} \langle k, \omega_k \rangle \xrightarrow{\text{CALL}(\text{RetPt}(k), \omega)} \langle h, \omega_h \rangle \xrightarrow{\Sigma_h^i} \langle i, \omega_i \rangle \xrightarrow{\text{RET}(j, \omega)} \langle j, \omega_j \rangle,$$

where  $\omega : \text{Locals}_P(i) \rightarrow \mathcal{D}$  is such that  $\omega(\mathbf{x}) = \omega_k(\mathbf{x})$ , with  $\mathbf{x} \in \text{Locals}_P(i)$ . This is a valid path. In fact  $\langle k, \omega_k \rangle \xrightarrow{\text{CALL}(\text{RetPt}(k), \omega)} \langle h, \omega_h \rangle$  and  $\langle i, \omega_i \rangle \xrightarrow{\text{RET}(j, \omega)} \langle j, \omega_j \rangle$  are legal transitions, because of  $\omega_k(\mathbf{e}) = \omega_h(\mathbf{y})$ ,  $\omega_k(\mathbf{g}) = \omega_h(\mathbf{g})$  and  $\omega_j(\mathbf{z}) = \omega_k(\mathbf{z})$ ,  $\omega_j(\mathbf{g}) = \omega_i(\mathbf{g})$  respectively.

Let  $\Pi^0$  be such that  $\Pi_1^0 = \{ \langle \omega, \omega \rangle : \omega(x) \in \mathcal{D} \text{ for all } x \in \text{InScope}_P(1) \}$  and  $\Pi_j^0 = \emptyset$  for all  $j \in N_P \setminus \{1\}$ .

**Theorem 3 (Completeness).** *Let  $\Pi^0$  be defined as above. If  $\langle \omega_h, \omega_j \rangle \in \Pi_j(P)$  then there exists  $\Pi^1$  such that  $\Pi^0 \Rightarrow^* \Pi^1$  and  $\langle \omega_h, \omega_j \rangle \in \Pi_j^1$ .*

*Proof.* Let  $\langle \omega_h, \omega_j \rangle \in \Pi_j(P)$ , then there exists a valid path  $\pi = \langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^h} \langle h, \omega_h \rangle \xrightarrow{\Sigma_h^j} \langle j, \omega_j \rangle$  for some valuation  $\omega_0$ . The proof is by induction on the length of  $\pi$ . In the base case, the length of  $\pi$  is 0, i.e.  $\pi = \langle 1, \omega_0 \rangle$ . We take  $\Pi^1 = \Pi^0$ . Both  $\Pi^0 \Rightarrow^* \Pi^1$  and  $\langle \omega_h, \omega_j \rangle = \langle \omega_0, \omega_0 \rangle \in \Pi_1^0$  trivially hold. In the step case, let  $\pi$  be of length  $n + 1$ . Let  $\pi' = \langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^e} \langle e, \omega_e \rangle \xrightarrow{\Sigma_e^i} \langle i, \omega_i \rangle$  be the prefix of  $\pi$  of length  $n$ . Obviously  $\pi'$  is a valid path and therefore  $\langle \omega_e, \omega_i \rangle \in \Pi_i$ . By induction hypothesis there exists  $\Pi'$  such that  $\Pi^0 \Rightarrow^* \Pi'$  and  $\langle \omega_e, \omega_i \rangle \in \Pi'_i$ . Path  $\pi$  is obtained from  $\pi'$  by adding a transition associated with  $s_j$ . The proof continues by a case analysis.

- If  $s_i$  is a ; (or a **return** ;), then  $\pi = \langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^h} \langle h, \omega_h \rangle \xrightarrow{\Sigma_h^i} \langle i, \omega_i \rangle \xrightarrow{\epsilon} \langle j, \omega_j \rangle$  with  $\omega_j = \omega_i$  and  $\langle h, \omega_h \rangle = \langle e, \omega_e \rangle$ . By the definition of  $\Rightarrow$  it follows that there exists  $\Pi^1$  such that  $\Pi' \Rightarrow \Pi^1$  (and therefore  $\Pi^0 \Rightarrow^* \Pi^1$ ) with  $\Pi_j^1 = \Pi'_j \cup \Pi'_i$ . From this and the fact  $\langle \omega_e, \omega_i \rangle \in \Pi'_i$  it readily follows that  $\langle \omega_h, \omega_j \rangle \in \Pi_j^1$ .
- The cases where  $s_i$  is an assignment  $\mathbf{y} = \mathbf{e}$ , then  $\pi = \langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^h} \langle h, \omega_h \rangle \xrightarrow{\Sigma_h^i} \langle i, \omega_i \rangle \xrightarrow{\epsilon} \langle j, \omega_j \rangle$  with  $\omega_j = \omega_i[\mathbf{d}/\mathbf{y}]$ , with  $\mathbf{d} \in \bar{\omega}_i(\mathbf{e})$ , and  $\langle h, \omega_h \rangle = \langle e, \omega_e \rangle$ . By the definition of  $\Rightarrow$  it follows that there

exists  $\Pi^1$  such that  $\Pi' \Rightarrow \Pi^1$  (and therefore  $\Pi^0 \Rightarrow^* \Pi^1$ ) with

$$\Pi_j^1 = \Pi_j' \cup \{ \langle \omega_e, \omega_i[\mathbf{d}/\mathbf{y}] \rangle : \langle \omega_e, \omega_i \rangle \in \Pi_i', \mathbf{d} \in \bar{\omega}_i(\mathbf{e}) \}$$

From this and the fact  $\langle \omega_e, \omega_i \rangle \in \Pi_i'$  it readily follows that  $\langle \omega_h, \omega_j \rangle \in \Pi_j^1$ .

- If  $s_i$  is **if**( $e$ ), **while**( $e$ ), or **assert**( $e$ ), then  $\pi = \langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^h} \langle h, \omega_h \rangle \xrightarrow{\Sigma_h^i} \langle i, \omega_i \rangle \xrightarrow{\epsilon} \langle j, \omega_j \rangle$ , with  $\omega_j = \omega_i$ ,  $j \in \{ \text{Tsucc}_P(i), \text{Fsucc}_P(i) \}$ , and  $\langle h, \omega_h \rangle = \langle e, \omega_e \rangle$ . Let us consider the case where  $j = \text{Tsucc}_P(i)$  (the case where  $j = \text{Fsucc}_P(i)$  can be proved similarly). By the definition of  $\Rightarrow$  it follows that there exists  $\Pi^1$  such that  $\Pi' \Rightarrow \Pi^1$  (and therefore  $\Pi^0 \Rightarrow^* \Pi^1$ ) with

$$\Pi_j^1 = \Pi_j \cup \{ \langle \omega_e, \omega_i \rangle \in \Pi_i' : d \in \bar{\omega}_i(e) \text{ for some } d \neq 0 \}$$

Since  $j = \text{Tsucc}_P i$ , by the definition of state transition,  $d \in \bar{\omega}_i(e)$  for some  $d \neq 0$ . From this and the fact  $\langle \omega_e, \omega_i \rangle \in \Pi_i'$  it readily follows that  $\langle \omega_h, \omega_j \rangle \in \Pi_j^1$ .

- If  $s_i$  is a procedure call  $pr(\mathbf{e})$ , then

$$\pi = \langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^e} \langle e, \omega_e \rangle \xrightarrow{\Sigma_e^k} \langle k, \omega_k \rangle \xrightarrow{\text{CALL}(\text{RetPt}_P(k), \omega_k)} \langle j, \omega_j \rangle,$$

with  $j = \text{First}_P(pr)$ , and  $\langle e, \omega_e \rangle \xrightarrow{\Sigma_e^k} \langle k, \omega_k \rangle$  is a same-level path. Therefore, by induction hypothesis, there exists  $\Pi'$  such that  $\Pi^0 \Rightarrow^* \Pi'$  and  $\langle \omega_e, \omega_k \rangle \in \Pi_k'$ . By the definition of  $\Rightarrow$  it follows that there exists  $\Pi^1$  such that  $\Pi' \Rightarrow \Pi^1$  (and therefore  $\Pi^0 \Rightarrow^* \Pi^1$ ) with

$$\Pi_j^1 = \Pi_j' \cup \{ \langle \omega_j, \omega_j \rangle : \omega_j(\mathbf{g}) = \omega_{k'}(\mathbf{g}), \omega_j(\mathbf{y}) \in \bar{\omega}_{k'}(\mathbf{e}), \langle \omega_{e'}, \omega_{k'} \rangle \in \Pi_{k'}', \\ \mathbf{g} = \text{Globals}_P, \mathbf{y} = \text{Formals}_P(pr) \}$$

On the other hand, by the definition of state transition for a procedure call, we also have that  $\omega_j(\mathbf{g}) = \omega_k(\mathbf{g})$  and  $\omega_j(\mathbf{y}) \in \bar{\omega}_k(\mathbf{e})$ . Therefore,  $\langle \omega_k, \omega_j \rangle \in \Pi_j^1$  as required.

