Joint NCC & IIS Bull., Comp. Science, 11 (1999), 15–38

Analysis of equality relationships: proofs and examples

Pavel G. Emelianov*

In this article, we discuss the analysis of equality relationships for program terms. We extend a description of the analysis presented in [7, 9, 10] and give attention to some new aspects which are not widely considered yet. In particular, among other illustrating examples, a program is given for which this analysis without a widening operator diverges.

1. Introduction

Semantic analysis is a powerful tool of building effective and reliable programming systems. In [7, 9, 10] we presented a new kind of the semantic analysis designed in the framework of *abstract interpretation* [4, 5, 6]. This analysis determining the sets of invariant term equalities $t_1 = t_2$ was said to be *the analysis of equality relationships for program terms.*

Unfortunately, we omitted there some important details of the analysis description because of the space limitations of these issues. A complete description of the equality relationship analysis (hereinafter referred to as **ERA**) was given in [8]. However, this work is not widely accessible. In this article, sharing the judgment of the author of [15] about "... deeper consideration of tradeoffs etc. of analyses..." and "... wide presentation of them to the semantic analysis community...", in order to involve the semantic analysis in the real process of program development and to build powerful practical analyzers, we give a detailed description of **ERA**.

It should be mentioned that most static analyses of imperative programs are interested in finding the equalities of some special kind (*value analyses*). In our case, there are no limitations on the type of terms: they represent all expressions computed in the program. This enables the analysis to take into account different aspects of program behavior in a unified way, and thereby the accuracy of analysis increases. This does not mean that **ERA** is a generalization of all other value analyses (except *constant propagation* one), because they use different approaches (semantic domains and transformers) to extracting effectively and precisely the limited classes of semantic properties. In general case, the results of the analyses are not comparable.

*This work was supported by the Russian Foundation for Basic Research (grant No 97-01-00724).

The peculiarity of **ERA** mentioned above allows us to discuss some common properties of the semantic analysis. Such taxonomic properties of the analysis algorithms as the forward/backward/bidirectional propagation of the semantic information, relativeness/attribute independence, context (in)sensitivity, flow (in)sensitivity, scalability and something else are well known. However, it is the author's opinion that a notion of "*interpretability of a semantic analysis*" has not yet been considered adequately. Here the interpretability of analysis means how deeply the properties of primitive operations of the language (arithmetical, logical, etc.) and type information are allowed for analyzing. Obviously, it is closely allied to the properties of flow sensitivity and scalability of the analysis algorithm. One extreme point of view on the interpretability is an approach accepted in the "pure" program schemata theory where any interpretation of functional symbols or type information is not allowed. Unfortunately, the results which can be obtained under this approach are not reasonably strong. Nevertheless, it must be underscored that **ERA** dates back to V. Sabelfeld's works in the program schemata theory [16, 17]. Another extreme leads to the complete description of the program behavior that is not workable, too. It is possible that the interpretability has not been highlighted enough, because the most analysis algorithms take into account the limited classes of primitive operations and type information. For example, the interval analysis is not able to incorporate the congruence properties, etc. Essentially another case is **ERA**. We intend to illustrate the notion of "interpretability of analysis", its importance and usefulness on this example of analysis.

This article is organized as follows: In **Sections 2.1** and **2.2** we describe the semantic properties, concrete and abstract, respectively, which are considered in **ERA**. In **Section 2.3** we discuss some basic operations over the semantic properties used to define the semantic transformers. In **Section 3** we consider a widening operator, and in **Section 4** the complexity of **ERA** is discussed. Finally, **Section 5** presents some results on our implementation of **ERA** and its testing.

2. Properties under consideration

2.1. Concrete properties

A usual choice for the description of the operational semantics is a specification of some transition relationship on the pairs <*control point, state of program memory*> (see, for example, [11, chapter 10]) where the states of program memory are described by mapping the cells of memory into a universe of values. Here variables (groups of cells) and their values (constants) are in the asymmetric roles. Another example of "asymmetry": manipulations over the structured objects of programs (arrays, records, etc.) are

not so transparent as over the primary ones. To describe the operational semantics for **ERA**, we use another approach. All objects of a program are considered to be "identical" in the following meaning.

Let $\mathcal{CV}$ be a set of 0–ary symbols representing variables and constants. The last ones may be of the following kinds: scalars, compositions over scalars (i.e., constant arrays, records, etc.), names of record fields, and *indefiniteness*. Let $\mathcal{FP}$ be a set of n–ary (*functional*) symbols which represent primitive operations of programming languages: arithmetic, logic, type casting, and all the kinds of memory addressing, as well. Let $\mathcal{TRS}$ be a set of regular tree terms over $\mathcal{CV}$ and $\mathcal{FP}$, hereinafter referred to as *program terms*. They represent expressions computed during execution of a program. So, as a state of program memory we take a reflexive, symmetrical, and transitive relationship (i.e., *the equivalence relationship*) over $\mathcal{TRS}$. The relationship defines some set of term equalities which we use to describe the operational semantics and call a *computation state*. Note that this set is always infinite because there exist infinitely consistent equalities of terms in contrast to the mapping *cells→values* where this is always possible for divergent programs.

Let us consider the following example

```
...
VAR x,i,j: INTEGER; a: ARRAY [1..3] OF INTEGER;
...
a[1]:=1; a[2]:=2; a[3]:=3;
i:=3; j:=i-1;
IF ODD(x) THEN
   i:=i MOD j; j:=1
ELSE
   j:=a[i]; a[i]:=a[1]; a[1]:=j
END
...
```

Example 1

Suppose that this piece of code is executed at least twice for the different evenness of the variable x. **Table 1** gives us the static semantics for five control points. We present a minimum subset of term equalities concerning the essential part of the behavior of the piece. We shall use the property π to illustrate our further reasoning.

It can be formally described in the following way. Let $\mathcal{EQS}$ be a set of all equalities of the terms from $\mathcal{TRS}$, i.e., $\mathcal{EQS} = \{t_1 = t_2 \mid t_1, t_2 \in \mathcal{TRS}\}$. A set $S \in \wp(\mathcal{EQS})$ is a computation state and it is interpreted as if for each equality $t_i = t_j \in S$ the expressions represented by t_i and t_j were computed on an execution trace and their values are equal. We take the set $\wp(\wp(\mathcal{EQS}))$ as a set for a concrete semantic domain describing *the static semantics* considered in **ERA**.

