UNIFICATION ALGORITHMS BASED ON PROGRAM TERMS

If the terms s and t represent a formal record of programs within the framework of some predetermined program machine, then we are talking not simply about unifying or comparing two programs in order to find common fragments in them, but about an attempt to find some third term p containing sufficient number of variables to find such substitutionsσAndτ, for which the following relations hold:

$$\sigma(p) = s, \tau(p) = t.$$

The difficulty is that for most algorithmic languages there is a trivial solution to this problem, for example, as follows. Let's create the following program fragment:

```
if ( x ) s else t;
```

In this caseσ= { x <- 1 }, andτ= { x <- 0 }, and then, by our definition, a program p is obtained that satisfies the requirement of comparison with the programs represented by the terms s and t. Such a trivial solution has the right to exist in the absence of any common fragments in the terms s and t, but in other cases it is a method of extensive development of software systems in order to acquire new functionality for the program. It should, however, be noted that this way of developing program components is currently common in groups of program developers using low-automated design with “menu” methods.

A more constructive requirement should be considered the requirement to simplify the resulting partial structural unification of the program p relative to the partially unified programs s and t. In the simplest way, the complexity G of a program p can be represented by a natural number n, corresponding to the total number of program statements: G(p) = n. Mathematically, strictly for any term s the complexity of G(s) is equal to the number of function symbols of which it consists. Thus, the requirement to reduce program complexity can be described by the following inequality.

$$G(p) < G(s) + G(t) .$$

The fulfillment of this predicate will guarantee the feasibility of using the result of the partial unification for programs s and t. For the trivial solution given earlier, this relation does not hold, because G(p) = G(s) + G(t) + G (fif) = G(s) + G(t) + 1.

The feasibility of partial unification in the case of execution G(p) < G(s) + G(t) is determined by the fact that when finding unified fragments of program terms, you can use modular, instrumental or frame approaches to obtain the resulting program that has the functionality of partially unified programs , which has a volume smaller than a trivially combined program.

Before presenting possible partial unification algorithms obtained from known unification algorithms, we note those differences that do not allow direct use of existing algorithms:

1) with partial unification of two program terms, it is necessary to find the most general term that is completely unified with both program terms, as well as two unifiers corresponding to each of the terms, while with complete unification it is necessary to find a single unifier for the two initial terms;

2) partial unification should provide a condition for reducing the total complexity of the final term;

3) the need to find, when completely non-unified fragments of a program are discovered, terms of maximum complexity to replace them with a variable;

4) not only individual variables, constants, terms, but also simply functional symbols can be replaced by variables, as is customary in attribute structures;

5) the found substitutions significantly influence the structure of the final term of the program.

It should be noted that the ideal option for comparing program terms is complete unification in its classical sense, which allows you to effectively form the resulting program term based on the found unifier.

For a more rigorous justification of the need to modify existing unification algorithms, we will consider various cases of comparison of program terms that make sense when optimizing programs, as well as during their automated development.

Classic unifier highlightingis implemented with complete unification of fragments of program terms, i.e. for terms s and t a common unifier must be foundσ, such that$\sigma(s) =\sigma(t)$. In this case, changes of variables are used to form the final program term p, such that$\sigma(p) =\sigma(s) =\sigma(t)$.

To execute properties$\sigma(p) = s$ and$\tau(p) = t$ substitutionσtransforms into two substitutions:$\alpha(p) = s$ and$\beta(p) = t$. This transformation is obtained as follows. When the unification algorithm detects a program variable that needs to be replaced by another variable or, in general, a term, a corresponding program element p is formed, consisting of one disconnected (new) variable. Next, in the corresponding replacement lists (for program s or t, depending on where the program variable was located and where the term was located), a reverse substitution is made in place of a disconnected variable for one program - a term, for another - a program variable.

If replacement listsαAndβ(to restore programs s and t, respectively) combine and remove substitutions from them in place of disconnected variables - program variables, then you get a complete unifier$\sigma$for the terms of the two original programs.

Let's look at a small example of unifying program fragments for the Lisp language.

<u>Program 1</u>

(append (eval (x) (quote description)))

Its formal representation in the form of a term is as follows:

s = fa(fe(x), fq(b))

Program 2

(append (eval (list(quote define)

(list (quote quote)

(list (list name (list (quote lambda)

(parametrlist description)))))) L))

For clarity, let’s write down the corresponding term line by line:

t = fa(fe(fl(fq(a),
fl (fq(c) ,
fl(fl(d), fl(fq(l)
p, b))))), y)
The following notations are used here: fa , fe , fl , fq - respectively, functional symbols for Lisp functions: append, eval, list, quote; x and y are variables corresponding to program variables x and L; constants a, b, c, d, l - denote program constants define, description, quote, name, lambda.