- If  $i = \text{Exit}_P(pr)$ , then

$$\pi = \langle 1, \omega_0 \rangle \xrightarrow{\Sigma_0^h} \langle h, \omega_h \rangle \xrightarrow{\Sigma_h^k} \langle k, \omega_k \rangle \xrightarrow{\text{CALL}(\text{RetPt}_P(k), \omega_k)} \langle e, \omega_e \rangle \xrightarrow{\Sigma_e^i} \langle i, \omega_i \rangle \xrightarrow{\text{RET}(j, \omega_k)} \langle j, \omega_j \rangle,$$

with  $e = \text{First}_P(pr)$  and  $j \in \text{Succ}_P(k)$ . Moreover, both  $\langle h, \omega_h \rangle \xrightarrow{\Sigma_h^k} \langle k, \omega_k \rangle$  and  $\langle e, \omega_e \rangle \xrightarrow{\Sigma_e^i} \langle i, \omega_i \rangle$  are same-level paths. Therefore, by induction hypothesis, there exists  $\Pi'$  such that  $\Pi^0 \Rightarrow^* \Pi'$  and both  $\langle \omega_h, \omega_k \rangle \in \Pi_k'$  and  $\langle \omega_e, \omega_i \rangle \in \Pi_i'$ . By the definition of  $\Rightarrow$  it follows that there exists  $\Pi^1$  such that  $\Pi' \Rightarrow \Pi^1$  (and therefore  $\Pi^0 \Rightarrow^* \Pi^1$ ) with

$$\Pi_j^1 = \Pi_j' \cup \{ \langle \omega_{h'}, \omega_j \rangle : \omega_j(\mathbf{z}) = \omega_{k'}(\mathbf{z}), \omega_j(\mathbf{g}) = \omega_i(\mathbf{g}), \\ \langle \omega_{h'}, \omega_{k'} \rangle \in \Pi_{k'}', \langle \omega_e, \omega_i \rangle \in \Pi_i', \\ \omega_e(\mathbf{y}) \in \bar{\omega}_{k'}(\mathbf{e}), \omega_{k'}(\mathbf{g}) = \omega_e(\mathbf{g}), \\ \text{RetPt}(k') = j, \mathbf{y} = \text{Formals}_P(pr), \\ \mathbf{z} = \text{Locals}_P(k'), \text{ and } \mathbf{g} = \text{Globals}_P \}$$

On the other hand, by the definition of state transition for a procedure call, we also have that  $\omega_e(\mathbf{g}) = \omega_k(\mathbf{g})$ , where  $\mathbf{g} = \text{Globals}_P$ , and  $\omega_e(\mathbf{y}) \in \bar{\omega}_k(\mathbf{e})$ , where  $\mathbf{y} = \text{Formals}_P(pr)$ . Similarly, by the definition of state transition for  $\text{Exit}_P(pr)$ ,  $\omega_j(\mathbf{g}) = \omega_i(\mathbf{g})$ , and  $\omega_j(\mathbf{z}) = \omega_k(\mathbf{z})$ , where  $\mathbf{z} = \text{Locals}_P(j)$ . Therefore,  $\langle \omega_h, \omega_i \rangle \in \Pi_j^1$  as required.

**Corollary 2.** *Let  $\Pi^0$  be defined as above and  $\Pi$  such that  $\Pi^0 \Rightarrow \Pi$ . For all  $i \in N_P$ ,  $i$  is reachable if and only if  $\Pi_i \neq \emptyset$ .*

## 6 Symbolic Model Checking of Linear Programs

Let  $e$  and  $ne$  be generalized linear expressions,  $B$  a set of atomic generalized boolean linear expressions. We define the relation  $e \rightarrow (B, ne)$  to be the smallest relation such that  $e \rightarrow (\emptyset, e)$  for all  $e \in V \cup \mathbb{Z} \cup \{\mathbf{u}\}$  and closed under the following inference rules:

$$\frac{e_1 \rightarrow (B_1, ne_1) \quad e_1 \rightarrow (B_2, ne_2)}{(e_1 \text{ op } e_2) \rightarrow (B_1 \cup B_2, (ne_1 \text{ op } ne_2))} \text{ if } \text{op} \in \{*, +, >=, =, <, >, ==, !=\}$$

$$\frac{b \rightarrow (B, nb) \quad e_1 \rightarrow (B_1, ne_1)}{(b ? e_1 : e_2) \rightarrow (B \cup B_1 \cup \{nb^+\}, ne_1)} \quad \frac{b \rightarrow (B, nb) \quad e_2 \rightarrow (B_2, ne_2)}{(b ? e_1 : e_2) \rightarrow (B \cup B_2 \cup \{nb^-\}, ne_2)}$$

where  $b^+$  ( $b^-$ ) is  $b != 0$  ( $b == 0$ , resp.) if  $b$  is a generalized linear expression and  $b$  ( $!b$ , resp.) if  $b$  is a generalized boolean expression. It can be shown that if  $e \rightarrow (ne, B)$ , then  $ne$  ( $B$ ) is a (set of, resp.) generalized linear expression(s) without conditionals.

Let  $U = \{u_1, u_2, \dots\}$  be an infinite set of variables such that  $U \cap V = \emptyset$ . If  $e$  is a generalized linear expression, by  $e^*$  we indicate the expression obtained from  $e$  by replacing every occurrence of  $!b$ ,  $e_1 != e_2$ , and  $e_1 == e_2$ , with  $\neg b$ ,  $\neg(e_1 = e_2)$ , and  $e_1 = e_2$ , respectively, and every occurrence of  $\mathbf{u}$  in  $e$  with a distinct variable  $u_m \in U$ . If  $w$  is a formula and  $u_{h_1}, \dots, u_{h_k}$  the variables from  $U$  occurring in  $w$ , by  $\exists U.w$  we denote the formula  $\exists u_{h_1} \dots \exists u_{h_k}.w$ . Notice that if  $e$  is a linear expression without conditionals, then  $e^*$  is a expression of the language of Linear Arithmetics.

Let  $\mathbf{y}$  and  $\mathbf{e}$  be  $k$ -uples of variables and generalized linear expressions with  $k > 0$  such that no variable in  $\mathbf{y}$  occurs in  $\mathbf{e}$ . We define

$$\alpha(\mathbf{y}, \mathbf{e}) = \bigvee \left\{ \bigwedge_{i=1}^k \left( \exists U. (y_i = ne_i \wedge \bigwedge B_i)^* \right) : e_i \rightarrow (B_i, ne_i) \text{ for } i = 1, \dots, k \right\}$$

Moreover, for a generalized linear expression  $e$  we define

$$\beta^+(e) = \bigvee \{ \exists U. (ne^+ \wedge \bigwedge B)^* : e \rightarrow (B, ne) \}$$

$$\beta^-(e) = \bigvee \{ \exists U. (ne_i^- \wedge \bigwedge B_i)^* : e \rightarrow (B, ne) \}$$

For example, if  $e = (3 * \mathbf{x} + (\mathbf{y} < 0 ? \mathbf{u} : \mathbf{y}))$  then  $e \rightarrow (\{\mathbf{y} < 0\}, 3 * \mathbf{x} + \mathbf{u})$  and  $e \rightarrow (\{\!|\mathbf{y} < 0\!\}, 3 * \mathbf{x} + \mathbf{y})$ . Therefore  $\alpha(z, e) = (\exists u_1. (y < 0 \wedge z = 3 * x + u_1) \vee (\neg(y < 0) \wedge z = 3 * x + y))$  which can be simplified to the logically equivalent formula  $(y < 0 \vee (\neg(y < 0) \wedge z = 3 * x + y))$ . Similarly,  $\beta^+(e) = (\exists u_1. (y < 0 \wedge \neg(3 * x + u_1 = 0)) \vee (\neg(y < 0) \wedge \neg(3 * x + y = 0)))$  which can be simplified to  $(y < 0 \vee (\neg(y < 0) \wedge \neg(3 * x + y = 0)))$ .

The following result states that  $\alpha(\mathbf{y}, \mathbf{e})$ ,  $\beta^+(e)$  and  $\beta^-(e)$  provide encodings in Linear Arithmetics of the semantics of linear expressions:

**Lemma 3.** *Let  $y$  be a variable and  $e$  a generalized linear (boolean) expression, such that  $y$  does not occurs in  $e$ , then for any valuation  $\omega$ :*

- $\models_{\omega} \beta^+(e)$  iff  $d \in \bar{\omega}(e)$ , for some  $d \neq 0$ ;
- $\models_{\omega} \beta^-(e)$  iff  $0 \in \bar{\omega}(e)$ ;
- if  $e$  is a generalized linear expression, then  $\bar{\omega}(e) = \{d \in \mathcal{D} : \models_{\omega[d/y]} \alpha(y, e)\}$ .

*Proof.* The proof is by induction on the structure of the expression  $e$ . The base case (for  $e$  a constant or a variable) is straightforward. Therefore, here we consider the step case only.