	THEN-brunch	ELSE-brunch
ENTRY	{a[1]=1,a[2]=j=2,a[3]=i=3, a=[1|2|3],ODD(x)=TRUE}	{a[1]=1,a[2]=j=2,a[3]=i=3, a=[1|2|3],ODD(x)=FALSE}
EXIT	{a[1]=i=j=1,a[2]=2,a[3]=3, a=[1|2|3],ODD(x)=TRUE}	{a[1]=i=j=3,a[2]=2,a[3]=1, a=[3|2|1],ODD(x)=FALSE}

EXIT OF IF-STATEMENT

$\pi =$ { {a[1]=i=j=1,a[2]=2,a[3]=3, a=[1|2|3],ODD(x)=TRUE}, {a[1]=i=j=3,a[2]=2,a[3]=1, a=[3|2|1],ODD(x)=FALSE} }

Table 1. The static semantics for **Example 1** (here the constant $\boxed{c_1}\boxed{c_2}\boxed{c_3}$ represents constant arrays)

The properties considered in **ERA** are presented by means of context–free grammars of a special type. We do not give their extended description and expect that it becomes apparent from the examples on **Figure 1** and further ones.

2.2. Abstract properties

It is an interesting peculiarity of **ERA** that abstract (i.e. approximate) properties have the same nature as the computation states of the operational semantics. Formally this approximation is defined as follows.

Given a concrete property $\pi \in \wp(\wp(\mathcal{EQS}))$ and an abstract property $\tilde{\pi} \in \mathcal{EQS}$, the abstraction function $\alpha : \wp(\wp(\mathcal{EQS})) \to \wp(\mathcal{EQS})$ and the concretization one $\gamma : \wp(\mathcal{EQS}) \to \wp(\wp(\mathcal{EQS}))$ are defined in the following way

$$\alpha(\pi) = \begin{cases} \mathcal{EQS}, & \text{if } \pi = \emptyset, \\ \bigcap_{S \in \pi} S & \text{otherwise} \end{cases} \quad \text{and} \quad \gamma(\tilde{\pi}) = \sqcup\{\ \pi \mid \alpha(\pi) \sqsupseteq' \tilde{\pi}\ \}.$$

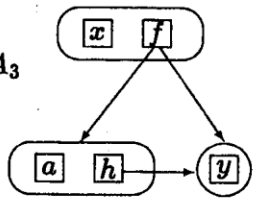
Equality relationships	Grammar representation	Computation net representation
$a = h(y)$	$S \to A_1 = A_1 \mid A_2 = A_2 \mid A_3 = A_3$	
$x = f(a, y)$	$A_1 \to x \mid f(A_2, A_3)$	
$x = f(h(y), y)$	$A_2 \to a \mid h(A_3)$	
$f(a, y) = f(h(y), y)$	$A_3 \to y$	

Figure 1. Semantic properties and their representations

Obviously, they are monotonic and give the best approximation and concretization of the corresponding properties. For the example of **Table 1**, the best approximation of the concrete property π is

$$\alpha(\pi) = \left\{ a[1] = i = j, a[2] = 2 \right\}. \qquad (*)$$

2.3. Operations over semantic properties

Now we discuss some basic operations over semantic properties used to define the semantic transformers of **ERA**(their description can be found in [7, 8]).

Operations over abstract computation states $\tilde{\pi}$ use certain common transformation of the sets of term equalities which consists in **removing** some subset S'. The following statement holds.

Lemma 1. *Removing any subset of term equalities preserves correctness of an approximation.*

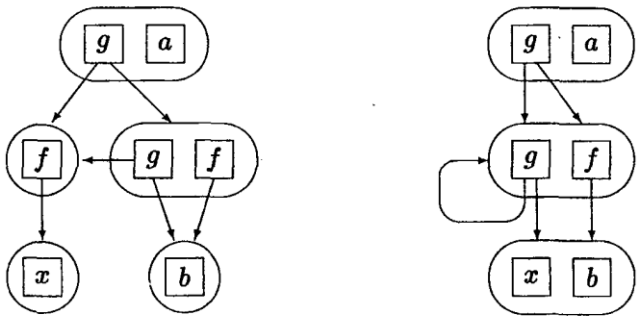
Proof. It easy to see that

$$\sqcup\{\ \pi \mid \alpha(\pi) \sqsupseteq' \tilde{\pi}\ \} = \gamma(\tilde{\pi}) \quad \sqsubseteq \quad \gamma(\tilde{\pi} \setminus S') = \sqcup\{\ \pi \mid \alpha(\pi) \sqsupseteq' (\tilde{\pi} \setminus S)\ \} = \sqcup\{\ \pi \mid \bigcap_{S \in \pi} (S \cup S') \sqsupseteq' \tilde{\pi}\ \},$$

which states that removing term equalities makes the approximation more rough but it does preserve its correctness. □

For $(*)$, for example, $\gamma(\alpha(\pi)) \sqsubseteq \gamma(\{a[1] = i\})$.

Unification of terms corresponds to a situation when, during execution of a program, it turns out that the values of computed expressions represented by the terms become equal. For example, an access term representing the *l*-value of an assignment defines the same value as an expression term



L $Int[\![x \equiv b]\!]L$

Figure 2. Unification of values of terms

representing the r-value after this statement. Also we can say that a value of a term representing the conditional expression of an **IF**-statement coincides with 0-ary terms representing the constants **TRUE** or **FALSE** when, respectively, **THEN**-brunch or **ELSE**-brunch is being executed. So, unification of terms, along with **semantic closure** considered below, provides a powerful facility for taking into account a real control flow in programs and makes **ERA** flow–sensitive. Let us briefly recall this transformation.

Unification of terms $Int[\![t_1 \equiv t_2]\!]L$.

1. If $t_1 = t_2 \in L$ then $Int[\![t_1 \equiv t_2]\!]L = L$.
2. Let $A_1 \overset{+}{\underset{G}{\Rightarrow}} t_1$ and $A_2 \overset{+}{\underset{G}{\Rightarrow}} t_2$. We replace the nonterminal A_2 by the nonterminal A_1 in all rules of $\mathcal{P}$. If rules with an identical right side $B_1 \to w, \ldots, B_k \to w$ appeared, then a certain nonterminal from the left side of a rule (for example B_1) must be taken and all nonterminals $B_2, \ldots, B_k$ in the grammar must be replaced by it.
3. Repeat step 2 until stabilization. If after that we have a state L' containing inconsistent term equalities[2], then the result is $\top'$, else it is a reduction of L'.

An example of unification is given in **Figure 2**.

Lemma 2. *Unification of values of terms is a correct transformation and the resulting state is unique.*

[2]There exists a wide spectrum of inconsistency conditions. The simplest of them is an equality of two different constants.