The complete unification algorithm for this example will find the following unifier:

σ= { x <- fl (fq (a), fl (fq(c) , fl (fl (d), fl(fq(l) p, b)))), y <- fq (b) }.

In this case, the condition is satisfied$\sigma(s) = \sigma(t)$, and the resulting program p, for which the condition would be satisfied$\sigma(p) = s, \tau(p) = t$, there will be a program corresponding to the term p = f_a(fe(x),y). For this program the unifierσwill be transformed into two substitutions:

α= { x <- fl (fq (a), fl (fq(c) , y <- yL },
β= { x <- xx , y <- fq (b) },

where xx and yL correspond to the program variables x and L. These substitutions will ensure that the condition is satisfied$\alpha(p) = t, \beta(p) = s$, which is a necessary condition for unification.

A positive property of obtaining substitutionsαAnd$\beta$is that they are unifiers for matching program p with programs s and t.

Selecting a unified framecharacterized by the fact that complete unification of two program fragments cannot be achieved due to the non-unification of any internal (nested at a deeper level of brackets) arguments. This, as a rule, occurs in the case when the initial function symbols of the allocated local blocks coincide, and, therefore, the number of their arguments coincides, but the arguments themselves do not have a common unifier.

Nevertheless, under these conditions it is possible to find a program term p for which the equality$\sigma(p) = s, \tau(p) = t$. To do this, you can do the following. In the process of comparing terms, their non-unified fragments fx, fy, fz, etc. replace with new term variables, for example, Xa, Xb, Xc, etc. . At the same time, enter the corresponding substitutions Xa <- fx, Xb <- fy, Xc <- fz, etc. into the lists of substitutions separately for term s and term t.

With a specific implementation of such an algorithm, the variables Xa, Xb, Xc may look like program stubs. In the simplest case - automatically replaced comments, in more interesting cases - variables of the “function pointer” type, a list for Lisp programs or preprocessor directives.

Let's give a simple example of identifying a unified framework for two fragments of programs in C++.

Program 1

	Formal presentation
while (n--) {	f_w(f-- (xn), f{ } (
if (Columns != n){	$f_{if\ e}$(f!=(xc, xn), f{ } (
printf ("Error!") fpr(e),	
exit(12);	f_e(a)),
else {	$f_{\{\}}$(
Columns--;	f_{--}(xc),
printf("\n =");	f_{pr}(b))),
}	
Ar[n] = n+1;	$f_{=}$(f[ ] (xa, f++(xn)))))
}	

Program 2

while (Line--) {	f_w(f-- (xl), f{ } (
if (Minutes != Line)	$f_{if\ e}$(f!=(xm, xl),
Minutes = Line;	$f_{=}$(xm, xl),
else {	$f_{\{\}}$(
printf("\n . ");	f_{pr}(k),
wait(1);	f_{wa}(r))))
}	
}	

In this example, the function symbols fw , f-- , f{ } , fif e , f!=, fpr, fe , f=, f[ ] , f++ , fwa correspond to the C++ programming language operations: while do, --, { }, if else, !=, printf, exit, =, [ ] . For the program variables n, Columns, Ar, Line, the term variables xn, xc, xa, xl are selected, respectively. The symbols a, b, e, k, r denote constants: 12, \n =, Error!, \n . , 1 .

It is quite obvious that the terms of the given program fragments cannot be completely unified. At the same time, there is a common framework for them, which can be expressed by the following term p:

p = fw (f-- (x0), f{ } (fif e(f!=(xt, x0), X_a) , f{ } (Xb)) Xc),

in which program variables are replaced by new variables x0, xt, and non-unified program fragments are replaced by term variables Xa, Xb, Xc.

It is easy to show that there are unifiers α And β, making it possible to compare the resulting program p with the original programs s and t:

α(p) = s, α= { x0 <- xn, xt <- xc, Xa <- f{ }(fpr(e), fe (a)),),
Xb <- f{ }(f--(xc), fpr(b)), Xc <- f=(f[ ] (xa, f++(xn))) },

β(p) = t, β= { x0 <- xl, xt <- xm, Xa <- f=(xm, xl),
X_b<- f{ } (fpr(k) , fwa(r)), Xc <- f; } ,

where f; - an empty "skip" operator.

The use of an empty unit operator makes it possible to easily complete the structure of a simpler compared program to the required one so that with partial unification it is possible to identify a common framework. At the same time, the limiting criterion for the use of this method is the expression G(p) < G(s) + G(t).

Identification of terms of software toolsis one of the functions of partial unification of terms of program structures in the case where two program fragments are non-unified in the classical sense, but contain completely unified local fragments.

In such a situation, it is possible not to adhere to the conditions of the relationship$\sigma(p) = s, \tau(p) = t$, since the only way to combine such program fragments is the trivial method described earlier. However, in this case, the term matching algorithm will bring a positive effect in identifying the so-called general software tools, i.e. almost completely identical local program fragments, with the exception of differences in the designation of variables. In practice, this selection of fragments is called a modular approach and is implemented through subroutines or functions written into a common tool library.