- $e = (e_1 \text{ op } e_2)$ , with  $\text{op} \in \{\ast, +, >=, =, <, >, ==, !=\}$ . Let  $\omega$  be a arbitrary valuation, and  $\bar{d} \in \bar{\omega}(e)$ . By the definition of  $\bar{\omega}$ ,  $\bar{d} = d_1 \text{ op } d_2$ , for some  $d_i \in \bar{\omega}(e_i)$  ( $i = 1, 2$ ). Moreover, both  $e_1$  and  $e_2$  are linear expressions. Therefore, by induction hypothesis we know that:

$$\bar{\omega}(e_i) = \{d \in \mathcal{D} : \models_{\omega[d/z_i]} \alpha(z_i, e_i)\}$$

for  $i = 1, 2$ . Thus, by the definitions of  $\alpha(z_i, e_i)$ , we have that for  $i = 1, 2$ :

$$\models_{\omega[d_i/z_i]} \exists U_i. (z_i = \bar{n}e_i \wedge \bigwedge \bar{B}_i)^*$$

for some  $e_i \rightarrow (\bar{B}_i, \bar{n}e_i)$ .

Without loss of generality, we may assume that  $z_1 \neq z_2$  and  $U_1 \cap U_2 = \emptyset$  (otherwise we can rename all the variables in  $U_2 \cup \{z_2\}$ ). Since no variable in  $U_1$  occurs free in  $(z_2 = \bar{n}e_2 \wedge \bigwedge \bar{B}_2)^*$  and no variable in  $U_2$  occurs free in  $(z_1 = \bar{n}e_1 \wedge \bigwedge \bar{B}_1)^*$ , it follows that:

$$\models_{\omega[d_1/z_1, d_2/z_2]} \exists U_1 \exists U_2. (z_1 = \bar{n}e_1 \wedge z_2 = \bar{n}e_2 \wedge \bigwedge \bar{B}_1 \wedge \bigwedge \bar{B}_2)^*$$

Taking  $U = U_1 \cup U_2$ , this is equivalent to

$$\models_{\omega[d_1/z_1, d_2/z_2]} \exists U. (z_1 = \bar{n}e_1 \wedge z_2 = \bar{n}e_2 \wedge \bigwedge \bar{B}_1 \wedge \bigwedge \bar{B}_2)^* \quad (1)$$

By the definition of  $e^+$ , it clearly follows that:

$$\models_{\omega[d_1/z_1, d_2/z_2]} \exists U. ((z_1 \text{ op } z_2)^+ \wedge z_1 = \bar{n}e_1 \wedge z_2 = \bar{n}e_2 \wedge \bigwedge \bar{B}_1 \wedge \bigwedge \bar{B}_2)^* \text{ iff } d_1 \text{ op } d_2 \neq 0.$$

and

$$\models_{\omega[d_1/z_1, d_2/z_2]} \exists U. ((z_1 \text{ op } z_2)^- \wedge z_1 = \bar{n}e_1 \wedge z_2 = \bar{n}e_2 \wedge \bigwedge \bar{B}_1 \wedge \bigwedge \bar{B}_2)^* \text{ iff } d_1 \text{ op } d_2 = 0.$$

Since  $\bar{d} = d_1 \text{ op } d_2$  and neither  $z_1$  nor  $z_2$  occurs in  $\bar{n}e_1$ ,  $\bar{n}e_2$ ,  $\bar{B}_1$  or  $\bar{B}_2$ , the following hold:

$$\models_{\omega} \exists U. ((\bar{n}e_1 \text{ op } \bar{n}e_2)^+ \wedge \bigwedge \bar{B}_1 \wedge \bigwedge \bar{B}_2)^* \text{ iff } \bar{d} \neq 0$$

$$\models_{\omega} \exists U. ((\bar{n}e_1 \text{ op } \bar{n}e_2)^- \wedge \bigwedge \bar{B}_1 \wedge \bigwedge \bar{B}_2)^* \text{ iff } \bar{d} = 0$$

By one application of the inference rule for linear operators, we know that  $e \rightarrow (\bar{B}_1 \cup \bar{B}_2, \bar{n}e_1 \text{ op } \bar{n}e_2)$ . Therefore, from the definition of  $\beta(e)$ , we immediately conclude that  $\models_{\omega} \beta^+(e)$  iff  $\bar{d} \in \bar{\omega}(e)$  and  $\bar{d} \neq 0$ , and  $\models_{\omega} \beta^-(e)$  iff  $0 \in \bar{\omega}(e)$ .

If, in addition,  $\text{op} \in \{\ast, +\}$ , from (1) follows that

$$\models_{\omega[d_1/z_1, d_2/z_2]} \exists U. (z_1 \text{ op } z_2 = \bar{n}e_1 \text{ op } \bar{n}e_2 \wedge \bigwedge \bar{B}_1 \wedge \bigwedge \bar{B}_2)^*$$

and from that

$$\models_{\omega[\bar{d}/z]} \exists U. (z = \bar{n}e_1 \text{ op } \bar{n}e_2 \wedge \bigwedge \bar{B}_1 \wedge \bigwedge \bar{B}_2)^*$$

where  $z$  is a variable not occurring in  $\bar{n}e_1$ ,  $\bar{n}e_2$ ,  $\bar{B}_1$  or  $\bar{B}_2$ . Therefore, from the definition of  $\alpha(z, e)$ , we immediately conclude that  $\models_{\omega[\bar{d}/z]} \alpha(z, e)$ . Hence,  $\bar{d} \in \{d \in \mathcal{D} : \models_{\omega[\bar{d}/z]} \alpha(z, e)\}$ .

For the other direction, assume that  $\models_{\omega} \beta^+(e)$  iff  $\bar{d} \in \bar{\omega}(e)$  and  $\bar{d} \neq 0$ ,  $\models_{\omega} \beta^-(e)$  iff  $0 \in \bar{\omega}(e)$ , and, if  $e$  is a linear expression, that  $\bar{d} \in \{d \in \mathcal{D} : \models_{\omega[\bar{d}/z]} \alpha(z, e)\}$ . Therefore, by the definitions of  $\beta^+$ ,  $\beta^-$  and  $\alpha$ , we have that

$$\models_{\omega} \beta^+(e) \text{ iff } \models_{\omega} \exists U.(ne^+ \wedge \bigwedge B)^* \text{ for some } (B, ne), \text{ with } e \rightarrow (B, ne_1)$$

$$\models_{\omega} \beta^-(e) \text{ iff } \models_{\omega} \exists U.(ne^- \wedge \bigwedge B)^* \text{ for some } (B, ne), \text{ with } e \rightarrow (B, ne)$$

and, if  $e$  is a linear expression,

$$\models_{\omega[\bar{d}/z]} \exists U.(z = ne \wedge \bigwedge B)^*$$

for some pair  $(B, ne)$ , with  $e \rightarrow (B, ne)$ .

Since  $e = e_1 \text{ op } e_2$ , both  $e_1$  and  $e_2$  are linear expressions and  $B = B_1 \cup B_2$  and  $ne = ne_1 \text{ op } ne_2$ , where  $e_1 \rightarrow (B_1, ne_1)$  and  $e_2 \rightarrow (B_2, ne_2)$ . Therefore, we have

$$\models_{\omega} \exists U.(ne^+ \wedge \bigwedge B)^* \text{ iff } \models_{\omega} \exists U.((ne_1 \text{ op } ne_2)^+ \wedge \bigwedge B_1 \wedge \bigwedge B_2)^* \quad (2)$$

$$\models_{\omega} \exists U.(ne^- \wedge \bigwedge B)^* \text{ iff } \models_{\omega} \exists U.((ne_1 \text{ op } ne_2)^- \wedge \bigwedge B_1 \wedge \bigwedge B_2)^* \quad (3)$$

the right-hand side of (2) (and (3), resp.) are equivalent to following:

$$\models_{\omega} \exists U.((ne_1 \text{ op } ne_2)^+ \wedge \bigwedge B_1 \wedge \bigwedge B_2)^* \quad (4)$$

$$\models_{\omega} \exists U.((ne_1 \text{ op } ne_2)^- \wedge \bigwedge B_1 \wedge \bigwedge B_2)^* \quad (5)$$

Moreover, if  $e$  is a linear expression,

$$\models_{\omega[\bar{d}/z]} \exists U.(z = ne_1 \text{ op } ne_2 \wedge \bigwedge B_1 \wedge \bigwedge B_2)^* \quad (6)$$

From (6) we immediately obtain:

$$\models_{\omega[\bar{d}/z]} \exists U.(z = ne_1 \text{ op } ne_2 \wedge \bigwedge B_1)^* \wedge \exists U.(z = ne_1 \text{ op } ne_2 \wedge \bigwedge B_2)^* \quad (7)$$

Let now  $z_1, z_2$  be two new variables. From (7), there must exist  $d_1, d_2 \in \mathcal{D}$  with  $\bar{d} = d_1 \text{ op } d_2$  and such that:

$$\models_{\omega[\bar{d}/z, d_1/z_1, d_2/z_2]} z = z_1 \text{ op } z_2 \wedge \exists U.(z_1 = ne_1 \wedge z_2 = ne_2 \wedge \bigwedge B_1)^* \wedge \wedge \exists U.(z_1 = ne_1 \wedge z_2 = ne_2 \wedge \bigwedge B_2)^* \quad (8)$$