Proof. Let $\pi = \{ L_i \mid L_i \in \wp(\mathcal{EQS}) \}$ be a concrete semantic property which holds before unification of terms t_1 and t_2. If the values of t_1 and t_2 are equal in the concrete semantics $\forall L_i : t_1 = t_2 \in L_i$, then they are equal in the abstract semantics $t_1 = t_2 \in \alpha(\pi)$, too. If their values are not equal, then unification gives us the inconsistent computation state which obviously includes $Int[\![t_1 \equiv t_2]\!]L$ for all t_1 and t_2. So, this transformation is correct.

Unification can be done in finite steps because the size of grammar decreases at each step. Uniqueness of the resulting state is explained by the following observation. If we have two pairs of terms which are candidates for unification, then unification of one of them does not close a possibility of it for another, because we remove a duplication of the functional symbols only. In fact, after unification of a pair of terms we obtain a new state, including the source one, and thus other existing unification possibilities retain. So, the order of "merging" of term pairs is not important for the resulting state. □

We do not yet consider any interpretation of constants and functional symbols. We could continue developing **ERA** in the same way. So, we obtain a noninterpretational version of the analysis as in the program schemata theory. However, it is natural to use the semantics of primitive operations of the programming language in order to achieve better accuracy.

ERA provides us with wide possibilities of taking into account the properties of the language constructs, and, what is especially important, we can easy handle the complexity of these manipulations. In fact, inclusion of these properties corresponds to carrying out a finite part of completion of the computation states by consistent equalities. This completion is named a **semantic closure**. The "size" of this part can be handled both by the developer (hardly embedded into the analyzer) and by the user (tuned by options of interpretability).

Let us describe this transformation. Next we suppose that **a)** a rule is added to the grammar if it is absent there; **b)** $f(\ldots)$ denotes some term derived by the correspondent rule; **c)** if at some step of the algorithm the state $\top'$ occurs, then the algorithm stops.

Semantic closure (a basic version)

1. Let the rule $A \to f(A_1, \ldots, A_n)$ such that $A_1 \Rightarrow c_1, \ldots, A_n \Rightarrow c_1$ be in the grammar. If for the constant $c = f(c_1, \ldots, c_n)$ there is no nonterminal A' such that $A' \Rightarrow c$, then add the rule $A \to c$ into the grammar. Otherwise, $Int[\![c \equiv f(\ldots)]\!]L(\mathbf{G})$.
2. Let the rule $A \to f(B, B)$ be in the grammar and f be a functional symbol representing a primitive operation from the left column of **Table 2**. If there is no nonterminal A' such that $A' \Rightarrow c$ where c is from the

f	c	f	c
$-$	0	$\neq$	FALSE
$/^{\star}$	1	$<$	FALSE
div*	1	$\leq$	TRUE
mod*	0	$>$	FALSE
xor	FALSE	$\geq$	TRUE
$=$	TRUE		

* It is possible that arithmetical errors appear (see a remark before the algorithm).

Table 2. Interpretation of primitive operations

f	c	t
$+$	0	a_j
$-^{\star}$	0	a_j
$*$	0	0
$*$	1	a_j
$/^{\star}$	1	a_j
div*	1	a_j
mod*	1	0
and	FALSE	FALSE
or	TRUE	TRUE
and	TRUE	a_j
or	FALSE	a_j

* For these operations it is necessary that $B_2 \Rightarrow c$.

Table 3. Interpretation of primitive operations (continuation)

right column, then add the rule $A \rightarrow c$ into the grammar. Otherwise, $Int[\![c \equiv f(\ldots)]\!]L(\mathbf{G})$.

3. Let the rule $A \rightarrow f(B_1, B_2)$ be in the grammar and f be a functional symbol representing a primitive operation from the first column of **Table 3**. If there exists a nonterminal B_i (i=1,2) such that $B_i \Rightarrow c$ where c is from the second column, then $Int[\![t \equiv f(\ldots)]\!]L(\mathbf{G})$ should be computed, where t is a term pointed out in the third column of **Table 3**. Here a_j denotes one of the terms such that $B_j \Rightarrow c$ $(j = 1,2 \wedge j \neq i)$.
4. Apply steps 1–3 until stabilization.

As mentioned above, some arithmetical errors (such as division by zero,

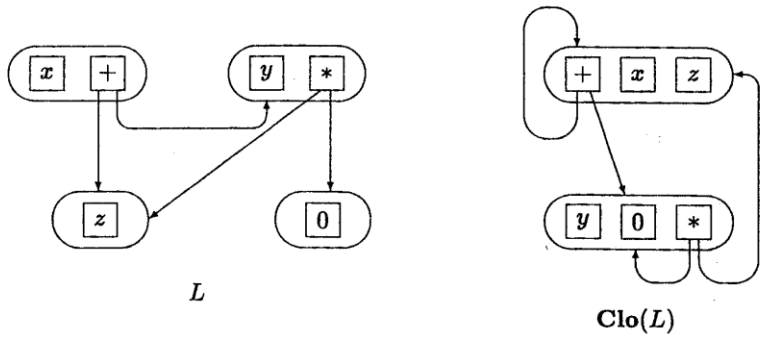


Figure 3. Semantic closure

out of type range, etc.) can appear during the constant closure. In this case the analyzer tells us about the error and sets the current computation state to $\top'$. Note that for the languages, where the incomplete boolean evaluation is admissible, the semantic closure over boolean expressions should be carefully designed. The following example demonstrates a probable problem: **(p≠NIL) AND (p^.f=a)**.

An example of the semantic closure is presented in **Figure 3**. Turning back to the unification example in **Figure 2**, we can consider the following interpretation of constants and functional symbols: g is the exclusive disjunction, f is the negation, a is the constant **TRUE** and b is the constant **FALSE**. It is easy to see that, in the case of the interpretational **ERA**, application of the semantic closure gives us $Int[\![x \equiv b]\!]L = \top'$.

In our analyzer we have implemented the interpretational version of **ERA** which uses the operation of the semantic closure $\mathbf{Clo}(L)$. Under this approach, the definitions of the basic transformations mentioned above are changed to the following ($Int[\![t]\!]L$ is the "evaluation of a term" operation; see [7, 8]):

$$Int[\![t]\!]'L = \mathbf{Clo}(Int[\![t]\!]L),$$
$$Int[\![t_1 \equiv t_2]\!]'L = \mathbf{Clo}(Int[\![t_1 \equiv t_2]\!]L).$$

Usually we omit this "interpretability" prime.

3. A widening operator and divergence of analysis

Our abstract semantic domain does not satisfy the chain condition and therefore it requires a *widening operator* whose construction is given in [7].