Let us give a brief example of such unification for programs in the Pascal language. Let there be two program fragments that calculate in different places the same function for converting positive decimal numbers to the binary number system.

<u>Program 1</u>	Formal presentation
if x > 25 then	f_{if}(f> (x, 25),
begin	f_b(
i := 1;	$f_=$(i, 1),
while x<> 0 do	f_w(f<>(x, 0),
begin	f_b(
res[i] := x mod 2;	$f_=$(f [ ](res, i), fm(x,2)),
x := x div 2;	$f_=$(x, fd(x,2)),
i := i + 1;	f_{++}(i);))
end	))
end	

<u>Program 2</u>	
while m > 0 do	f_w(f>(m, 0),
begin	f_b(
m := m - 1;	f_{--}(m),
i := 1;	$f_=$(i, 1),
while x<> 0 do	f_w(f<>(x, 0),
begin	f_b(
res[i] := x mod 2;	$f_=$(f [ ](res, i), fm(x,2)),
x := x div 2;	$f_=$(x, fd(x,2)),
i := i + 1;	f_{++}(i);))
end	))
end	

This example uses the following functional notation for Pascal language operators: fw ,f>, fb,f--,f=,f<>,f [ ] , fm, fd - respectively: while, >, begin end, decrement, = , <>, [ ], mod, div. For variables and constants in the example, their program designations are preserved.

For this example, the resulting program p could be represented as follows:
X (fb(f--(m), f=(i, 1), fw (f<>(x, 0), f_b(f=(f [ ](res, i), fm(x,2)),
f=(x, fd(x,2)) , f++(i);)))).
In this case, it is not possible to maintain the ratio$\sigma(p) = s, \tau(p) = t$ by using any substitutionsαAndβterms in place of the term variable X. At the same time, we can select a unified common part of the term after X, replacing it with some variable Y. Then the structures of both programs can be expressed quite simply:

s = fif (f> (x, 25), Y), t = fw(f>(m, 0), Y),

preserving instrumental substitution for themσ:
σ= { Y <- (fb(f--(m), f=(i, 1), fw (f<>(x, 0), f_b(f=(f [ ](res, i), fm(x,2)),
f=(x, fd(x,2)) , f++(i);))) }.

If it is necessary to combine programs into a single module (unification composition), it will be possible to use the axioms of equivalent transformation or the method of the trivial menu-merger set out at the beginning of this section.

Thus, we have shown that classical unification algorithms do not make it possible to process all cases of similarity between terms of the formal structure of a program. In order to obtain an acceptable algorithm for partial matching of terms, it is necessary to modify the known algorithms in order to implement all the considered cases:

- classic selection of the unifier,
- highlighting the unified frame,
- highlighting terms of software tools.

We will begin our consideration of this kind of algorithms with the simplest modified unification algorithm, which is based on Robinson's algorithm. Robinson was one of the first to study unification algorithms for algebraic terms based on the so-called string representation of terms.

The first algorithms based on the basic heuristics of enumerating options showed that the complexity of the unification algorithm in time exponentially depends on the length of the unified terms (the number of variables in the terms). The development of unification methods led to works based on lattice theory and the structure separation method. These methods gave better measures of the algorithm's time complexity, rated as almost exponential. In later works, based on the Robinson algorithm, methods for special marking of terms were obtained, which made it possible to reduce the complexity of unification to quadratic. The best characteristics of complexity, approaching linear $O(n + m \log m)$, where m is the number of different variables in a term, have been obtained in recent years based on methods of graphical representation of terms and largely due to specific software implementations that use list representation of data with separate formation of equivalence classes of subterms.

Consideration of the modified Robinson algorithm is very significant, since specific implementations of unification algorithms with good complexity characteristics are largely determined by the speed of operations of modern programming languages working with string (list) representation of data. Such languages include Refal, Lisp, and the input language of the K-system.

