ISSN 0146-4116, Automatic Control and Computer Sciences, 2013, Vol. 47, No. 7, pp. 373–384. © Allerton Press, Inc., 2013.

Formal Verification of Programs in the Functional Data-flow Parallel Language[1]

M. S. Kropacheva* and A. I. Legalov**

Siberian Federal University, Institute of Space and Information Technology

*e-mail: *ksv@akadem.ru, **legalov@mail.ru*

Received August 23, 2012

Abstract—The article is devoted to the methods of proving parallel programs correctness, that are based on the axiomatic approach. Formal system for functional data-flow parallel programming language Pifagor is described. On the basis of this system programs correctness could be proved.

Keywords: functional data-flow parallel programming, Pifagor programming language, programs formal verification

DOI: 10.3103/S0146411613070225

1. INTRODUCTION

Nowadays there is an overall adoption of parallel programming, caused by the wide use of multicore processors, clusters and graphics processing units for various problems solving. Usually, programs are written in imperative programming languages. The risk of making an error in a parallel program is higher in comparison to the sequential one, as parallel programs have their specific errors.

Since the majority of system fails are the result of software malfunction, it is extremely important to develop reliable software. Formal verification could be used to increase software reliability. Formal verification is a proof of programs correctness by finding the correspondence between the program and its specification, which describes the aim of the development [1]. The main advantage of formal verification is the capability to prove the absence of errors in the program, while testing only allows to detect errors. In contrast to other methods formal verification requires analytical treatment of source code properties, that is why the aim of formal verification could be achieved only by rigorous mathematical proof of program to specification correspondence. This requires formalization of objects used in verification.

One method of formal verification was introduced by Hoare [2]. It utilises an axiomatic approach based on Hoare logic. Hoare logic is an extension of a formal system $\mathcal{J}$ with certain formulas, containing the source code in the verified programming language, that are called Hoare triples. A Hoare triple is an annotated program, namely the source code and two formulas of the theory $\mathcal{J}$, that describe restrictions on input variables and conditions of the program execution result correctness. These formulas are named precondition and postcondition, respectively. A Hoare triple is usually of the form $\{\varphi\}\texttt{Prog}\{\psi\}$, where `Prog` is a program, φ is a precondition and ψ is a postcondition for `Prog`. The extended formal system is distinguished from $\mathcal{J}$ by additional axioms and inference rules, that allow to deduce assertions of program properties, particularly of the program correctness. Then the program is correct if its Hoare triple is identically true. So the main idea of this approach is to derive a formula of formal system $\mathcal{J}$ from the Hoare triple applying rules of inference and using axioms as premises, and then prove the truth of this formula within the formal system $\mathcal{J}$.

There are certain achievements in practical application of such an approach for imperative programming languages [3]. However formal verification complexity for parallel imperative programs increases rapidly for systems with both shared and distributed memory. In general, the main problem is the system resource conflicts. The examples are improper shared memory use for shared memory systems, and deadlocks for distributed memory systems.

An alternative to imperative programming is the functional data-flow paradigm for parallel programming that represents a program as a directed data-flow graph. One implementation of the functional data-flow paradigm is the Pifagor language [4]. The basis of this model is computation control based on the data

[1] The article is published in the original.

readiness. The computations are carried out in the unlimited resources. And functions relationships enable us to represent a program as an acyclic data-flow graph. Parallelism is implemented at the level of operations. So the process of formal verification is simplified as there is no need to analyse the resource conflicts.

Nowadays there exist some works dedicated to functional data-flow programs debugging [5] but formal verification problem is not developed. So development of the formal verification methods for functional data-flow programs is topical.

2. THE AXIOMATIC THEORY FOR THE FUNCTIONAL DATA-FLOW PARALLEL LANGUAGE

Firstly, the formal system for the functional data-flow parallel language Pifagor is needed to perform formal verification of program correctness based on the axiomatic approach. The axiomatic theory for the Pifagor language is developed similarly to the theory for imperative programming languages described in [1]. It is necessary to define:

(1) the language of the theory (alphabet and expressions);

(2) axioms;

(3) inference rules.