Similarly, from (4) there must exist  $d_1, d_2 \in \mathcal{D}$  with  $d_1 \text{ op } d_2 \neq 0$  and:

$$\models_{\omega[d_1/z_1, d_2/z_2]} (z_1 \text{ op } z_2)^+ \wedge \exists U.(z_1 = ne_1 \wedge z_2 = ne_2 \wedge \bigwedge B_1)^* \wedge \wedge \exists U.(z_1 = ne_1 \wedge z_2 = ne_2 \wedge \bigwedge B_2)^* \quad (9)$$

and from (5) there must exist  $d_1, d_2 \in \mathcal{D}$  with  $d_1 \text{ op } d_2 = 0$ , and

$$\models_{\omega[d_1/z_1, d_2/z_2]} (z_1 \text{ op } z_2)^- \wedge \exists U.(z_1 = ne_1 \wedge z_2 = ne_2 \wedge \bigwedge B_1)^* \wedge \wedge \exists U.(z_1 = ne_1 \wedge z_2 = ne_2 \wedge \bigwedge B_2)^* \quad (10)$$

Each of the equations (8), (9), (10) implies that:

$$\models_{\omega[d_1/z_1]} \alpha(z_1, e_1) \quad \text{and} \quad \models_{\omega[d_2/z_2]} \alpha(z_2, e_2)$$

By induction hypothesis,  $d_i \in \bar{\omega}(e_i)$  (for  $i = 1, 2$ ). Hence, by the definition of  $\bar{\omega}(e)$ ,  $\bar{d} = d_1 \text{ op } d_2 \in \bar{\omega}(e)$ . Moreover, from (9) we can conclude that  $\bar{d} \neq 0$ , and from (10) we can conclude that  $\bar{d} = 0$ , as required.

–  $e = (b ? e_1 : e_2)$  where, for  $i = 1, 2$ ,  $e_i$  is a linear expression. Let  $\omega$  be a arbitrary valuation, and  $\bar{d} \in \bar{\omega}(e)$ . By induction hypothesis we know that  $\bar{\omega}(e_i) = \{d \in \mathcal{D} : \models_{\omega[d/z_i]} \alpha(e_i, z_i)\}$  for  $i = 1, 2$  and  $\models \beta^+(b)$  iff  $d' \in \bar{\omega}(b)$  for some  $d' \neq 0$ , and  $\models \beta^-(b)$  iff  $0 \in \bar{\omega}(b)$ .

By the definition of  $\bar{\omega}$ , either  $\bar{d} \in \bar{\omega}(e_1)$  and  $d' \neq 0 \in \bar{\omega}(b)$ , or  $\bar{d} \in \bar{\omega}(e_2)$  and  $0 \in \bar{\omega}(b)$ . Let us consider the first case (the second case is similar).

Thus, by the definition of  $\alpha(e_i, z_i)$  and  $\beta^+(b)$ , we have that:

$$\models_{\omega[\bar{d}/z_1]} \exists U_1. (z_1 = ne_1 \wedge \bigwedge B_1)^*$$

for some pair  $(B_1, ne_1)$  with  $e_1 \rightarrow (B_1, ne_1)$  and

$$\models_{\omega} \exists U'. (nb'^+ \wedge \bigwedge B')^*$$

for some pair  $(B', nb')$  with  $b \rightarrow (B', nb')$ .

Again, without loss of generality, we may assume  $U_1 \cap U' = \emptyset$ . Since no variable in  $U_1$  occurs free in  $(nb'^+ \wedge \bigwedge B')^*$  and no variable in  $U'$  occurs free in  $(z_1 = ne_1 \wedge \bigwedge B_1)^*$ , it follows that:

$$\models_{\omega[\bar{d}/z_1]} \exists U_1 \exists U'. (z_1 = ne_1 \wedge \bigwedge B_1 \wedge nb'^+ \wedge \bigwedge B')^* \quad (11)$$

Now, under the assumptions we have made,  $e \rightarrow (B_1 \cup B' \cup \{nb'^+\}, ne_1)$  (by one application of the inference rule for the positive case of conditional expression). Therefore, from the condition above and the definition of  $\alpha(e, z_1)$ , it follows that  $\models_{\omega[\bar{d}/z_1]} \alpha(e, z_1)$ , and, therefore,  $\bar{d} \in \{d \in \mathcal{D} : \models_{\omega[d/z_1]} \alpha(e, z_1)\}$ . Moreover, by the definition of  $e^+$ , equation (11) is equivalent to  $\models_{\omega} \exists U_1 \exists U'. (ne_1^+ \wedge \bigwedge B_1 \wedge nb'^+ \wedge \bigwedge B')^*$  iff  $\bar{d} \neq 0$ . Hence, from the definition of  $\beta^+(\cdot)$  and the reduction rules for conditional expressions, we can conclude that  $\models_{\omega} \beta^+(e)$  iff  $\bar{d} \in \bar{\omega}(e)$ , for some  $\bar{d} \neq 0$ . In a very similar way we can conclude also that  $\models_{\omega} \beta^-(e)$  iff  $0 \in \bar{\omega}(e)$ .

For the other direction consider any  $\bar{d} \in \{d \in \mathcal{D} : \models_{\omega[d/z]} \alpha(e, z)\}$ . Then,

$$\models_{\omega[\bar{d}/z]} \exists U. (z = \bar{ne} \wedge \bigwedge \bar{B})^*$$

for some  $e \rightarrow (\bar{B}, \bar{ne})$ . There are two cases:

- $\bar{ne} = ne_1$  for some  $e_1 \rightarrow (B_1, ne_1)$ , and  $\bar{B} = B_1 \cup B' \cup nb'^+$ , for some  $b \rightarrow (B', nb')$ .
- $\bar{ne} = ne_2$  for some  $e_2 \rightarrow (B_2, ne_2)$ , and  $\bar{B} = B_1 \cup B' \cup nb'^-$ , for some  $b \rightarrow (B', nb')$ .

Let us consider the first case (the proof in the second case is similar). Then,

$$\models_{\omega[\bar{d}/z]} \exists U. (z = ne_1 \wedge \bigwedge B_1 \wedge \bigwedge B' \wedge nb'^+)^* \quad (12)$$

and therefore

$$\models_{\omega[\bar{d}/z]} \exists U. (z = ne_1 \wedge \bigwedge B_1)^* \wedge \exists U. (nb'^+ \wedge \bigwedge B')^*$$

from two conjuncts we obtain  $\models_{\omega[\bar{d}/z]} \alpha(z, e_1)$  and  $\models_{\omega} \beta^+(b)$ , and, by the inductive hypothesis,  $\bar{d} \in \bar{\omega}(e_1)$  and  $d' \in \bar{\omega}(b)$  with  $d' \neq 0$ .

As a consequence, by the definition of  $\bar{\omega}(e)$ ,  $\bar{d} \in \bar{\omega}(e_1) \subseteq \bar{\omega}(e)$ .

Moreover, equation (12) is equivalent to

$$\models_{\omega} \exists U. (ne_1^+ \wedge \bigwedge B_1 \wedge \bigwedge B' \wedge nb'^+)^*$$

if and only if  $\bar{d} \neq 0$ , and equation (12) is equivalent to

$$\models_{\omega} \exists U. (ne_1^- \wedge \bigwedge B_1 \wedge \bigwedge B' \wedge nb'^+)^*$$

if and only if  $\bar{d} = 0$ . From that immediately follows that  $\models_{\omega} \beta^+(e)$  iff  $\bar{d} \in \bar{\omega}(e)$  for some  $\bar{d} \neq 0$ , and  $\models_{\omega} \beta^-(e)$  iff  $0 \in \bar{\omega}(e)$ .

Let  $e$  be an expression, by  $e'$  we denote the expression obtained from  $e$  by replacing each variable, say  $v$ , with the corresponding primed version (i.e.  $v'$ ). This notation is extended over sets (tuples) of expressions in the obvious way, i.e.  $E' = \{e' : e \in E\}$  ( $\mathbf{e}' = \langle e'_1, \dots, e'_n \rangle$  if  $\mathbf{e} = \langle e_1, \dots, e_n \rangle$ , resp.).

**Corollary 3.** *Let  $\mathbf{y}'$  and  $\mathbf{e}$  be  $k$ -uples of primed variables and generalized linear expressions (resp.) with  $k > 0$  such that no variable in  $\mathbf{y}'$  occurs in  $\mathbf{e}$ . Let  $\omega : V \rightarrow \mathcal{D}$  and  $\omega' : V' \rightarrow \mathcal{D}$ . The following fact holds:  $\models_{\omega \cup \omega'} \alpha(\mathbf{y}', \mathbf{e})$  if and only if  $\omega'(\mathbf{y}') \in \bar{\omega}(\mathbf{e})$*

We now define a symbolic counterpart of the  $\Rightarrow$  relation which is amenable to mechanization.