Let us describe the modified Robinson algorithm in some abstract algorithmic language, the syntax of which is intuitive due to its similarity with Algol-like languages such as Pascal or C. The input of the algorithm is two comparable terms s and t, the output is a logical variable unifiable, taking the value "true" if the terms programs were partially unified, and "false" - otherwise. In addition, the output of the algorithm is the term p, which is the resulting generalized program, as well as substitution listsαAndβ, satisfying the requirement $G(p) < G(s) + G(t)$ and a list of substitutions$\sigma$for the formation of software tools.

```
unify (s, t) => (unifiable: boolean,
p: program term,α,β,b: substitutions)
begin
  if ( s or t is a variable ) then
begin
Let x be a variable and t another term, then
if x = t then (unifiable, p,α,β,σ) <- ( "true", x,0,0,0) return
else if ( x is in t ) then unifiable <- "false" return
else (unifiable, p,α,β,σ) <- ("true", x, 0, x <- t, 0 ) return
  end
  else
begin
Let s = f (x1, ..., xn), t = g (y1, ..., ym)
if f≠g then
begin
p <- Xfg;
 α<- {Xfg <- f ( x1, ..., xn )},β<- {Xfg <- g (y1, ..., ym )}
{ Select terms
software tools }
   for i <- 1 to n do
for j <- 1 to m do
result <- unify(xi, yj);
compose ( result, (unifiable, p,α,β,σ) )
end for
end for
end
else { (f = g) => ( m = n ) }
                 { Search for a complete unifier or unified framework }
begin
p <- f
unifiable <- "true"
 α,β,σ<- nil
for i = 1 to n
result <- unify( xi , yi )
compose ( result, (unifiable, p,α,β,σ) )
end for
```

```
end
end
return (unifiable, p,α,β,σ)
end
```

The given algorithm provides partial unification of program terms, although it does not claim to be efficient in terms of time complexity. In algorithms, the words “Let” indicate an assumption about the possible structure of the compared elements of terms. The <- signs indicate writing the result of an action to the right of the sign into the structure to the left. The algorithm is recursive and therefore, at a new level of localization of program terms, it uses the unify function again. The compose operation is designed to perform compositions of data structure values (unifiable, p,α,β,σ) obtained at previous levels of comparison with the results of the current level.

The modified Robinson algorithm allows you to significantly optimize programs in source texts in various algorithmic languages, both at the stage of their complete completion and during design. However, its significant drawback is insufficiently complete optimization when identifying terms of software tools. For more complete optimization, it is necessary to consider the unification of all terms of the program structure at different localization levels (each with each). In Robinson's algorithm, this will lead to an unjustified deterioration in the complexity of matching terms over time. Next, we will present algorithms in which the optimization problem when identifying terms of software tools is solved more effectively.

Tree representation of terms became another serious theoretical basis for the creation of modern unification algorithms. The most well-known algorithm using such a representation is the Hewitt unification algorithm, which will be given below in a modified form to meet the requirements for partial unification of program terms.

Representing a program term in the form of a tree assumes that any program construct is designated as the top of a tree, and the simpler syntactic constructs included in it are its descendant nodes. In this case, any linear sequence of operators is always combined into a block construction with the name f{}, as was done in the previous examples of this section. Since under these conditions there are no operators left that are not included in any other operators, the possibility of constructing a program term tree becomes obvious. The only exception is the top level of the program, which can consist of several unblocked statements, but this problem is simply solved either by introducing a fictitious Start node, or by introducing a general block structure of the program (which, as a rule, is already included in the syntax of most algorithmic languages).

To demonstrate the advantages of a tree representation of program terms, consider a short example

<u>Program</u>	Formal presentation
begin	$f_b($
i := i+ 1;	$f_{++}(i),$
while x<> 0 do	$f_w(f<>(x, 0),$

```
begin
res[i] := x mod 2;
x := x div 2;
i := i + 1;
end
end
```

```
fb(
f=(f [ ](res, i), fm(x,2)),
f=(x, fd( x,2 ) ),
        f++(i); ))
)
```

A graphical representation of the term corresponding to this program fragment is shown in Fig. 1. With this representation, it becomes possible to use well-known graph transformation algorithms, as well as to involve, in a specific software implementation of the algorithm, the unification of programming languages that allow the processing of lists or trees (for example, the corresponding C++ class libraries).

Using a program term tree allows you to search in advance for equivalence classes of terms and subterms, as well as use simultaneous processing of equal terms, for which the program graph is converted to a form as shown in Fig. 2.

In this case, identical terms are represented by the same vertices. To indicate the repeated occurrence of a subterm, variable or constant in a term, an additional arc is used.

Let us present a modified Hewitt algorithm for partial unification of program terms.

```
unify (s, t) => (unifiable: boolean ,
p: program term,α,β,b: substitutions)
begin
<List of pairs for unification> <- { s, t }
for (for each node z in s and t)
z.class <- z
while ( <List of pairs to unify>≠ ∅) do
```