2.1. Language of the Axiomatic Theory

The Hoare triples are the main objects of the formal system.

2.1.1. The Pifagor programming language. The operator of interpretation is the main operator that forms a program. It defines a transformation of an argument by a function and has two inputs: one for a function and one for an argument. The operator of interpretation has prefix and postfix forms, which are denoted by the symbols ":" and "^" respectively. Only the postfix form of the operator is used in the article. For instance, `X:F` means that a function `F` is applied to an argument `X`. Some functions used in the article are described below. The detailed description of the Pifagor syntax and semantics could be found in [6].

The "length of the list" function is denoted by the symbol "|". An argument of this function is a datalist of an arbitrary length and any type of elements. A result is an integer corresponding to the number of elements of the list first nested level. If an argument is not of the list type, then the result of the interpretation operation is the error `BASEFUNCERROR`.

The "select list element" function is denoted by `p:n`, where `n` is an integer constant, that is applied to the list `p`. If the argument `p` is not a list then the function returns the error `BASEFUNCERROR`. The error `BOUNDERR0R` is returned if the absolute value of the constant `n` is greater than the list length. In case `n` < 0, the *n*-th element is excluded from the list `p`. When `n` $= 0$ the function returns an "empty element" (a signal value) denoted by ".".

The function `type` returns a type of an object.

2.1.2. The language of logic specification. Preconditions and postconditions are formulas in first-order logic, which is expressive enough for most assertions about the program.

The alphabet of first order logic, functional and predicate symbols, corresponding to functions of the programming language, are used for constructing expressions.

Domain variables (input and output program variables) could be of different types, that is belong to different sets, corresponding to the types of the programming language:

$T = \{$*signal*, *int*, *float*, *char*, *bool*, *func*, *error*, *datalist*, *delaylist*, *parlist*, *asynclisttype*, `user_type`$\}$,

where `user_type` is a set of user-defined types. Here and below braces are used in expressions to denote sets by enclosing the list of its members.

Functions, which together with variables form terms, and predicates, which form formulas, are distinguished in first-order logic. It is not necessary to separate predicates from formulas in this case. Predicates could be considered as a subset of functions with the range of values from the set *bool*. Let us introduce certain functional and predicate symbols, denoting them by the following characters:

(1) arithmetic operations ($+, -, *, /$);

(2) relational symbols ($=, \neq, >, <, \geq, \leq$);

(3) logical operators and quantifiers ($\vee, \wedge, \neg, \Rightarrow, \Leftrightarrow, \forall, \exists$);

(4) "length of the list" function (*len*), "select list element" function (*select*), "is of type" function ($\in$).

The functions *len, select*, $\in$ are equivalent to the corresponding built-in functions of the Pifagor language. The type signatures of the functions mentioned above correspond to the those of the functions and predicates of first-order logic and arithmetic.

Define the set of elementary terms inductively:

1. Any domain variable is a term.

2. If t_1, t_2, ..., t_n are terms then any function $f(t_1, t_2, ..., t_n)$ is a term. In particular, symbols denoting individual constants are nullary function symbols.

3. Terms are confined by the expressions which can be obtained by finitely many applications of rules 1 and 2 to terms.

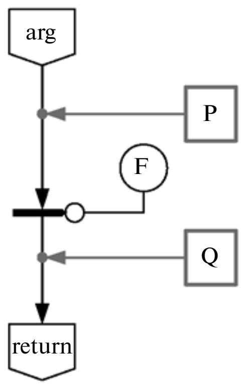

Fig. 1. A Hoare triple for the function `Fun`.

An elementary formula is an elementary term with the range *bool*. Then any formula is inductively defined by the following rules:

1. Any elementary formula is a formula.

2. If A is a formula then $\forall xA(x)$ and $\exists xA(x)$ are formulas.

3. Formulas are confined by the expressions which can be obtained by finitely many applications of rules 1 and 2 to formulas.