Infinite chains stem from occurrence of cyclic derivations in grammars if we do not restrict their form. The subsemilattice of the finite languages[3] generated by acyclic grammars of the semilattice $\wp(\mathcal{EQS})(\sqsupseteq', \mathcal{EQS}, \sqcap')$ satisfies the ascendant chain condition, but such languages are not expressive enough. Our solution is as follows. The grammars are not originally restricted but if in the course of abstract interpretation the grammar's size becomes greater than some parameter, then the "harmful" cycles must be destructed. To this end we remove the grammar rules which participate in cyclic derivations. Correctness of this approximation of intersection follows from **Lemma 1**. The lengths of such chains are bounded by $|\mathbf{G}_0|$, where $\mathbf{G}_0$ are the first acyclic grammars in the chains under consideration.

Detecting these rules is not simpler than the "minimum-feedback-arc/vertex-set" problem (**MFAS** or **MFVS**) if we consider the grammars as directed graphs. These sets are the smallest sets of arcs or vertices, respectively, whose removal makes a graph acyclic. In [7] it was proposed to consider feedback arc sets. But now we suppose that the "feedback vertices" choice is more natural for our purposes. In the general case this problem is $\mathcal{NP}$–hard, but there are approximate algorithms that solve this problem in a polynomial [18] or even linear [14] time. Consideration of the weighted feedback problems makes possible to distinguish grammar rules with respect to their worth for accuracy of the analysis algorithm. However, the perspectives of this are not clear now.

The widening operator for the analysis of equality relationships is defined in the following way. A transformation of a grammar graph which consists in detecting some **FVS** and removing all feedback vertices is said to be an **FVS**–transformation (an example is shown in **Figure 4**). Let $L\backslash_{\mathbf{FVS}}$ be a language obtained from L by **FVS**–transformation applied to the grammar which generates L. We define

$$L(\mathbf{G}_1)\widetilde{\nabla}L(\mathbf{G}_2) = \begin{cases} L(\mathbf{G}_1)\backslash_{\mathbf{FVS}} \sqcap' L(\mathbf{G}_2) & \text{if } |\mathbf{G}_2| > |\mathbf{G}_1| > d, \\ L(\mathbf{G}_1) \sqcap' L(\mathbf{G}_2)\backslash_{\mathbf{FVS}} & \text{if } |\mathbf{G}_1| > |\mathbf{G}_2| > d, \\ L(\mathbf{G}_1) \sqcap' L(\mathbf{G}_2) & \text{otherwise,} \end{cases}$$

where d is a user-defined parameter. It is reasonable to choose this parameter, depending on the number of variables of the analyzed program, as a linear function with a small factor of proportionality. Note that in this case the lengths of appearing chains linearly depend on the number of variables.

Is the widening operator, being rather complex, really needed for the analysis of equality relationships? Do programs exist which, being analyzed, generate infinite chains of semantic properties? It should be mentioned that

[3]Note that the sets of term equalities of a special kind corresponding to these languages were used by V. Sabelfeld to develop effective algorithms of recognizing equivalences for some classes of program schemata.

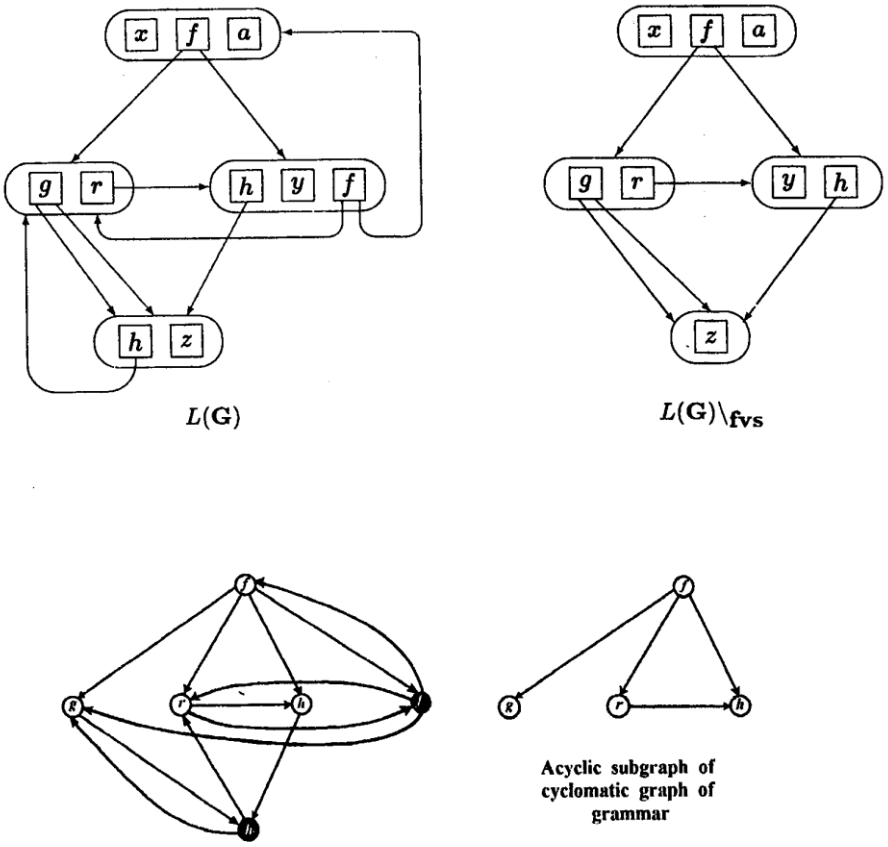


Figure 4. FVS-transformation

constructing such program examples has been a problem for a long time. In [7] we stated our belief that their rise seems hardly probable. These attempts failed, because they were concentrated on constructing an example with completely noninterpretable functional symbols (i.e., in the frame of the "pure" theory of program schemata).

As already noted, we can widely variate the interpretability of the analysis algorithm. In order to construct the required example, it will suffice to consider a standard interpretation of the Boolean type and the comparison *equ*, namely:

if the computation state L knows an equality $equ(t_1, t_2) = TRUE$ then $\gamma(L) \sqsubseteq \gamma(Int[\![t_1 \equiv t_2]\!]L)$.

```
...
x:=f(y);
IF  f(x)=f(y)  THEN
  WHILE  y=f(g(y))  DO
    y:=g(y)
  END
END
...
```

program scheme

```
...
x:=sign(y);
IF  sign(x)=sign(y)  THEN
  WHILE  y=sign(abs(y))  DO
    y:=abs(y)
  END
END
...
```

"real-world" program

The properties computed at the body's entry belong to an infinite decreasing chain of the abstract semantic domain. In **Figure 5** a state L_e describes the properties valid before the cycle execution; states L_1 and L_2 describe the properties at the entry of the cycle body for the first and second iteration, respectively. It is easy to see that $L_1 \sqcap' L_2$ coincides with L_2 except the equality relationships containing terms generated by g^*. So, every time we obtain the next state, the functional element g^* is absent and the subnet placed in the dashed box repeats time and again.