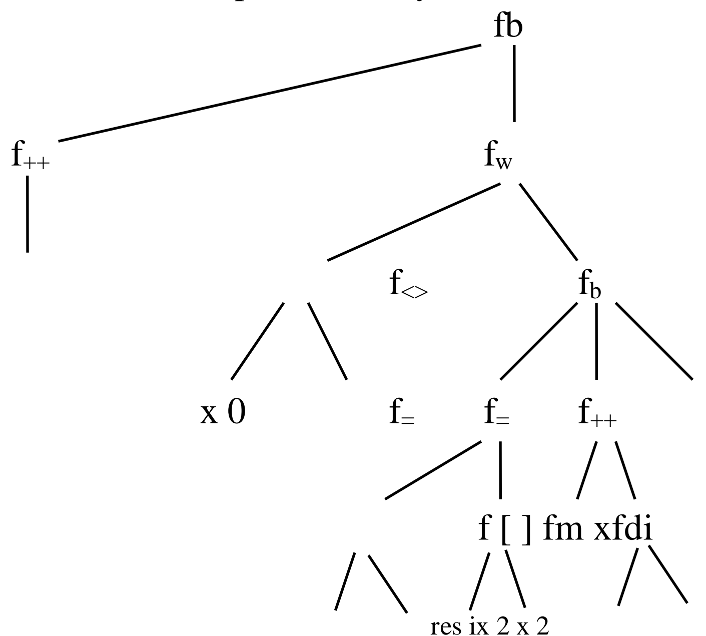

Fig. 1 Tree representation of the program term

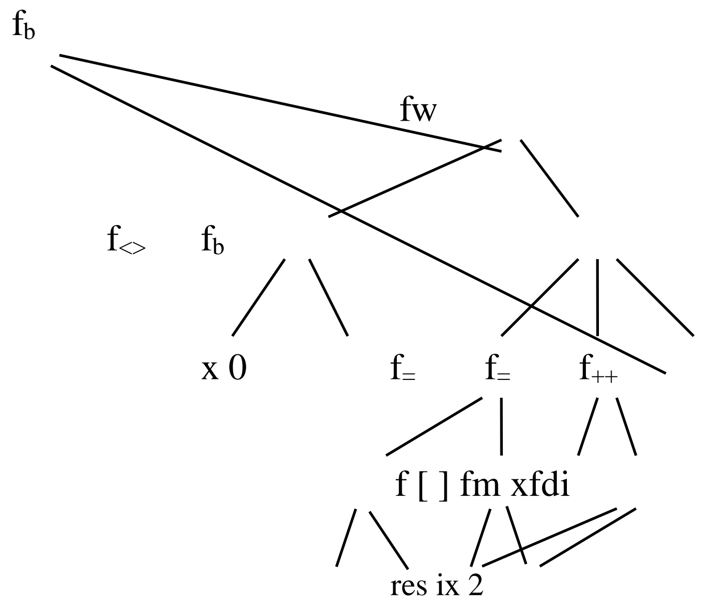

Fig. 2 Modified term representation of the program

```
begin
(x, y) <- pop ( <List of pairs to unify> )
u <- FIND(x)
  v <- FIND(y)
  if(u≠v)then
begin
  if(uand v are not variables and u.symbol≠v.symbol ) then
begin
p <- Xuv
 α<- {Xuv <- f ( x1, ..., xn )},β<- {Xuv <- g (y1, ..., ym)}
  end
   end
w <- UNION (u, v)
if ( w = v and u is a variable ) then
u.class <- v
if ( u and v are not variables ) then
begin
Let (u1, ..., un) = u.subnodes
   Let (v1, ..., vm) = v.subnodes
for i <- 1 to n do
for j <- 1 to n do
push((ui, vj), <List of pairs to unify>);
end for
end for
end
end
end
FORM (unifiable, p,α,β, b)
```

if VERIFY (p, s, t) return (unifiable, p,α,β, b)
end

This algorithm assumes that each of the nodes of the program term graph is described using the following structure:

structure node
symbol: symbol of a function, variable, or constant from a term programs,
subnodes: list of nodes that are children of this node
class: node representing the equivalence class for of this node
end.

The essence of the algorithm is to split the vertices of the tree into equivalence classes, then compare these classes with each other and combine them into a single tree after partial unification. This method of unification allows for very complete processing of the original terms even at different levels of localization. The FIND function finds the current nodes in a term for comparison, the UNION function is designed to combine separated sets of terms, taking into account unifiable data, p,α,β, b. The FORM function generates a new graph made up of the root nodes of the equivalence classes. This graph is the result of partial unification. The final VERIFY procedure checks the criterion $G(p) < G(s) + G(t)$ about the effectiveness of the comparison system produced.

The Robinson and Hewitt algorithms, modified to partially unify program terms, have a number of disadvantages:

- Robinson's algorithm does not allow partial unification of terms of different localization levels,

- both algorithms do not provide for the use of properties of operations, and therefore can classify completely comparable program terms that require preliminary equivalent transformation as non-unifying,

- these algorithms do not use preliminary evaluation functions, such as similarity or difference functions, which reduces their efficiency,

- Hewitt's algorithm uses a large number of preliminary steps to construct term trees and vertex equivalence classes, which also affects efficiency.

From the above, we can conclude that it is advisable to construct a new algorithm that would take into account the listed disadvantages.