Hence, having an alphabet and formation rules one can form preconditions and postconditions for the Hoare triples. Though the form $\{\varphi\}\mathtt{Prog}\{\psi\}$ is not convenient for Pifagor programs, as using braces could cause an ambiguity with braces of delaylists. As the result of this, the following notation is used for the Hoare triple:

$$\boxed{\varphi(x)}\ \mathtt{Prog(x)} \rightarrow r\ \boxed{\psi(r)},$$

`Prog(x)` being a program with input argument x, φ and ψ being the precondition and the postcondition for `Prog` respectively, r denoting the result of the program execution. There are no variables in the Pifagor language, therefore introducing r is necessary for stating the postcondition.

For example, consider a function with the following code:

```
Fun << funcdef arg {
  arg:F >> return
}
```

Let P and Q be the precondition and postcondition for this program. Then the Hoare triple has the following form:

$$\boxed{P(arg)}\ \mathtt{arg:F} \rightarrow r\ \boxed{Q(r)},$$

Since a program written in Pifagor is more demonstrable when represented as a data-flow graph, it is useful to attach a precondition and a postcondition to the edges of this graph. The example of the above triple in the form of data-flow graph is shown in Fig. 1.

2.2. Axioms

Any formal system requires axioms, that is some formulas, which are defined to be true within this system. Let us consider Hoare triples for build-in functions to be axioms. These triples are true by definition and are formed on the basis of semantic rules of the Pifagor language [4, 7, 8]. To describe the method of axioms inference from the semantic rules, consider every semantic rule as a directed tree. For example, consider the rules for the “duplicating” function dup in Fig. 2. The leaves of the tree are possible results of the function execution. To choose the required result, one must start from the root, calculate the values of the expressions in “grey” nodes and proceed to the subtrees, whose parent is the “white” child-node of the current “grey” node, containing the value of the evaluated expression.

Each path of such a tree from the root to a leave corresponds to one axiom. The axiom is formed by the translation of all the expressions of the path to the language of first-order logic. A precondition is a con-

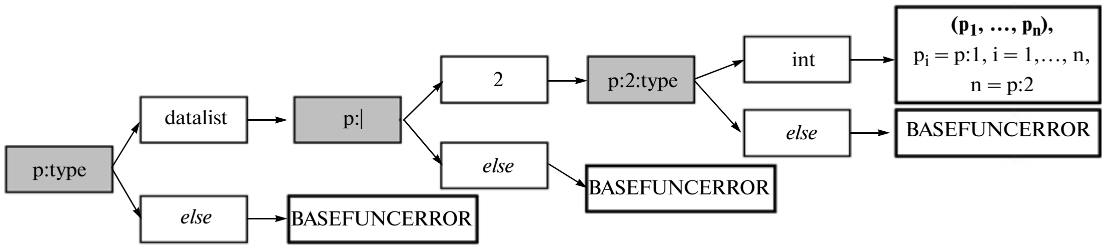

Fig. 2. The semantic rule for "duplicating" function dup represented as a tree.

junction of all argument requirements and a postcondition is an assignment of execution result identifier to the expression in the leaf. All paths with *else* could be merged into a single axiom if the expressions in their leaves are equal. Then, the axioms for "duplicating" function are the following:

$$\boxed{\begin{array}{l}(p \in datalist)\wedge\\ (len(p) = 2)\wedge\\ (select(p, 2) \in int)\end{array}}\ \texttt{p:dup} \to r\ \boxed{\begin{array}{l}(r \in datalist)\wedge\\ (len(r) = select(p, 2))\wedge\\ \forall k(k \in \{1, 2, \dots, select(p, 2)\}\wedge\\ \qquad select(r, k) = (select(p, 1))\end{array}},$$

$$\boxed{\begin{array}{l}\neg(p \in datalist)\vee\\ \neg(len(p) = 2)\vee\\ \neg(select(p, 2) \in int)\end{array}}\ \texttt{p:dup} \to r\ \boxed{r = \texttt{BASEFUNCERROR}}.$$