A *Disjunctive Linear Constraint*  $D$  (DLC for short) is a formula of the form  $D = \bigvee_i \exists U. \bigwedge_j c_{ij}$ , where  $c_{ij}$  are linear constraints. The symbol  $\perp$  stands for an unsatisfiable linear constraint. An *Abstract Disjunctive Linear Constraint of arity  $n$*  (ADLC for short) is an expression of the form  $\lambda \mathbf{x}. \lambda \mathbf{x}'. D$ , where  $D$  is a DLC and  $\mathbf{x}$  is an  $n$ -uple comprising the free variables occurring in  $D$ .<sup>9</sup> We define the following operations on DLCs:

- *Application.* Let  $\lambda \mathbf{x} \mathbf{x}'. D$  be an ADLC of arity  $n$  and  $\mathbf{s}$  and  $\mathbf{t}$  be  $n$ -uples of linear expressions. The *application* of  $\delta = \lambda \mathbf{x} \mathbf{x}'. D$  to  $(\mathbf{s}, \mathbf{t})$ , in symbols  $\delta(\mathbf{s}, \mathbf{t})$ , is the DLC obtained by simultaneously replacing from  $D$  the variables in  $\mathbf{x}$  ( $\mathbf{x}'$ ) with the corresponding element of  $\mathbf{s}$  ( $\mathbf{t}$  resp.).
- *Conjunction and Disjunction.* Let  $D_1$  and  $D_2$  be DLCs, then  $D_1 \sqcap D_2$  and  $D_1 \sqcup D_2$  are any DLCs logically equivalent to  $D_1 \wedge D_2$  and  $D_1 \vee D_2$  respectively. These operations are extended to ADLCs as follows. Let  $\delta_1$  and  $\delta_2$  be ADLCs of the same arity, then  $\delta_1 \sqcap \delta_2 = \lambda \mathbf{x} \mathbf{x}'. (\delta_1(\mathbf{x}, \mathbf{x}') \sqcap \delta_2(\mathbf{x}, \mathbf{x}'))$  and  $\delta_1 \sqcup \delta_2 = \lambda \mathbf{x} \mathbf{x}'. (\delta_1(\mathbf{x}, \mathbf{x}') \sqcup \delta_2(\mathbf{x}, \mathbf{x}'))$ .
- *Quantifier Elimination.* Let  $D$  be a DLC, then  $\exists \mathbf{x}. D$  is any DLC equivalent to  $D$  obtained by eliminating from  $D$  the variables in  $\mathbf{x}$ .
- *Entailment.* Let  $\delta_1$  and  $\delta_2$  be ADLCs of the same arity, then  $\delta_1 \sqsubseteq \delta_2$  iff all the pairs of valuations satisfying  $\delta_1$  satisfy also  $\delta_2$ , i.e.  $\llbracket \delta_1 \rrbracket \subseteq \llbracket \delta_2 \rrbracket$  where  $\llbracket \delta \rrbracket = \{ \langle \omega, \omega' \rangle : \models_{\omega \cup \omega'} \delta(\mathbf{x}, \mathbf{x}'), \omega : V \rightarrow \mathcal{D}, \omega' : V' \rightarrow \mathcal{D} \}$ .

By  $\mathcal{A}(P)$  we denote the  $N_P$ -indexed family of sets of ADLCs such that  $\mathcal{A}_i(P)$  is the set of ADLCs of arity  $|\text{InScope}_P(i)|$  for all  $i \in N_P$ . Let  $\Delta, \Delta^1 \in \mathcal{A}(P)$ <sup>10</sup>, we define the binary relation  $\Delta \rightarrow \Delta^1$  as follows. Let  $i \in N_P$ ,  $\Delta_j^1 = \Delta_j$  for all  $j \notin \text{Succ}_P(i)$  and

<sup>9</sup> In the sequel we will abbreviate  $\lambda \mathbf{x}. \lambda \mathbf{x}'. D$  with  $\lambda \mathbf{x} \mathbf{x}'. D$ .

<sup>10</sup> With an abuse of notation we write  $\Delta \in \mathcal{A}(P)$  for  $\Delta_i \in \mathcal{A}_i(P)$ , for each  $i \in N_P$ .

- if  $i$  corresponds to a skip (;) or return statement, then

$$\Delta_{\text{sSucc}_P(i)}^1 = \Delta_{\text{sSucc}_P(i)} \sqcup \Delta_i$$

- if  $i$  corresponds to an assignment  $\mathbf{y}=\mathbf{e}$ , then

$$\Delta_{\text{sSucc}_P(i)}^1 = \Delta_{\text{sSucc}_P(i)} \sqcup \lambda \mathbf{x} \mathbf{x}' . \exists \mathbf{x}' . (\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \alpha(\mathbf{y}'', \mathbf{e}') \sqcap \mathbf{z}'' = \mathbf{z}')$$

where  $\mathbf{x} = \text{InScope}_P(i)$  and  $\mathbf{z} = \text{InScope}_P(i) \setminus \mathbf{y}$ ;<sup>11</sup>

- if  $i$  corresponds to an **if**( $b$ ), **while**( $b$ ), or **assert**( $b$ ) statement, then

$$\begin{aligned} \Delta_{\text{Tsucc}_P(i)}^1 &= \Delta_{\text{Tsucc}_P(i)} \sqcup \lambda \mathbf{x} \mathbf{x}' . (\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \beta^+(b')) \\ \Delta_{\text{Fsucc}_P(i)}^1 &= \Delta_{\text{Fsucc}_P(i)} \sqcup \lambda \mathbf{x} \mathbf{x}' . (\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \beta^-(b')) \end{aligned}$$

with  $\mathbf{x} = \text{InScope}_P(i)$ ;

- if  $i$  corresponds to a procedure call  $pr(\mathbf{e})$ , then

$$\Delta_{\text{sSucc}_P(i)}^1 = \Delta_{\text{sSucc}_P(i)} \sqcup \lambda \mathbf{w} \mathbf{w}'' . (\exists \mathbf{x} \mathbf{x}' . (\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \alpha(\mathbf{y}'', \mathbf{e}') \sqcap \mathbf{g}'' = \mathbf{g}') \sqcap \mathbf{l} = \mathbf{l}'')$$

with  $\mathbf{x} = \text{InScope}_P(i)$ ,  $\mathbf{y} = \text{Formals}_P(pr)$ ,  $\mathbf{w} = \text{InScope}_P(\text{sSucc}_P(i))$ ,  $\mathbf{g} = \text{Globals}_P$ , and  $\mathbf{l} = \text{Locals}_P(\text{sSucc}_P(i))$ ;

- if  $i = \text{Exit}_P(pr)$  then

$$\Delta_j^1 = \Delta_j \sqcup \lambda \mathbf{w} \mathbf{w}'' . \exists \mathbf{w}''' . (\Delta_k(\mathbf{w}, \mathbf{w}''') \sqcap \mathbf{y}'' = \mathbf{y}''' \sqcap \exists \mathbf{x}' \mathbf{z}'' . (\Delta_i(\mathbf{x}', \mathbf{x}'') \sqcap \alpha(\mathbf{f}', \mathbf{e}''') \sqcap \mathbf{g}' = \mathbf{g}'''))$$

if  $j \in \text{Succ}_P(i)$  and  $k$  such that  $s_k = pr(\mathbf{e})$  and  $\text{RetPt}(k) = j$ ,  $\mathbf{w} = \text{InScope}_P(k)$ ,  $\mathbf{f} = \text{Formals}(pr)$ ,  $\mathbf{y} = \text{Locals}_P(k)$ ,  $\mathbf{x} = \text{InScope}_P(i)$ ,  $\mathbf{z} = \text{Locals}_P(i)$ .

For example, let  $s_i$  be the parallel assignment  $y_1, y_2 = ((y_1 > 0) ? y_1 + 1 : u), y_2 + 1$  and  $\text{InScope}_P(i) = \{x, y_1, y_2\}$ . Hence,  $\mathbf{e} = \langle (y_1 > 0) ? y_1 + 1 : u, y_2 + 1 \rangle$ ,  $\mathbf{y} = \langle y_1, y_2 \rangle$  and  $\mathbf{z} = \langle x \rangle$ . Then,  $\alpha(\mathbf{y}'', \mathbf{e}') = ((y_1'' = y_1' + 1 \wedge y_2'' = y_2' + 1 \wedge y_1' > 0) \vee \exists u_1 . (y_1' = u_1 \wedge y_2'' = y_2' + 1 \wedge \neg y_1' > 0))$ . This can be simplified to  $\alpha(\mathbf{y}'', \mathbf{e}') = ((y_1'' = y_1' + 1 \wedge y_2'' = y_2' + 1 \wedge y_1' > 0) \vee (y_2'' = y_2' + 1 \wedge \neg y_1' > 0))$ .