Let us propose an algorithm for partial unification of program terms, built on the basis of a matrix representation of trees of program terms.

Let F be some program term containing function symbols f1, f2, ..., fn, variable symbols x1, x2, ..., xm and constant symbols c1, c2, ..., ck. Then the tree that corresponds to the term F can be denoted by GF:

$GF = \langle \rightarrow, \{ f1, f2, ..., fn , x1, x2, ..., xm , c1, c2, ..., ck \} \rangle$,

Where→- subordination relation for the tree vertices.

Let us create an adjacency matrix M for the tree of the program term GF, which will reflect the connections between specific symbols of the syntactic structure of the program. This matrix looks like this:

f1, f2, ..., fn, x1, x2, ..., xm, c1, c2, ..., ck

f1, a11 a12 ... a1n ... ... a1m a1k

f2, a21 a22 ... a2n ... ... a2m a2k

..., ...

fn , an1 an2 ... ann ... ... anm ... ... ank

x1, ...

x2, ...

..., ...

xm, am1 am2 ... amn ... ... amm ... ... amk

c1, ...

c2, ...

..., ...

ck ak1 ak2 ... akn ... ... akm ... ... akk

The matrix element aij is set equal to 1 if the vertices at the intersection of the column and row of which this element is located are connected by an edge of the GF tree, and equal to zero otherwise. In the matrix M, the row and column names can be arranged in the same way as they follow each other in a program term. At the same time, with a large program volume, the adjacency matrix can have a very large dimension, which will complicate its use in real unification programs. It is necessary to find a form of matrix representation in which it occupies an acceptable amount of computer memory. In connection with this, we prove the following theorem.

Theorem

For any adjacency matrix M corresponding to the tree GF of a program term, there is such an arrangement of column and row names f1, f2, ..., fn, x1, x2, ..., xm, c1, c2, ..., ck, with in which each of the rows of the matrix an1 an2 ... ann ... ... anm ... ... ank contains a single chain ai1 ai2 ... ait , all of whose elements are identity. All other elements of the line are zero.

Proof

Consider a tree GF corresponding to some term F. All descendants of any vertex of this tree are one level of tree depth below this vertex. Let's arrange the names f1, f2, ..., fn , x1, x2, ..., xm , c1, c2, ..., ck so that first comes the name of the root vertex, then all its descendant vertices, then all its descendants descendants of the root node, etc. . Let us now assume that in the matrix M there is a certain row in which there are three consecutive elements ai1 ai2 ai3 equal to 1, 0, 1, respectively. This will mean that in the tree corresponding to the term, a vertex has been found or is simultaneously connected to at least two vertices different levels, or with at least two vertices corresponding to the internal elements of different terms. According to our constructions, such a situation cannot happen; therefore, in no row of the matrix M there are subrows ai1 ai2 ai3 equal to 1, 0, 1, respectively. This, in essence, means the validity of the theorem.

Consequence

In the adjacency matrix M there are rows whose element values are only zeros.

Indeed, any program term contains the simplest constructions: variables and constants that have no arguments, and therefore are represented as terminal vertices in the tree corresponding to the term. Terminal vertices do not have edges leading to descendant vertices; therefore, by constructing the matrix, the row of the matrix corresponding to the terminal vertex will be completely zero.

The proven theorem allows us to significantly (by two orders of magnitude) reduce the size of the adjacency matrix. To represent it, it is enough to leave two vectors, one of which will contain the number of the element in the row of the matrix M, from which a continuous unit substring begins, the other - the number of unit elements of this substring. In real unification programs, more memory is spent on the representation of integers that are elements of vectors than on the bit representation of the elements of the matrix M. At the same time, the memory savings are greater, the more complex the program is represented by a matrix, since in this case a very large number of zeros are excluded elements.

When presenting the essence of the proposed unification algorithm, we will, however, use the adjacency matrix itself, leaving the details of its effective implementation in a specific programming language outside the scope of the presentation of the algorithm.

A positive property of the adjacency matrix is that it quite completely describes the structure of the program. This makes it possible to say that to design a partial unification algorithm, it is sufficient to use as initial data only two adjacency matrices corresponding to the input terms of the programs s and t.

Another positive property of the matrix M is the ability to represent a large part of the axioms of the equivalent transformation of programs as properties of operations of any particular programming language by a pair of such matrices. In software implementation, each axiom can be represented by a two-dimensional array of size 2xN, where N is the maximum of the lengths of the terms of the left and right parts of the transformation axiom.

The algorithm is built on the basis of an evaluation function that determines, from fragments of rows of matrix M, the measure of similarity of subterms as classifying them to known types of comparison:

- classic highlighting of the unifier,
- selection of a unified frame,
- highlighting terms of software tools.

The PROMPT scoring function works as follows.

Let the adjacency matrix Ms correspond to the program term s, and the matrix Mt - to the term t.