It should be pointed out that a function `f` could be a result of previous computations. These are two functions: "select list element" function `(p:<int, f >)` and "selector" function `(p:<bool, f>)`. In this case the precondition and the postcondition are formed in the same way, with the exception that the function also occurs in the expression. For example, if `p` is an argument, the axioms of "selector" function `p:f` are:

$$\boxed{f = true}\ \texttt{p:f} \to r\ \boxed{r = p},$$

$$\boxed{f = false}\ \texttt{p:f} \to r\ \boxed{r = .}.$$

2.3. The Inference Rules

To determine whether a Hoare triple is true or not, the inference rules should be defined. These rules allow to bind axioms, based on the built-in functions, to an arbitrary program. The set of axioms and inference rules form the Hoare formal system for the Pifagor language, that enables us to infer the true assertions of the program properties. The main idea of applying the inference rules is a transition from Hoare formal system to the first order logic, namely the step-by-step transformation of a Hoare triple into a first-order formula. Then it is possible to use classical approaches to prove the formula within the first-order logic [9, 10].

There are two alternative ways to prevent the ambiguity of the inference [1]:

1. Using the rules of *forward tracing*, when inference rules are applied form top to bottom of the data-flow graph (from input values towards the result).

2. Using the rules of *backward tracing*, when inference rules are applied from bottom to top of the data-flow graph (from the result towards input values).

Consider the first alternative. By applying the rules of forward tracing, it is possible to transform any Hoare triple, when we start with a function applied directly to input value, namely the function that is executed on the first step. There could be more that one function in case of the functional data-flow parallel

Pifagor language. Then any of such functions could be selected. In the general case the rule of forward tracing for some function with the code "$\mathtt{x : F_1 : F}$" has the following form:

$$\boxed{P_1(x_1)}\ \mathtt{x_1 : F} \to r\ \boxed{Q(r)}\ ,\ A_1, A_2 \vdash\ \boxed{P(x)}\ \mathtt{x : F_1 : F} \to r\ \boxed{Q(r)},$$
$$A_1 := \boxed{\varphi(x)}\ \mathtt{x : F} \to x_1\ \boxed{\psi(x_1)},\quad A_2 := P(x) \Rightarrow \varphi(x),\quad P_1(x_1) := P(x) \Rightarrow \varphi(x) \Rightarrow \psi(x_1), \tag{1}$$

This means that by applying the axiom $\boxed{\varphi(x)}\ \mathtt{x : F} \to r_1\ \boxed{\psi(r_1)}$ to the function $\mathtt{F_1}$, the Haore triple (on the right) is transformed into a new triple (on the left) with the "shorter" program, the precondition $P(x)$ is replaced by $P_1(x)$, and input argument x is replaced by x_1, that is by the result of applying of the function $\mathtt{F_1}$ to x. The turnstile symbol $\vdash$ indicates that if the left Hoare triple is true, then the right triple is also true.

So, sequential application of inference rules leads to the "shortening" of the program, which results in the Hoare triple with an "empty" program: $\boxed{\mathrm{P}}\boxed{\mathrm{Q}}$. This case corresponds to the situation when both precondition and postcondition are attached to the same edge of the data-flow graph. Introduction on the following inference rule enables us to transform this Hoare triple into the first-order logic formula:

$$P \Rightarrow Q \vdash \boxed{\mathrm{P}}\boxed{\mathrm{Q}}. \tag{2}$$

Thus, using inference rules for any Hoare triple one can transform it into the first-order logic formula, the truth of which could be proved within the first-order logic. If the formula is true then the Hoare triple is also true, and therefore the program is correct.

Let us analyse the transformation of precondition by applying the rule of forward tracing. Consider a Haore triple

$$\boxed{P(x)}\ \mathtt{x : f : g} \to r\ \boxed{Q(r)}$$

and the set of axioms for the function $\mathtt{f}$

$$\boxed{P_i(x_a)}\ \mathtt{x}_a : \mathtt{f} \to r_a\ \boxed{Q_i(r_a)},\ i = \overline{1, n},$$

where x, r are the input and output variables of the program that could be used as the identifiers (labels) of input and output edges of the program data-flow graph (it is required that all identifiers are unique), x_a and r_a are the input and output variables in the axioms for the function $\mathtt{f}$.