Therefore, the new set of path edges added to  $\Delta_{\text{sSucc}_P(i)}$  is represented by the following ADLC  $\lambda x, y_1, y_2, x'', y_1'', y_2'' . \exists x', y_1', y_2' . (\Delta_i(\langle x, y_1, y_2 \rangle, \langle x', y_1', y_2' \rangle) \sqcap ((y_1'' = y_1' + 1 \wedge y_2'' = y_2' + 1 \wedge y_1' > 0 \wedge x'' = x') \vee (y_2'' = y_2' + 1 \wedge \neg y_1' > 0 \wedge x'' = x')))$ .

As a further example, let  $s_i$  be the statement **if**( $y > 0 \&\& \mathbf{y} = 2 * \mathbf{u}$ ) and  $\text{InScope}_P(i) = \{x, y\}$ , then (after some simplifications)  $\beta^+(b') = \exists u_1 . (y' > 0 \wedge y' = 2 * u_1)$  and  $\beta^-(b') = \exists u_1 . (y' > 0 \wedge \neg y' = 2 * u_1) \vee (\neg y' > 0)$ . The new set of path edges added to  $\Delta_{\text{Tsucc}_P(i)}$  (to  $\Delta_{\text{Fsucc}_P(i)}$ , resp.) is represented by the following ADLC  $\lambda x . x' . y . y' . \exists u_1 . (\Delta_i(\langle x, y \rangle, \langle x', y' \rangle) \sqcap (y' > 0 \wedge y' = 2 * u_1 \wedge x' = x))$  ( $\lambda x . x' . y . y' . \exists u_1 . (\Delta_i(\langle x, y_1, y_2 \rangle, \langle x', y_1', y_2' \rangle) \sqcap ((y' > 0 \wedge \neg y' = 2 * u_1 \wedge x' = x) \vee (\neg y' > 0 \wedge x' = x)))$ , resp.).

Let  $\Delta \in \mathcal{A}(P)$ , we define  $\llbracket \Delta \rrbracket$  as the  $N_P$ -indexed family of sets  $\llbracket \Delta \rrbracket_i$  such that  $\llbracket \Delta \rrbracket_i = \llbracket \Delta_i \rrbracket$ , for all  $i \in N_P$ .

**Theorem 4 (Soundness and Completeness).** *Let  $P$  be a linear program and  $\Delta, \Delta^1 \in \mathcal{A}(P)$ . The following fact holds:  $\Delta \rightarrow^* \Delta^1$  if and only if  $\llbracket \Delta \rrbracket \Rightarrow \llbracket \Delta^1 \rrbracket$ .*

<sup>11</sup> We abbreviate  $x'_1 = x_1 \wedge \dots \wedge x'_n = x_n$  with  $\mathbf{x}' = \mathbf{x}$ , where  $\mathbf{x} = \langle x_1, \dots, x_n \rangle$ .

*Proof.* It suffices to observe that by replacing each ADLC  $\delta$  with  $\llbracket \delta \rrbracket$  throughout the definition of  $\rightarrow$  we obtain the definition of  $\Rightarrow$ . Let  $i \in N_P$ . If  $j \notin \text{Succ}_P(i)$ , then  $\Delta_j^1 = \Delta_j$  becomes  $\llbracket \Delta_j^1 \rrbracket = \llbracket \Delta_j \rrbracket$  and therefore  $\llbracket \Delta^1 \rrbracket_j = \llbracket \Delta \rrbracket_j$ .

- If  $s_i$  is a `;` statement (or a `return ;`), then  $\Delta_{\text{sSucc}_P(i)}^1 = \Delta_{\text{sSucc}_P(i)} \sqcup \Delta_i$  becomes  $\llbracket \Delta^1 \rrbracket_{\text{sSucc}_P(i)} = \llbracket \Delta \rrbracket_{\text{sSucc}_P(i)} \cup \llbracket \Delta \rrbracket_i$ ;
- if  $s_i$  an assignment  $\mathbf{y}=\mathbf{e}$  then  $\Delta_{\text{sSucc}_P(i)}^1 = \Delta_{\text{sSucc}_P(i)} \sqcup \Delta^*$  and we must show that  $\llbracket \Delta^* \rrbracket = \{\langle \omega_e, \omega_i[\mathbf{d}/\mathbf{y}] \rangle : \langle \omega_e, \omega_i \rangle \in \llbracket \Delta \rrbracket_i, \mathbf{d} \in \bar{\omega}_i(\mathbf{e})\}$  where  $\Delta^* = \lambda \mathbf{x} \mathbf{x}'. (\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \alpha(\mathbf{y}'', \mathbf{e}') \sqcap \mathbf{z}'' = \mathbf{z}')$ , where  $\mathbf{x} = \text{InScope}_P(i)$  and  $\mathbf{z} = \text{InScope}_P(i) \setminus \mathbf{y}$ . By definition  $\llbracket \Delta^* \rrbracket$  is equal to

$$\{\langle \omega_e, \omega_j \rangle : \models \exists \mathbf{x}'. (\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \alpha(\mathbf{y}'', \mathbf{e}') \sqcap \mathbf{z}'' = \mathbf{z}')\} \quad (13)$$

By Corollary 3, (13) is equivalent to:

$$\{\langle \omega_e, \omega_j \rangle : \langle \omega_e, \omega_i \rangle \in \llbracket \Delta \rrbracket_i, \omega_j(\mathbf{y}) \in \bar{\omega}_i(\mathbf{e}), \omega_j(\mathbf{z}) = \omega_i(\mathbf{z})\} \quad (14)$$

and (14) can be finally simplified to:

$$\{\langle \omega_e, \omega_i[\mathbf{d}/\mathbf{y}] \rangle : \langle \omega_e, \omega_i \rangle \in \llbracket \Delta \rrbracket_i, \mathbf{d} \in \bar{\omega}_i(\mathbf{e})\}$$

- if  $i$  corresponds to an `if(b)`, `while(b)`, or `assert(b)` statement, then  $\Delta_{\text{Tsucc}_P(i)}^1 = \Delta_{\text{Tsucc}_P(i)} \sqcup \Delta^*$  (the case of  $\Delta_{\text{FSucc}_P(i)}^1$  is symmetric) and we must show that  $\llbracket \Delta^* \rrbracket = \{\langle \omega_e, \omega_i \rangle : \langle \omega_e, \omega_i \rangle \in \llbracket \Delta \rrbracket_i, d \in \bar{\omega}_i(b) \text{ and } d \neq 0\}$ , where  $\Delta^* = \lambda \mathbf{x} \mathbf{x}'. (\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \beta^+(b))$ , and  $\mathbf{x} = \text{InScope}_P(i)$ . By definition  $\llbracket \Delta^* \rrbracket$  is equal to<sup>12</sup>

$$\{\langle \omega_e, \omega_j \rangle : \models_{\omega_e \cup \omega'_j} \Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \beta^+(b)\} \quad (15)$$

where  $j = \text{Tsucc}_P(i)$ . By definition,  $\models_{\omega_e \cup \omega'_j} \Delta_i(\mathbf{x}, \mathbf{x}')$  if and only if  $\langle \omega_e, \omega_j \rangle \in \llbracket \Delta \rrbracket_i$ . Therefore, (15) is equivalent to

$$\{\langle \omega_e, \omega_i \rangle : \langle \omega_e, \omega_i \rangle \in \llbracket \Delta \rrbracket_i \text{ and } \models_{\omega_e \cup \omega'_i} \beta^+(b)\} \quad (16)$$

Finally, by Lemma 3, (16) is equivalent to:

$$\{\langle \omega_e, \omega_i \rangle : \langle \omega_e, \omega_i \rangle \in \llbracket \Delta \rrbracket_i, d \in \bar{\omega}_i(b) \text{ and } d \neq 0\} \quad (17)$$

- if  $i$  corresponds to a procedure call `pr(e)`, then  $\Delta_{\text{sSucc}_P(i)}^1 = \Delta_{\text{sSucc}_P(i)} \sqcup \Delta^*$ , and we must show that  $\llbracket \Delta^* \rrbracket = \{\langle \omega_j, \omega_j \rangle : \langle \omega_e, \omega_i \rangle \in \llbracket \Delta \rrbracket_i, \omega_j(\mathbf{g}) = \omega_i(\mathbf{g}), \omega_j(\mathbf{y}) \in \bar{\omega}_i(\mathbf{e})\}$ , where  $j = \text{sSucc}_P(i)$ ,  $\Delta^* = \lambda \mathbf{w} \mathbf{w}''. (\exists \mathbf{x} \mathbf{x}'. (\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \alpha(\mathbf{y}'', \mathbf{e}') \sqcap \mathbf{g}'' = \mathbf{g}') \sqcap \mathbf{l} = \mathbf{l}'')$ , and  $\mathbf{x} = \text{InScope}_P(i)$ ,  $\mathbf{y} = \text{Formals}_P(\text{pr})$ ,  $\mathbf{w} = \text{InScope}_P(j)$ ,  $\mathbf{g} = \text{Globals}_P$ , and  $\mathbf{l} = \text{Locals}_P(\text{sSucc}_P(i))$ . By definition  $\llbracket \Delta^* \rrbracket$  is equal to

$$\{\langle \omega_h, \omega_j \rangle : \models_{\omega_h \cup \omega'_j} \exists \mathbf{x} \mathbf{x}'. (\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \alpha(\mathbf{y}'', \mathbf{e}') \sqcap \mathbf{g}'' = \mathbf{g}') \sqcap \mathbf{l} = \mathbf{l}''\} \quad (18)$$