0) If the algorithm is running for the first time, go to 1, otherwise go to 8.

1) A cycle of viewing the column names of the matrix Mt relative to the matrix Ms is organized in order to search for the first matching names, excluding names corresponding to completely zero rows.

2) If none of the names Ms matches any of the names Mt, then the output of the function is “failure,” which indicates the incomparability of the terms.

3) Let two equal elements fs, from the matrix Ms, and ft, from the matrix Mt (fs = ft), be found.

4) For these elements, the length of the string of names is calculated, which correspond to the child vertices of the vertices fs and ft, and also sequentially coincide in the matrices Ms and Mt. Let this be a number equal to n. Let further the number of ones in the lines corresponding to fs and ft be ms and mt, respectively.

5) If n = ms = mt, then the output of the function for a specific current subterm will be “true”, which indicates the complete unification of the subterms of the terms s and t.

6) If ms > mt or mt > ms , the possibility of selecting a framework is signaled, which will be a term consisting of the names of the columns of the adjacency matrix (in this case, any: Ms or Mt); The output of the function for the current subterms is “partial matching”.

7) If all rows of matrices Ms and Mt have been viewed, complete the algorithm with the possibility of subsequent launch, otherwise go to step 1.

8) Output the previously found function value for the current compared subterms.

A feature of this evaluation function algorithm is the preview and comparison of two matrices with the “marking” of vertices as the main functions (standing at the beginning of a local subterm) of complex terms, which are attributed to the comparability or incomparability of each specific subterm with all others. Thus, classes of comparability of terms are formed. Repeated calls to the PROMPT function make it possible to obtain all the necessary information about the comparability of any subterm with others.

Using the PROMPT function greatly simplifies the task of further partial unification of terms. The main algorithm must now implement the following three components:

- coordination of program variables and variables entered during comparison,
- synthesis of the generalized resulting structure of the program,
- formation of substitutions according to the condition $G(p) < G(s) + G(t)$.

The partial unification algorithm that implements the above functions is as follows.

```
unify+ (s, t) => (unifiable: boolean ,
p: program term,α,β,b: substitutions)
begin
{ Formation of adjacency matrices Ms and Mt based on
terms of input programs s and t }
Matrix_Forming (s, t, Ms, Mt);
{ First PROMPT call }
PROMPT(0, Ms , Mt );
  Let n be the length of the path in the graph
to the current terminal vertex
<List of unprocessed vertices> <-
All names of zero rows of matrices Ms, Mt;
fi <- pop _from_end(<List of unprocessed vertices> );
i <- n;
```

```
if Instrumentary(p,α,β,b, Ms , Mt, fi) then
begin
return ("true", p,α,β,b);
Stop;
end
i <- 0;
<List of unprocessed vertices> <-
All names of non-zero rows of matrices Ms, Mt;
fi <- pop _from_first(<List of unprocessed vertices> );
Frame_result = "failure";
if Frame(p,α,β,b, Ms , Mt, fi) = "failure" then
begin
return ("failure", 0, 0, 0, 0);
Stop;
end
else
if Length (s) + Length (t) < Length (p) then
return ("partial comparability", p,α,β,b);
else
return ("failure", 0, 0, 0, 0);
Stop;
end

recursive procedure Instrumentary( p,α,β,b, Ms , Mt, fi) ;
{Stage of software tools selection}
begin
if PROMPT(fi, Ms , Mt ) = "true" then
begin
i <- i - 1;
if i = 0 then return ("true", p,α,β,b);
fj = parent(fi);
        if Instrumentary(p,α,β,b, Ms , Mt, fj) = "true" then
        (p,α,β,b, Ms , Mt, fi) <- UNION (Result (fj) );
        else
begin
{Take the next one from the raw}
fi <- pop _from_end(<List of unprocessed vertices> );
Instrumentary(p,α,β,b, Ms , Mt, fi) ;
end
end
else return ("failure",0,0,0,0,0,0);
end

recursive procedure Frame ( p,α,β,b, Ms , Mt, fi) ;
```

```
{Frame assembly stage}
begin
if PROMPT(fi, Ms , Mt ) = “true” then
begin
i <- i + 1;
fj = subnodes(fi);
        if Frame(p,α,β,b, Ms , Mt, fj) = “true” then
begin
Frame_result = “partial comparability”;
         (p,α,β,b, Ms , Mt, fi) <- UNION (Result (fj) );
end
        else
begin
{Take the next one from the raw}
fi <- pop _from_first(<List of unprocessed vertices>);
Frame(p,α,β,b, Ms , Mt, fi) ;
end
end
else return (“failure”,0,0,0,0,0,0);
end
```

The algorithm consists of three functions: the main one - unify, and two recursive functions Instrumentary and Frame, designed respectively for selecting instruments and assembling frames.