The algorithm of applying the rule of forward tracing is the following:

1. Set a unique identifier x_1 to the output edge of the expression $\mathtt{x : f}$, as shown in Fig. 3.

2. If condition

$$P(x) \Rightarrow P_i(x), \tag{3}$$

holds, then the forward tracing rule can be used by applying the i-th axiom. This axiom corresponds to the path in the semantic rule tree, whose conditions are thus also satisfied, and the result from the corresponding leaf could be returned. Discard all axioms that do not satisfy (3). As the result, k axioms are left (they could be renumbered from 1 to k). These k axioms correspond to k possible paths of the program execution. Each of them should be considered independently. Since all functions in Pifagor are completely defined, there is at least one axiom that could be used.

3. Transform Hoare triple by Replacing precondition with the expression $P(x) \Rightarrow P_i(x) \Rightarrow Q_i(x_1)$, $i = \overline{1, k}$, at the same time "shortening" the program. The expression $\mathtt{x : f}$ is replaced by the identifier x_1 of the output edge, introduced on step 1 (see Fig. 3b). As the result, as many as k new Hoare triples are derived:

$$\boxed{P(x) \Rightarrow P_i(x) \Rightarrow Q_i(x_1)}\ \mathtt{x_1 : g} \to r\ \boxed{Q(r)},\quad i = \overline{1, k}. \tag{4}$$

The initial Hoare triple is true if all the k derived triples are also true.

Let us make sure that Hoare formal system with the rule of forward tracing is consistent, namely true triples allow to infer only true triples. Consider the forward tracing rule. Let A_1, A_2 and $A = \boxed{P_1(x_1)}\ \mathtt{x_1 : F} \to r\ \boxed{Q(r)}$

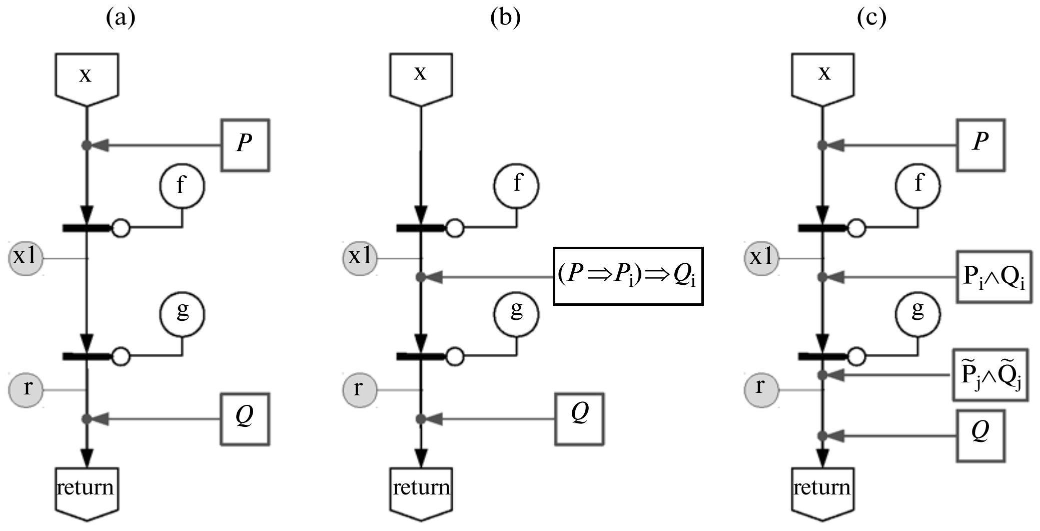
(a)
(b)
(c)
x
P
f
x1
g
r
Q
return
$(P \Rightarrow P_i) \Rightarrow Q_i$
$P_i \wedge Q_i$
$\tilde{P}_j \wedge \tilde{Q}_j$

Fig. 3. Transformations of the Hoare triple with the code "`x:f:g`": a—setting identifier to the output edge of the program data-flow graph subtree, b—the Hoare triple for the program `p:f:g` after applying the rule of forward tracing to the function `f`, c—the Hoare triple with all edges marked.

be identically true. If $P(x)$ is true then by A_2, $\varphi(x)$ is true. Further, by A_1, $\psi(F_1(x))$ is true, and hence, $P_1(F_1(x))$ is true, and finally, by A, $Q(F(F_1(x))) = Q(r)$ is true. The proof for the rule (2) is tautology.