To obtain a "real-world" program from this program scheme, we can interpret the functional symbols in the manner like this: $f \equiv sign$ and $g \equiv abs$. We would like to highlight the following interesting point. Execution of this piece of code (i.e., its behavior determined with the standard semantics) diverges only for two values of **y**: 0 and 1. At the same time the analysis algorithm (i.e. execution of the piece of code under a nonstandard semantics) is always divergent on condition that the widening operator is not used and the assumptions on the interpretation mentioned above hold.

Is this program actually a real-world one? The reader can decide this by himself but we note the following. On the one hand, the interpretability of the analysis algorithm can be variated in wide ranges and, on the other hand, we are not able to formally prove impossibility of such a behavior of

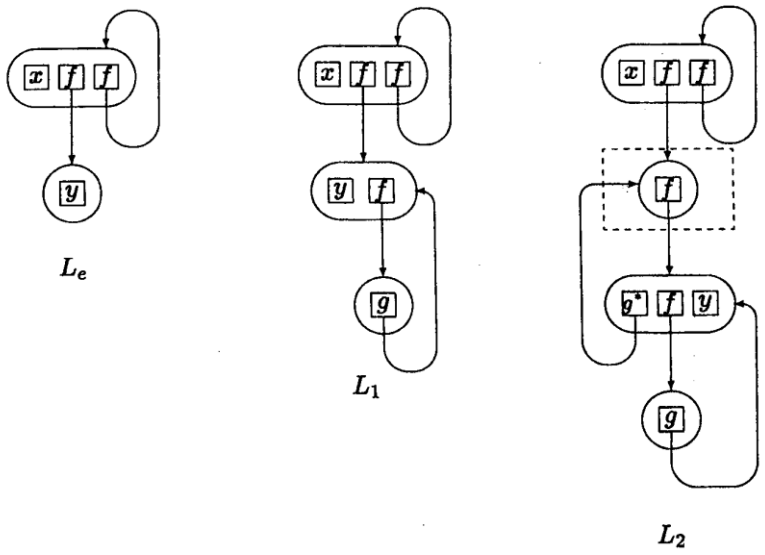


Figure 5. The case of divergence of analysis

the analyzer under the considered interpretation. So, we can choose either a lean analysis using acyclic grammars only or another one using arbitrary grammars and a widening operator.

4. Once again on complexity of analysis

In [7] we pointed out the following upper bound on time for the algorithm of **ERA**: $O(n^2 G_{max}^2)$, where n is the size of the program, and G_{max} is the maximum of the sizes of grammars appearing in the course of the analysis. Now we give an extended motivation of this bound.

It can be deduced with the help of **Theorem 6** in [1]. The theorem states that for the recursive strategy of the chaotic iteration (see all definitions given in [1]) the maximum complexity is

$$h \cdot \sum_{c \in C} \delta(c) \qquad \left(\leq h \cdot |C| \cdot |W| \right), \qquad (**)$$

where h is the maximum length of the increasing chains built by the widening operator, C is the set of control program points, $\delta(c)$ is the depth of the control point c in hierarchy of nested strongly connected components of the control flow graph containing c, and W is the set of vertices where the widening operator is applied during the analysis.

We recall that Modula-programs are well-structured and we also can suppose that the maximum depth of nested loops does not depend on the program size and is bounded by some constant. Taking into account these properties and the construction of our widening operator (namely, our choice of the parameter d) and using $(**)$, we conclude that the number of algorithm steps does not exceed $O(n^2)$. Since time complexity of all operations used in the analysis is estimated by G_{max}^2[4], we obtain our upper bound.

However, experimental results show that an approximation of a fixed point for the heads of cycle bodies is usually attained after at most two iterations and the time complexity of the analysis does not normally exceed $O(nG_{max})$. Also, the user can turn off the infinite chain control. In this case, he (consciously) admits some chance that the analysis diverges but we believe that this chance is not too big.

It is easy to see that the space complexity of the equality relationship analysis is $O(nG_{max})$. Unfortunately, this bound does not provide useful information about the actual space requirements for the analysis algorithm. We estimate them as 1.5–2.0 Mb RAM per 1000 program lines and emphasize that it comes into particular importance for virtual memory systems. Random management of swapping makes impracticable analyzing large programs in small RAM.

5. Experimental results

A family of semantic properties which can be detected in the automatic mode of the analyzer were presented in [7, 10]. Apart from the automatic mode, we provide an interactive mode to visualize the results of our analysis in some hypertext system[5]. There are two facilities here: visualization of properties detected in the automatic mode and the user-driven visualization of properties. The last one can be elucidated as follows.

The experiments show that not all program properties of interest can be automatically extracted out of the computed invariants. It is not judicious to consider many particular cases and to hardly embed them into the system (for the purpose of automatic making of a corresponding solution). Instead, the user has a possibility to specify its request by a friendly interface. He can choose a program point and an expression and obtain those and only those equality relationships, valid at this point, where this expression occurs. The following strategies of selection of the subsets of properties are provided: **a)** all terms occurring as subterms of the term under consideration; **b)** all

[4]For the **FVS**-transformation we can achieve the same bound.

[5]HyperCode [3]: a system based on databases for visualization of properties of syntactic structures (being developed in the Laboratory of Mixed Computation of Institute of Informatics Systems).

terms using the term under consideration as a subterm; **c)** the union of sets determined by the previous cases **a)** and **b)**. These problems are obviously reduced to computing a transitive closure of the corresponding digraph with respect to some vertex and orientation of arcs. Note that an automatic solver can be based on the term–rewriting technique [13].