Here is a list of auxiliary functions used when writing the algorithm:

PROMPT - previously described evaluation function,

pop_from_end - selection of the next, possibly unprocessed during earlier calls of the Instrumentary and Frame functions, the top of the program term tree, starting from the end of the list of names of the adjacency matrix,

pop_from_first - the same thing, starting from the beginning of the list of adjacency matrix names,

Length - function for calculating the length of a program term,

parent - function for calculating the name of the ancestor node for a given node of the term tree,

subnodes - function for calculating the names of descendant vertices for a given node of the term tree,

UNION - a procedure for combining the results of processing nodes at different levels of recursion; its functions include combining substitutions and fragments of the resulting program term with decoupling of program variables.

In other words, the algorithm works as follows.

If the current term and all its descendants up to the terminal vertices, represented in the adjacency matrices by zero rows, are completely unified, but the external term is not, then a software toolkit is formed. If, in addition, the external term of the root level is unified, then the programs are completely unified.

If several (possibly one) levels of a term are unified, but terms of a deeper level are not unified, a software framework is formed.

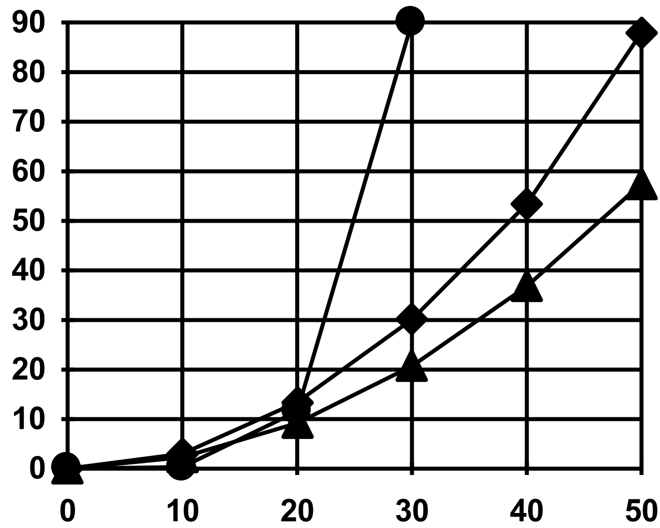

Fig.3. Comparative Difficulty Characteristics

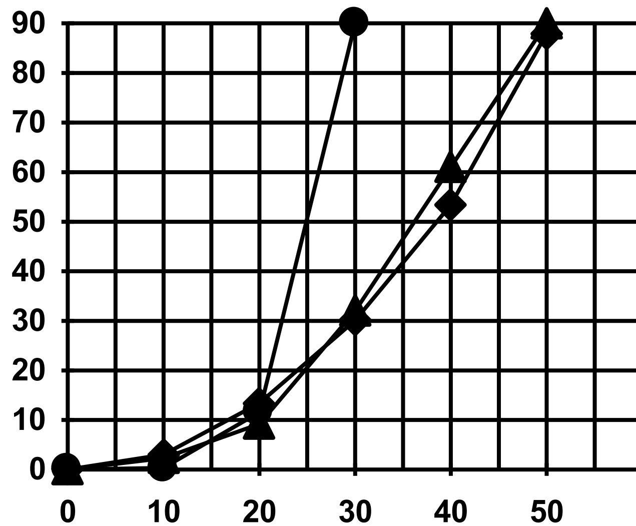

Fig.4. Characteristics of complexity when using axioms

At the end of the algorithm, its result is checked to satisfy the rule G(p) < G(s) + G(t).

For each case of incomparability, an appeal to the axioms of equivalent transformation is used. If such a transformation is possible, it is performed,

otherwise a final decision on incomparability is made. This item is assumed to be included in the Frame and Instrumentary functions.

In the graph, an experimental test of the considered algorithms, carried out on an IBM-486 DX4 PC (66 MHz) in the C programming language, based on average values, showed the following indicators of the time complexity of the algorithms (Fig. 3, 4).

The vertical axis shows the running time of the algorithms in seconds, and the horizontal axis shows the number of connected variables in the terms they process. The graph marked with a circle corresponds to the characteristics of the modified Robinson algorithm, with a square - to the modified Hewitt algorithm, and with a triangle - to the proposed matrix algorithm. The characteristics presented in the graph are characteristics of real programs and therefore largely reflect the characteristics of specific implementations. In addition, the figure shows the characteristics of the matrix algorithm, in which, like the Robinson and Hewitt algorithms, equivalent transformations based on the properties of program operators were not used. When using an equivalent transformation, the complexity of the matrix algorithm is almost the same as the complexity of the modified Hueth algorithm.

Nevertheless, the given characteristics make it possible to see the approximate order of complexity of the algorithms and the feasibility of using a matrix approach with partial unification of program terms.