After the application of the rule of forward tracing (1), the precondition P is true by definition, so the expression $P \Rightarrow Q$ could be replaced by $P \wedge Q$, and the postcondition $P \Rightarrow P_i \Rightarrow Q_i$ could be written as $P \wedge P_i \wedge Q_i$.

The data-flow graph could help to demonstrate the process of triple transformation by marking edges with formulas. In this case each transformation of the triple would lead to a new mark for the output edge of the considering function, at the same time previously considered formulas persist. For example in Fig. 3c all edges of the dataflow graph are marked with formulas so it corresponds to the triple with the "empty" program: $\boxed{P(x) \wedge P_i \wedge Q_i \wedge \tilde{P}_j \wedge \tilde{Q}_j}\ \boxed{Q(r)}$.

3. THE ANALYSIS OF FUNCTIONAL DATA-FLOW PARALLEL PROGRAM CORRECTNESS

Depending on the input data, the program could have different paths of execution. Axioms, based on the build-in functions, completely define the tree $\mathcal{I}_0$ of all possible paths of the program execution. The number of different paths of the function execution equals the number of axioms for this function. If the restrictions (precondition) are imposed on the input values, then some paths of the tree become unattainable. These paths correspond to axioms, whose precondition is not derivable from the program precondition. Omitting all unattainable paths, one gets a new tree $\mathcal{I}_1$, that is the subtree of $\mathcal{I}_0$. Each path in $\mathcal{I}_1$ could be considered independently and transformed into a first-order formula using the inference rules. As the result of such transformations, we get exactly k formulas, where k is the number of leaves in $\mathcal{I}_1$. If the truth of each formula implies the truth of the program postcondition, then this program is correct. Otherwise, the program is incorrect, and an error could be in the paths that correspond to formulas that are not identically true. So proving of the program correctness is equivalent to proving the truth of several formulas.

4. THE ANALYSIS OF RECURSION CORRECTNESS

The main problem of the program verification concerns repeatedly executed code, that in case of incorrect program could lead to infinite looping. In this case data-flow graph is infinite. The Pifagor language has no looping constructs and relies solely on recursion. The program is correct if on the one hand, the program terminates in a finite number of steps, and on the other hand, its output result is correct.

Let us analyse the recursive programs peculiarities. If a program contains recursion, then the same code is called several times, and differences concern only its arguments. Then necessary condition of a recursion termination is the sequence of the arguments passed to the recursive function being unduplicated.

It is obligatory for a correct recursive function to have a “branch point”, where the further execution path is selected. In several paths, or in the base cases, the result of the function is produced trivially (without recurring), and in the other (or, recursive) cases, the program recurs (calls itself) and new iteration starts. The choice of the case on each iteration is defined by a certain function on the input arguments. This function can be thought of as a counterpart of an if-else construct in an imperative programming language in the sense that the conditions of recurring or leaving the recurrent function are mutually exclusive, i.e. $\neg(\textit{recursive case} = \textit{true}) \Leftrightarrow (\textit{base case} = \textit{true})$.

Proving of the correctness of recursion may be done by induction (the proof of program termination is similar). Define the notion of the *bound function*. A bound function is a function, bounded above, that maps recursive function argument to the set of natural numbers $\mathbb{N}$, and all arguments, for which the base case is true, are mapped to 1.

Let us consider the proof scheme of a recursive function *Rec* correctness. It is done inductively by the values of the bound function *f*.

The basis: check whether the program is correct for argument $x = p_0$, so that $f(p_0) = 1$.