An example of program is presented below. The properties detected by the analyzer are indicated in comments.

```
MODULE Example;
VAR x,y,z: INTEGER;
PROCEDURE P(a,b: INTEGER): INTEGER;
BEGIN                    (*parameters are always equal*)
  RETURN a+b             (*expression can be simplified: 2*a*)
END P;
BEGIN
  Read(x);
  WHILE x≤0 DO
   Read(x);
   INC(x);
   z := x+z;             (*variable z might be uninitialized*)
   y := x+1;
   IF x=0 THEN
     z := y;             (*r-value can be simplified: z:=1*)
   ELSE
     z := x+1;           (*r-value can be simplified: z:=y*)
     x := y;
   END;
   Write( P(y,z) )       (*call can be transformed: Write(2*y)*)
  END ;
  x := z MOD (y - z);    (*arithmetical error*)
  Write(x)               (*inaccessible point*)
END Example.
```

On the basis of the analysis, this program can be transformed into the following form:

```
MODULE Example;
VAR x:INTEGER;
BEGIN
  Read(x);
  WHILE x≤0 DO
    Read(x);  INC(x,2);  Write( 2*x )
  END;
  ERROR_EXCEPTION
END Example.
```

program	length (lines)			size (bytes)		
	M2Mix	ASP	improv.	*M2Mix*	ASP	improv.
KMP	167	133	20.35%	2996	2205	26.4%
Lambert	361	326	9.7%	6036	2564	57.5%
Automation	37	35	5.4%	969	926	4.5%
$\mathbf{Int}_{Fib}$	87	77	11.5%	1647	1432	13.05%
Ackerman	64	62	3.1%	1384	1322	4.5%
	average		10.01%	average		21.19%

Table 4. Comparison between ASP and *M2Mix*

In **Table 4** we present some results of optimization based on our analysis (ASP) of residual programs generated by *M2Mix* specializer [2, 12]. The following programs were investigated:

- **KMP** — the "naïve" matching algorithm specialized with respect to some pattern; the residual program is comparable to Knuth, Morris, and Pratt's algorithm in efficiency (see also **Appendix**).
- **Lambert** — a program drawing the Lambert's figure and specialized with respect to the number of points.
- **Automation** — an interpreter of a deterministic finite-state automation specialized with respect to some language.
- $\mathbf{Int}_{Fib}$ — an interpreter of MixLan [12] specialized with respect to a program computing Fibonacci's numbers.
- **Ackerman** — a program computing some values of Ackerman's function and specialized with respect to the first argument.

Let us briefly comment the obtained results. Reducing the length of a program can be considered as reducing the number of operators and declarations. In these examples the optimizing effect was typically attained by the removal of redundant assignments and unused variables and the reduction of the operator strength. The only exception is the **KMP** program characterized by a high degree of polyvariance and the active use of array references. Here some if-statements with constant conditions and redundant range checks were eliminated. Note that the last optimizing transformation is very important for Modula-like languages. Such a notable optimizing effect for the **Lambert** program is explained by a deep reduction of the power of floating-point operations. Since the **Automation** and **Ackerman** programs are quite small, their optimization gives conservative results. However, they would be better for the **Ackerman** program if the implementation

of **ERA** were context-sensitive. A substantial speed-up of these optimized programs was not obtained and this is not surprising since the great bulk of specializers take it as a criterion of optimality.

These experiments show that an average reduction of size of residual programs is 20–25%. Because the case of the **KMP** program seems to be the most realistic, we suppose that such an improvement can be achieved in practice for real-world programs and it will be increased for large residual programs with a high degree of polyvariance and active use of arrays and float-point arithmetics. It is the author's opinion that the analysis of automatically generated programs which can be used for their optimization is the most perspective direction of its application, especially in the context-sensitive implementation of **ERA**.

References

[1] F. Bourdoncle, *Efficient chaotic iteration strategies with widenings*, Proc. of the International Conference on Formal Methods in Programming and Their Applications (Berlin a.o.), Lecture Notes in Computer Science, **735**, 1993, 129–141.

[2] M. Bulyonkov and D. Kochetov, *Practical aspects of specialization of Algol-like programs*, Proc. of the International Seminar on Partial Evaluation (Berlin a.o.), Lecture Notes in Computer Science, **1110**, 1996, 17–32.

[3] M. Bulyonkov and D. Kochetov, *Visualization of program properties*, Research Report, Institute of Informatics Systems, Novosibirsk, No. 51, 1998.

[4] P. Cousot and R. Cousot, *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, Rec. of the 4^{th} ACM Symposium on Principles of Programming Languages (New-York), ACM Press, 1977, 238–252.

[5] P. Cousot and R. Cousot, *Abstract interpretation and application to logic program*, Journal of Logic Programming, **13**, No. 2–3, 1992, 63.

[6] P. Cousot and R. Cousot, *Abstract interpretation frameworks*, Journal of Logic and Computation, **2**, No. 4, 1992, 511–547.

[7] P.G. Emelianov, *Analysis of the equality relations for the program terms*, Proc. of the Third Static Analysis Symposium (Berlin a.o.), Lecture Notes in Computer Science, **1145**, 1996, 174–188.

[8] P.G. Emelianov, *Methods and tools for the static analysis of semantic properties of programs*, Ph.D. Thesis, Novosibirsk State University, Novosibirsk, Russia, 1997.

[9] P.G. Emelianov and D.E. Baburin, *Semantic analyzer of Modula-programs*, Proc. of the Fourth International Static Analysis Symposium (Berlin a.o.), Lecture Notes in Computer Science, **1302**, 1997, 361–363.

[10] P.G. Emelianov and V.K. Sabelfeld, *Analyzer of semantic properties of Modula-programs*, Software intellectualization and quality, Institute of Informatics Systems, Novosibirsk, 1994, 100–107.

[11] N.D. Jones and S.S. Muchnick (eds.), *Program flow analysis: Theory and applications*, Prentice-Hall, Englewood Cliffs, USA, 1981.

[12] D.V. Kochetov, *Effective specialization of Algol–like programs*, Ph.D. Thesis, Institute of Informatics Systems, Novosibirsk, Russia, 1995.

[13] G. Nelson and D.C. Oppen, *Fast decision procedures based on congruence closure*, Journal of the ACM, **27**, No. 2, 1980, 356–364.

[14] B.K. Rosen, *Robust linear algorithms for cutsets*, Journal Algorithms, **3**, 1982, 205–217.

[15] B.G. Ryder, *Practical compile-time analysis*, Proc. of the Fourth International Static Analysis Symposium (Berlin a.o.), Lecture Notes in Computer Science, **1302**, 1997, 406–412.

[16] V.K. Sabelfeld, *Polynomial upper bound for the complexity of the logic-termal equivalence decision*, Doklady Mathematics, **249**, No. 4, 1979, 793–796.

[17] V.K. Sabelfeld, *The logic–termal equivalence is polynomial–time decidable*, Information Processing Letters, **10**, No. 2, 1980, 102–112.