By the last conjunct ( $\mathbf{l} = \mathbf{l}''$ ) in the DLC above, (18) is equivalent to

$$\{\langle \omega_j, \omega_j \rangle : \models_{\omega_j \cup \omega'_j} \exists \mathbf{x} \mathbf{x}'. (\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \alpha(\mathbf{y}'', \mathbf{e}') \sqcap \mathbf{g}'' = \mathbf{g}')\} \quad (19)$$

By Corollary 3, (19) is equivalent to:

$$\{\langle \omega_j, \omega_j \rangle : \langle \omega_e, \omega_i \rangle \in \llbracket \Delta \rrbracket_i, \omega_j(\mathbf{y}) \in \bar{\omega}_i(\mathbf{e}) \text{ and } \omega_j(\mathbf{g}) = \omega_i(\mathbf{g})\} \quad (20)$$

<sup>12</sup> Given  $\omega : V \rightarrow \mathcal{D}$ , with  $\omega'$  ( $\omega''$ , resp.) we denote the valuation  $\omega' : V' \rightarrow \mathcal{D}$  ( $\omega'' : V'' \rightarrow \mathcal{D}$ , resp.) such that  $\omega'(x') = \omega(x)$ , for all  $x' \in V'$  ( $\omega''(x'') = \omega(x)$ , for all  $x'' \in V''$ ).

– if  $i = \text{Exit}_P(pr)$  then  $\Delta_j^1 = \Delta_j \sqcup \Delta^*$ , where  $j \in \text{Succ}_P(i)$ , and we must show that

$$\llbracket \Delta^* \rrbracket = \{ \langle \omega_e, \omega_j \rangle : \langle \omega_e, \omega_k \rangle \in \llbracket \Delta \rrbracket_k, \langle \omega_h, \omega_i \rangle \in \llbracket \Delta \rrbracket_i, \omega_k(\mathbf{g}) = \omega_h(\mathbf{g}), \\ \omega_j(\mathbf{y}) = \omega_k(\mathbf{y}), \omega_h(\mathbf{f}) \in \bar{\omega}_k(\mathbf{e}), \text{ and } \omega_j(\mathbf{g}) = \omega_i(\mathbf{g}) \}$$

with  $k$  such that  $s_k = pr(\mathbf{e})$  and  $\text{RetPt}(k) = j$ ,  $\mathbf{w} = \text{InScope}_P(k)$ ,  $\mathbf{y} = \text{Locals}_P(k)$ ,  $\mathbf{f} = \text{Formals}(pr)$ ,  $\mathbf{x} = \text{InScope}_P(i)$ ,  $\mathbf{z} = \text{Locals}_P(i)$ , and

$$\Delta^* = \lambda \mathbf{w} \mathbf{w}'' . \exists \mathbf{w}''' . (\Delta_k(\mathbf{w}, \mathbf{w}''') \sqcap \mathbf{y}'' = \mathbf{y}''' \sqcap \\ \exists \mathbf{x}' \mathbf{z}'' . (\Delta_i(\mathbf{x}', \mathbf{x}'') \sqcap \alpha(\mathbf{f}', \mathbf{e}''') \sqcap \mathbf{g}' = \mathbf{g}'''))$$

By definition we have that:

$$\llbracket \Delta^* \rrbracket = \{ \langle \omega_e, \omega_j \rangle : \models_{\omega_e \cup \omega_j''} \exists \mathbf{w}''' . (\Delta_k(\mathbf{w}, \mathbf{w}''') \sqcap \mathbf{y}'' = \mathbf{y}''' \sqcap \\ \sqcap \exists \mathbf{x}' \mathbf{z}'' . (\Delta_i(\mathbf{x}', \mathbf{x}'') \sqcap \alpha(\mathbf{f}', \mathbf{e}''') \sqcap \mathbf{g}' = \mathbf{g}''')) \}$$

Therefore, a pair  $\langle \omega_e, \omega_j \rangle \in \llbracket \Delta^* \rrbracket$  if and only if there exists some valuation  $\omega_k$  such that  $\langle \omega_e, \omega_k \rangle \in \llbracket \Delta_k \rrbracket$  with  $\omega_j(\mathbf{y}) = \omega_k(\mathbf{y})$  (since  $\models_{\omega_j'' \cup \omega_k'''} \mathbf{y}''' = \mathbf{y}''$  must hold), and such that

$$\models_{\omega_j'' \cup \omega_k'''} \exists \mathbf{x}' \mathbf{z}'' . (\Delta_i(\mathbf{x}', \mathbf{x}'') \sqcap \alpha(\mathbf{f}', \mathbf{e}''') \sqcap \mathbf{g}' = \mathbf{g}''') \quad (21)$$

On the other hand, by definition of  $\llbracket \Delta_i \rrbracket$  and Lemma 3, Equation (21) holds if and only if there exist two valuations  $\omega_h$  and  $\omega_i$  such that  $\langle \omega_h, \omega_i \rangle \in \llbracket \Delta_i \rrbracket$ ,  $\omega_h(\mathbf{f}) \in \bar{\omega}_k(\mathbf{e})$ ,  $\omega_h(\mathbf{g}) = \omega_k(\mathbf{g})$  (since it must hold  $\models_{\omega_h' \cup \omega_k'''} \mathbf{g}' = \mathbf{g}'''$ ), and  $\omega_j(\mathbf{g}) = \omega_i(\mathbf{g})$  (since  $\mathbf{x}'' = \mathbf{z}'' \cup \mathbf{g}''$ , and the variables  $\mathbf{g}''$  in equation (21) are not quantified away).

Summarizing, from the above reasoning, we obtain that

$$\llbracket \Delta^* \rrbracket = \{ \langle \omega_e, \omega_j \rangle : \langle \omega_e, \omega_k \rangle \in \llbracket \Delta \rrbracket_k, \langle \omega_h, \omega_i \rangle \in \llbracket \Delta \rrbracket_i, \omega_k(\mathbf{g}) = \omega_h(\mathbf{g}), \\ \omega_j(\mathbf{y}) = \omega_k(\mathbf{y}), \omega_h(\mathbf{f}) \in \bar{\omega}_k(\mathbf{e}), \text{ and } \omega_j(\mathbf{g}) = \omega_i(\mathbf{g}) \}$$

as required.

Let  $\Delta^0 \in \mathcal{A}(P)$  such that  $\Delta_1^0 = \lambda \mathbf{x} \mathbf{x}' . \mathbf{x}' = \mathbf{x}$  with  $\mathbf{x} = \text{InScope}_P(1)$  and  $\Delta_i^0 = \lambda \mathbf{x}_i \mathbf{x}'_i . \perp$  with  $\mathbf{x}_i = \text{InScope}_P(i)$  for all  $i \in N_P \setminus \{1\}$ .

**Corollary 4.** *Let  $\Delta^0$  be defined as above and  $\Delta^1$  such that  $\Delta^0 \rightarrow^* \Delta^1$ . The following fact holds:  $i$  is reachable if and only if  $\llbracket \Delta_i^1 \rrbracket \neq 0$ .*

## 7 Implementation and experimental results

We have extended the model checking procedure for linear programs of EUREKA along the lines discussed in Section 6 so that it now supports the analysis of linear programs extended with the  $\mathbf{u}$  symbol and conditional expressions. Quantifier elimination is implemented by means of the Fourier-Motzkin method [12]. Entailment checking is done by using ICS v2.0 [13] as a decision procedure for the boolean combination of linear arithmetic constraints. This is done by reducing the problem of determining whether  $\delta_1 \sqsubseteq \delta_2$  holds to that of determining the unsatisfiability of the formula  $\delta_1(\mathbf{c}, \mathbf{c}') \wedge \neg \delta_2(\mathbf{c}, \mathbf{c}')$ , where  $\delta_1$  and

$\delta_2$  are ADLCs of arity  $n$  and  $\mathbf{c}$  is an  $n$ -uple of distinct constants. In order to assess the effectiveness of the approach we have also developed in EUREKA [7] a prototype implementation of the counterexample-guided abstraction refinement procedure outlined in Section 2. In order to check the satisfiability of the formulae encoding the feasibility of the traces generated by the model checker (see Section 2) we use CVC Lite [14] as it is a complete prover for the union of the theory of arrays and linear arithmetics.

Preliminary experiments confirm the effectiveness of the proposed approach. For instance, EUREKA readily concludes that the program in Figure 1 is safe. Quite interestingly Blast incorrectly reports that the same program is unsafe. This is due to the fact that Blast models any “[...] expression  $p + i$ , where  $p$  is a pointer and  $i$  is an integer, as yielding a pointer value that points to the object pointed to by  $p$ ” [3]. Since in the C language  $\mathbf{a}[\mathbf{i}]$  is a shorthand for  $\mathbf{*}(\mathbf{a}+\mathbf{i})$ , this means that all array elements are undistinguishable for Blast during analysis. Since SLAM deals with pointers in the same way, we expect it should exhibit the same behaviour, but we could not verify this since the SLAM toolkit is not publicly available. On other linear programs with arrays (successfully analysed by EUREKA) Blast reports an error due to the incapacity to discover new predicates for the abstraction.