The inductive step: assume that the program is correct for all arguments for which the bound function values are less than N. A parameter p_N corresponds to the number N such that $f(p_N) = N$. Then it is sufficient to show that

(1) during the $Rec(p_N)$ function execution, the function *Rec* is called recursively only with arguments p_i, $i = \overline{1, n}$, such that $f(p_i) < N$;

(2) parameters p_i, $i = \overline{1, n}$, must be permitted inputs of the function *Rec* (this condition is independent of the bound function and could be used as a primary criterion of the function correctness);

(3) if $Rec(p_i)$, $i = \overline{1, n}$ return correct results then $Rec(p_N)$ terminates and it remains to show that $Rec(p_N)$ returns the correct result.

The algorithm described above is a sufficient condition of the recursive function correctness. If we can not prove the correctness of the function, then either the program is incorrect or the bound function was not properly selected. In the latter case a new bound function is needed.

5. AN EXAMPLE

Let us demonstrate the peculiarity of proving the correctness of a program, which evaluates a quotient and a remainder of division. Such an example is considered in [1] for proving the imperative program correctness. Source code in the Pifagor language of the function evaluating a quotient and a remainder of integer *x* divided by integer *y* is given below:

```
DIV << funcdef arg {
  x<<arg:1; y<<arg:2;
  (x,y,0,x):div_rec >> return
}

div_rec << funcdef arg {
  x<<arg:1; y<<arg:2; q1<<arg:3; r1<<arg:4;
  ({(x,y,(q1,1):+, (r1,y):-):div_rec},(q1,r1)):
  [((y,r1) : [<=, >]):?]:. >> return
}
```

The main function `DIV` takes a datalist with two integers *x* and *y* as an argument, and has a recursive function `div_rec`, which evaluates the quotient and the remainder of division. We define the following triple for the function `DIV` (Fig. 4a)

$$\boxed{\begin{array}{l}(x, y \in int)\wedge \\ (x \geq 0) \wedge (y > 0)\end{array}}\ \texttt{(x,y):DIV} \rightarrow (q, r)\ \boxed{\begin{array}{l}(x = y \cdot q + r)\wedge \\ (r < y)\end{array}}.$$

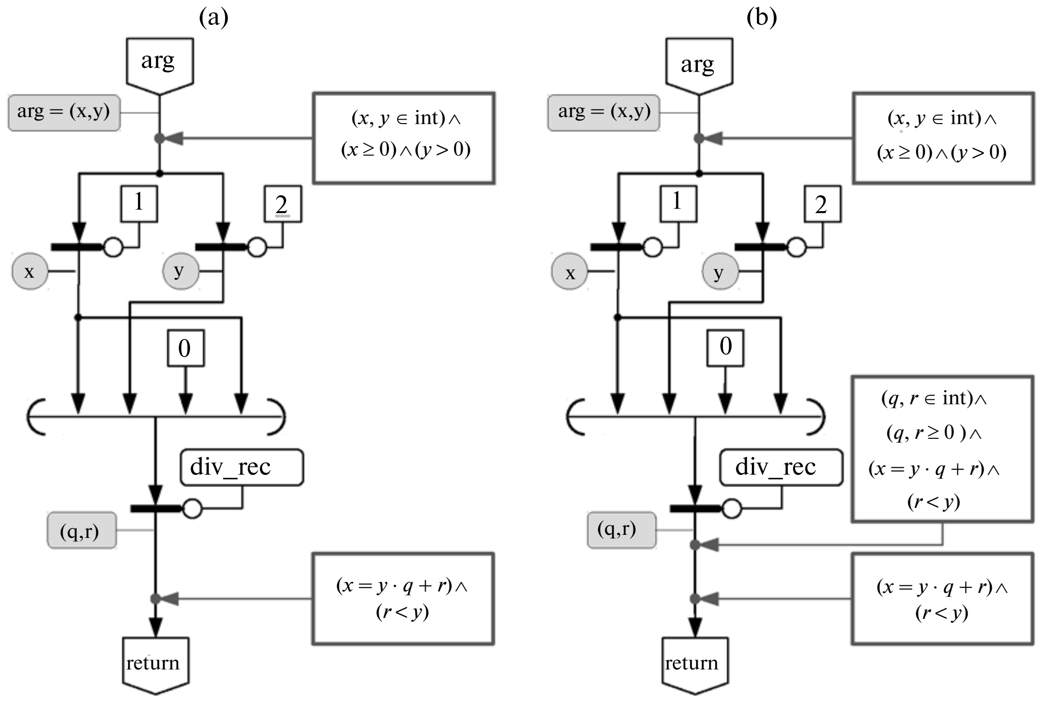
(a)
(b)
arg
arg = (x,y)
$(x, y \in \text{int}) \wedge$
$(x \geq 0) \wedge (y > 0)$
1
2
x
y
0
div_rec
(q,r)
$(q, r \in \text{int}) \wedge$
$(q, r \geq 0) \wedge$
$(x = y \cdot q + r) \wedge$
$(r < y)$
$(x = y \cdot q + r) \wedge$
$(r < y)$
return