[18] E. Speckenmeyer, *On feedback problems in digraphs*, Proceedings of the 15^{th} International Workshop on Graph–Theoretic Concepts in Computer Science (Berlin a.o.), Lecture Notes in Computer Science, **411**, 1990, 218–231.

Appendix. Analysis of KMP

The appendix presents the results of the application of **ERA** to the **KMP** program generated by the specializer *M2Mix*. This program is a specialization of a program which is an implementation of naïve pattern matching **Match(p,str)** with respect to the pattern **p="ababb"**.

Let us consider the results of our analysis of the program **Match("ababb", str)** written as comments at program points where they hold. Only the most interesting of them are given. We can conclude that

- The target string necessarily ends with "#" and the variable **ls** is equal to the string length (line 10).
- Every time when some element of **str** (lines 20, 31, 42, 53, 58, 67, 72,77,85, 107,112,128,139,150,161) is used in the second **LOOP**, the value of its index expression does not exceed the value of the variable **ls**. We can say the same about the value of a variable before the increment statements **INC(s)** (lines 23, 34, 45, 62, 80, 88, 93, 97, 102, 115, 119, 131, 142, 153, 164). Therefore, it suffices to check that a value of **ls** is not beyond the ranges determined by the type **_TYPE354a04** during input of the target string (line 8). After that all range checks can be eliminated in the second cycle.
- The assignment **_cfg_counter:=0** is redundant (line 24).
- **str[s+2]='a'** and **str[s]='a'** are always false (lines 77 and 85, respectively). So, computing them is redundant and the code of THEN-branches is dead.
- The conditions at lines 63, 68, 73, 81, 103, 108 are false, too. However it is not automatically detected by the analyzer in our implementation and this is the case when the user's participation is required. Other implementations of **ERA** can handle such things automatically.