We also compared EUREKA with CBMC.<sup>13</sup> At the core of CBMC lies a procedure that encodes the (bounded) verification problem for C programs into a SAT formula. The number of propositional variables in the SAT formula generated by CBMC grows linearly with the size of the arrays defined in the input program. In contrast EUREKA, by considering the array elements in an incremental and counterexample-driven manner, uses—in many cases of interest—time and memory resources independently from the size of the arrays defined in the input program. (The program in Figure 1 is a typical example of this.) More in general, we tested EUREKA and Blast on a number of C programs that make use of arrays and that we use as regression tests for EUREKA. The results are summarized in Table 2.<sup>14</sup> The reader may refer to the EUREKA project webpage (URL: <http://www.ai.dist.unige.it/eureka>) for a detailed description of the experimental results. Array out of bounds properties are not shown here, but it is easy to see that they can be checked by instrumenting the code by putting `assert( $e < \dim(a)$ )  $\wedge$  ( $e \geq 0$ )` before any occurrence of  $a[e]$ , for any  $a \in A$  in the program.

## 8 Related Work

BLAST and SLAM are the classical points of reference for describing the successful approach of the abstraction/refinement paradigm. They have mainly been used to verify (even huge) security properties of device drivers written in C, which are particularly control-intensive (as an opposite to data-intensive) programs. As shown in this paper and in [7] as well, these tools seem to encounter some trouble in reasoning about (even small) data-intensive programs. The MAGIC tool also exploits boolean abstraction and refinement of C programs, with the extra burden that the user has to supply the specification (in an automata-based language) of the behaviour that the program has to obey. CBMC [9] is the bounded model checker for C programs that we used for a comparison with EUREKA (section 7 describes the details).

<sup>13</sup> CBMC uses a “physical” model of memory: every integer variable is modeled as a word of  $n$  bits, where  $n$  is either 8, 16, 32, or 64. The default (used for our experiments) is  $n = 32$ .

<sup>14</sup> The *mout* results of CBMC depend on both time and resources spent in finding the proper unwinding assertions in order to automatically determine a safe bound for model checking the program. This limitation can be overcome by setting a bound at command line. Of course then, bounded model checking may lead to incomplete answers from the tool.

**Table 2.** Results of the experimental analysis

Program	EUREKA	BLAST	CBMC
array_init.c	safe	error	safe
array_init_assign.c	safe	unsafe	safe
complex_guard.c	safe	unsafe	safe
simple_swap.c	safe	safe	safe
sequential_swap_call.c	safe	error	safe
simple_array_inversion.c	safe	unsafe	safe
bubblesort_inner_loop.c	unsafe	error	unsafe
bubblesort.c	unsafe	safe	unsafe
array_max.c	safe	error	safe
wrong_loop.c	unsafe	error	mout
loop_on_input.c	unsafe	unsafe	mout
simple_control_on_input.c	unsafe	unsafe	mout

**Legenda:**

- *error*: the tool halts and reports that it cannot find a suitable refinement.
- *safe*: no error state can be reached by the program.
- *unsafe*: an error state may be reached by the program.
- *mout*: the tool goes in memory out.

Flanagan deeply examines the connection between Constraint Logic Programs (CLPs) and imperative and object oriented programs [15–17]. Particularly, in [17] CLPs are imbedded in the (boolean) abstraction/refinement paradigm in order to combine and unify theorem proving (performed with  $CLP(\mathcal{D})$ ) and boolean model checking (performed using  $CLP(\mathcal{B})$ ).

Choi et al. [18] investigate domain abstraction (that is, an abstraction over the input domain of a system) based on data equivalence and trajectory reduction, and show that, assuming the input system being totally deterministic, the abstracted system *simulates* the original system.

Abstract Interpretation [19–21] not only is used to statically verify properties of systems, but it also represents the semantic model of abstraction and of refinement [22] in the CEGAR framework. In the context of Abstract Interpretation several verification tools have been developed, such as [23] for invariant discovery, and [24] for the detection of run-time errors.

A great number of verification tools and algorithms have been developed in the last years, for other programming paradigms such as the object oriented one [25, 26], for concurrent systems [27, 28], and many others.

Finally, the reader may refer to [29] for issues about *verification decidability* of (presburger) array programs.

## 9 Conclusions

The work presented in this paper relates to the body of work developed within the counterexample-guided (predicate) abstraction refinement paradigm [4, 3, 30–32, 17]. We have proposed an alternative abstraction and refinement schema for a subset of the C programming language that employs linear arithmetics and arrays, which allows for the analysis of a wide class of imperative programs. At the

core of our approach lies a model checking procedure for Linear Programs augmented with a symbol for undefined values and conditional expressions. We have provided a formal semantics and a symbolic model checking procedure for this extended class of Linear Programs. Preliminary experimental results obtained with our prototype model checker EUREKA indicate that our procedure correctly analyses programs on which other tools either fail or provide incorrect answers.

## References

1. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ansi-c programs using sat (2003)
2. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Proc. of SPIN, Springer New York, Inc. (2001) 103–122
3. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL 2002. (2002) 58–70
4. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: SIGPLAN PLDI Conference. (2001) 203–213
5. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for boolean programs. In: Proc. of SPIN. (2000) 113–130
6. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Proc. of CAV. Number 2102 in LNCS, Springer (2001) 324–336
7. Armando, A., Castellini, C., Mantovani, J.: Software model checking using linear constraints. In: Proc. of ICFEM04. Volume LNCS 3308., Springer (2004)
8. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In Jensen, K., Podelski, A., eds.: Proc. of TACAS 2004. Volume 2988 of LNCS., Springer (2004) 168–176
9. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ansi-c programs using sat. *Formal Methods in System Design* **25** (2004) 105–127
10. Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA (1986)
11. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proc. of POPL '95, ACM Press (1995) 49–61
12. Lassez, J.L., Maher, M.: On Fourier's Algorithm for Linear Arithmetic Constraints. *Journal of Automated Reasoning* **9** (1992) 373–379
13. de Moura, L., Ruess, H., Shankar, N., Rushby, J.: The ICS decision procedure for embedded deduction. In: Proc. of IJCAR 2004, LNCS 3097, Springer (2004)
14. Barrett, C., Berezin, S.: Cvc lite: A new implementation of the cooperating validity checker. In Alur, R., Peled, D.A., eds.: Proc. of CAV 2004. LNCS, Springer (2004)
15. Flanagan, C.: Automatic software model checking using clp. In: Proc. of ESOP. Volume 2618 of LNCS., Springer (2003) 189–203
16. Flanagan, C.: Automatic software model checking via constraint logic. *Sci. Comput. Program.* **50** (2004) 253–270
17. Flanagan, C.: Software model checking via iterative abstraction refinement of constraint logic queries. *CP+CV'04* (2004)
18. Choi, Y., Rayadurgam, S., Heimdahl, M.P.: Automatic abstraction for model checking software systems with interrelated numeric constraints. In: Proc. of ESEC/FSE, ACM Press (2001) 164–174
19. Cousot, P., Cousot, R.: Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. *JTASPEFL '91, Bordeaux. BIGRE* **74** (1991) 107–110
20. Cousot, P.: Verification by abstract interpretation. In Dershowitz, N., ed.: Proc. Int. Symp. on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday, Taormina, Italy, © Springer, Berlin, Germany (2003) 243–268
21. Cousot, P.: Semantic foundations of program analysis. In Muchnick, S., Jones, N., eds.: *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1981) 303–342

22. Cousot, P., Cousot, R.: Refining model checking by abstract interpretation. *Automated Software Engineering* **6** (1999) 69–95
23. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: SAS. (2004) 53–68
24. Mauborgne, L.: ASTRÉE: Verification of absence of run-time error. In Jacquart, R., ed.: Building the information Society (18th IFIP World Computer Congress), The International Federation for Information Processing, Kluwer Academic Publishers (2004) 384–392
25. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from java source code. In: Proc. of the 22nd int. conf. on Software engineering, ACM Press (2000) 439–448
26. Visser, W., Havelund, K., Brat, G., Park, S.: Java pathfinder - second generation of a java model checker (2000)
27. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. In: Proc. of PLDI 2004, ACM Press (2004) 14–24
28. Bultan, T., Gerber, R., Pugh, W.: Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems* **21** (1999) 747–789
29. Suzuki, N., Jefferson, D.: Verification decidability of presburger array programs. *J. ACM* **27** (1980) 191–205
30. S. Graf, H. Saidi: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: Proc. of CAV '97. Volume 1254., Springer (1997) 72–83
31. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. (2000) 154–169
32. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular verification of software components in c. In: Proc. of ICSE, IEEE Computer Society (2003) 385–395