Using this semantic information, we can build a new program functionally equivalent to **Match("ababb", str)**. Their characteristics are presented in **Table 4**. To compile these examples, we used XDS Modula/Oberon compiler (v.2.30). Note that it detected unique redundancy in the source program associated with the constant condition in line 81.

```
MODULE KMP;
   FROM FIO IMPORT File,Open,ReadChar,WriteInt,stdout;
   VAR _cfg_counter : CARDINAL;
   TYPE _TYPE354a04 = [0..20];
   TYPE _TYPE355004 = ARRAY _TYPE354a04 OF CHAR;
   VAR str : _TYPE355004;
   VAR str_file : File;
   VAR ls,s : _TYPE354a04;
BEGIN
1:  str_file := Open("target.dat");
2:  ls := 0;
3:  LOOP
4:     str[ls] := ReadChar(str_file);
5:     IF (str[ls]='#') THEN                    (*str[ls] ='#'*)
6:        EXIT
7:     ELSE                                     (*str[ls] ≠'#'*)
8:        INC(ls)
9:     END
10: END;                                        (*str[ls] ='#'*)
11: s := 0;
12: _cfg_counter := 0;                          (*s = _cfg_counter = 0*)
13: LOOP
14:    CASE _cfg_counter OF
15:    | 0 :                                    (*_cfg_counter = 0*)
16:       IF ((s+0)≥ls) THEN                    (*s ≥ ls*)
17:          WriteInt(stdout,(-1),0);
18:          EXIT
19:       END;                                  (*0 < ls, s < ls*)
20:       IF (str[(s+0)]='a') THEN              (*str[s] ='a', 0 < ls, s < ls*)
21:          _cfg_counter := 1
22:       ELSE                                  (*0 < ls, s < ls, str[0] ≠'a'*)
23:          INC(s);                            (*_cfg_counter = 0*)
24:          _cfg_counter := 0
25:       END                                   (*0 < ls, s < ls*)
```

```
26: | 1 :                                  (*_cfg_counter = 1*)
27:     IF ((s+1)≥ls) THEN                 (*s + 1 ≥ ls*)
28:         WriteInt(stdout,(-1),0);
29:         EXIT
30:     END;                               (*s + 1 < ls*)
31:     IF (str[(s+1)]='b') THEN           (*str[s + 1] ='b'*)
32:         _cfg_counter := 2
33:     ELSE                               (*str[s + 1] ≠'b'*)
34:         INC(s);                        (*str[s] ≠'b', s < ls*)
35:         _cfg_counter := 0
36:     END
37: | 2 :                                  (*_cfg_counter = 2*)
38:     IF ((s+2)≥ls) THEN                 (*s + 2 ≥ ls*)
39:         WriteInt(stdout,(-1),0);
40:         EXIT
41:     END;                               (*s + 2 < ls*)
42:     IF (str[(s+2)]='a') THEN           (*str[s + 2] ='a'*)
43:         _cfg_counter := 3
44:     ELSE                               (*str[s + 2] ≠'a'*)
45:         INC(s);                        (*str[s + 1] ≠'a', s + 1 < ls*)
46:         _cfg_counter := 4
47:     END
48: | 3 :                                  (*_cfg_counter = 3*)
49:     IF ((s+3)≥ls) THEN                 (*s + 3 ≥ ls*)
50:         WriteInt(stdout,(-1),0);
51:         EXIT
52:     END;                               (*s + 3 < ls*)
53:     IF (str[(s+3)]='b') THEN           (*str[s + 3] ='b'*)
54:         IF ((s+4)≥ls) THEN             (*s + 4 ≥ ls, s + 3 < ls*)
55:             WriteInt(stdout,(-1),0);
56:             EXIT
57:         END;                           (*s + 4 < ls*)
58:         IF (str[(s+4)]='b') THEN       (*str[s + 3] = str[s + 4] ='b'*)
59:             WriteInt(stdout,s,0);
60:             EXIT
61:         ELSE                           (*str[s+3] ='b', str[s+4] ≠'b', s+4 < ls*)
62:             INC(s);                    (*str[s+2] ='b', str[s+3] ≠'b', s+3 < ls*)
63:             IF ((s+0)≥ls) THEN         (*s ≥ ls, s+3 < ls*)
64:                 WriteInt(stdout,(-1),0);
65:                 EXIT
66:             END;                       (*str[s+2] ='b', str[s+3] ≠'b', s+3 < ls*)
67:             IF (str[(s+0)]='a') THEN   (*str[s] ='a', str[s+2] ='b', str[s+3] ≠'b',
                                               s+3 < ls*)
68:                 IF ((s+1)≥ls) THEN     (*s+1 ≥ ls, s+3 < ls*)
69:                     WriteInt(stdout,(-1),0);
70:                     EXIT
71:                 END;                   (*str[s] ='a', str[s+2] ='b', str[s+3] ≠'b',
                                               s+3 < ls*)
72:                 IF (str[(s+1)]='b') THEN (*str[s] ='a', str[s+1] ='b', str[s+2] ='b',
                                               str[s+3] ≠'b', s+3 < ls*)
73:                     IF ((s+2)≥ls) THEN (*s+2 ≥ ls, s+3 < ls*)
```

```
74:             WriteInt(stdout,(-1),0);
75:                EXIT
76:             END;                          (*str[s]='a', str[s+1]='b', str[s+2]='b',
                                                 str[s+3] ≠'b', s+3 < ls*)
77:             IF (str[(s+2)]='a') THEN      (*inaccesible point*)
78:                _cfg_counter := 3
79:             ELSE                          (*str[s]='a', str[s+1]='b', str[s+2]='b',
                                                 str[s+3] ≠'b', s+3 < ls*)
80:                INC(s);                    (*str[s-1]='a', str[s]='b', str[s+1]='b',
                                                 str[s+2] ≠'b', s+2 < ls*)
81:                IF ((s+0)≥ls) THEN         (*s ≥ ls, s+2 < ls*)
82:                   WriteInt(stdout,(-1),0);
83:                   EXIT
84:                END;                       (*str[s-1]='a', str[s]='b', str[s+1]='b',
                                                 str[s+2] ≠'b', s+2 < ls*)
85:                IF (str[(s+0)]='a') THEN(*inaccessible point*)
86:                   _cfg_counter:=14
87:                ELSE                       (*str[s-1]='a', str[s]='b', str[s+1]='b',
                                                 str[s+2] ≠'b', s+2 < ls*)
88:                   INC(s);                 (*str[s-2]='a', str[s-1]='b', str[s]='b',
                                                 str[s+1] ≠'b', s+1 < ls*)
89:                   _cfg_counter:=4
90:                END
91:             END                           (*str[s-2]='a', str[s-1]='b', str[s]='b',
                                                 str[s+1] ≠'b', s+1 < ls*)
92:          ELSE                             (*str[s]='a', str[s+1] ≠'b', str[s+2]='b',
                                                 str[s+3] ≠'b', s+3 < ls*)
93:             INC(s);                       (*str[s-1]='a', str[s] ≠'b',
                                                 str[s+1]='b', str[s+2] ≠'b', s+2 < ls*)
94:             _cfg_counter := 12
95:          END
96:       ELSE                                (*str[s]='a', str[s+2]='b', str[s+3] ≠'b',
                                                 s+3 < ls*)
97:          INC(s);                          (*str[s-1]='a', str[s+1]='b',
                                                 str[s+2] ≠'b', s+2 < ls*)
98:          _cfg_counter := 12
99:       END
100:   END
101: ELSE                                     (*str[s+3] ≠'b', s+3 < ls*)
102:    INC(s);                               (*str[s+2] ≠'b', s+2 < ls*)
103:    IF ((s+0)≥ls) THEN                    (*s ≥ ls, s+2 < ls*)
104:       WriteInt(stdout,(-1),0);
105:       EXIT
106:    END;                                  (*str[s+2] ≠'b', s+2 < ls*)
107:    IF (str[(s+0)]='a') THEN              (*str[s]='a', str[s+2] ≠'b', s+2 < ls*)
108:       IF ((s+1)≥ls) THEN                 (*s+1 ≥ ls, s+2 < ls*)
109:          WriteInt(stdout,(-1),0);
110:          EXIT
111:       END;                               (*str[s]='a', str[s+2] ≠'b', s+2 < ls*)
```

```
112:            IF (str[(s+1)]='b') THEN    (*str[s] ='a', str[s+1] ='b', str[s+2] ≠'b',
                                              s + 2 < ls*)
113:                _cfg_counter := 2
114:            ELSE                        (*str[s] ='a', str[s + 2] ≠'b', s + 2 < ls*)
115:                INC(s);                 (*str[s−1] ='a', str[s+1] ≠'b', s+1< ls*)
116:                _cfg_counter := 10
117:            END
118:        ELSE                            (*str[s + 2] ≠'b', s +2 < ls*)
119:            INC(s);                     (*str[s + 1] ≠'b', s + 1 < ls*)
120:            _cfg_counter := 10
121:        END
122:    END
123: | 4 :                                  (*_cfg_counter = 4*)
124:    IF ((s+0)≥ls) THEN                  (*s ≥ ls*)
125:        WriteInt(stdout,(-1),0);
126:        EXIT
127:    END;                                (*s < ls*)
128:    IF (str[(s+0)]='a') THEN            (*str[s] ='a', s < ls*)
129:        _cfg_counter := 1
130:    ELSE                                (*str[s] ≠'a', s < ls*)
131:        INC(s);                         (*str[s − 1] ≠'a', s ≤ ls*)
132:        _cfg_counter := 0
133:    END
134: | 10:                                 (*_cfg_counter = 10*)
135:    IF ((s+0)≥ls) THEN                  (*s ≥ ls*)
136:        WriteInt(stdout,(-1),0);
137:        EXIT
138:    END;                                (*s < ls*)
139:    IF (str[(s+0)]='a') THEN            (*str[s] ='a', s < ls*)
140:        _cfg_counter := 1
141:    ELSE                                (*str[s] ≠'a', s < ls*)
142:        INC(s);                         (*str[s − 1] ≠'a', s ≤ ls*)
143:        _cfg_counter := 0
144:    END
145: | 12:                                 (*_cfg_counter = 12*)
146:    IF ((s+0)≥ls) THEN                  (*s ≥ ls*)
147:        WriteInt(stdout,(-1),0);
148:        EXIT
149:    END;                                (*s < ls*)
150:    IF (str[(s+0)]='a') THEN            (*str[s] ='a', s < ls*)
151:        _cfg_counter := 14
152:    ELSE                                (*str[s] ≠'a', s < ls*)
153:        INC(s);                         (*str[s − 1] ≠'a', s ≤ ls*)
154:        _cfg_counter := 4
155:    END
156: | 14:                                 (*_cfg_counter = 14*)
157:    IF ((s+1)≥ls) THEN                  (*s + 1 ≥ ls*)
158:        WriteInt(stdout,(-1),0);
159:        EXIT
160:    END;                                (*s + 1 < ls*)
```

```
161:        IF (str[(s+1)]='b') THEN    (*str[s + 1] ='b', s + 1 < ls*)
162:            _cfg_counter := 2
163:        ELSE                        (*str[s + 1] ≠'b', s + 1 < ls*)
164:            INC(s);                 (*str[s] ≠'b', s < ls*)
165:            _cfg_counter := 4
166:        END
167:    END
    END
END KMP.
```