Fig. 4. Transformations of Hoare triple for the program DIV: a—the initial Hoare triple, b—the Hoare triple after transformation according the forward tracing rule.

To simplify the description we omit the assignment of identifiers to *x* and *y* with the "select list element" function, and the argument *arg* is represented as a datalist (*x*, *y*).

Firstly let us prove the correctness of the function DIV, supposing that the div_rec function correctness is proved. If the function div_rec is correct, then the following Hoare triple corresponds to correct input arguments:

$$\boxed{\begin{array}{l}(x, y, q_1, r_1 \in int)\wedge\\(x \geq 0) \wedge (y > 0)\wedge\\(q_1 \geq 0) \wedge (r_1 \geq 0)\wedge\\(x = y \cdot q_1 + r_1)\end{array}} \, \texttt{(x,y,q1,r1):div\_rec} \rightarrow (q, r) \, \boxed{\begin{array}{l}(q, r \in int)\wedge\\(q \geq 0) \wedge (r \geq 0)\wedge\\(x = y \cdot q + r)\wedge\\(r < y)\end{array}}. \tag{5}$$

So this triple could be used as a theorem when applying the forward tracing rule to the DIV program triple. There is a one more triple *T* for the function div_rec with the precondition equal to the negation of the triple (5) precondition. But in this case the arguments are incorrect so the result is incorrect and the function DIV is considered as incorrect.

Let us show that the condition (3) is satisfied. The function DIV passes the list of arguments (x,y,0,x) to the function div_rec. Then in the precondition of the function div_rec the variables q1 and r1 are replaced with 0 and x respectively. We have the following expression:

$$P_{DIV}(x, y) \Rightarrow P_{div\_rec}(x, y, 0, x) \equiv ((x, y \in int) \wedge (x \geq 0) \wedge (y > 0))$$
$$\Rightarrow (x, y, 0 \in int) \wedge (x \geq 0) \wedge (y > 0) \wedge (0 \geq 0) \wedge (x \geq 0) \wedge (x = (y \cdot 0 + x))$$
$$\equiv ((x, y \in int) \wedge (x \geq 0) \wedge (y > 0)) \Rightarrow ((x, y \in int) \wedge (y > 0) \wedge (x \geq 0) \wedge (x = x)) \equiv true.$$

Thus, the Hoare triple (5) could be used in the DIV triple transformations with the forward tracing rule (4).

It should be pointed out that the precondition, as it is a negation of the triple *T* could not be deduced from the program precondition, as it is a negation of the triple (5) precondition. So the triple *T* is excluded.

After transformation based on the theorem (5) the precondition of the `DIV` triple has the following form:

$$P_{DIV} \Rightarrow P_{div\_rec} \Rightarrow Q_{div\_rec} \equiv ((x, y \in int) \wedge (x \geq 0) \wedge (y > 0))$$
$$\Rightarrow ((q, r \in int) \wedge (q, r \geq 0) \wedge (x = y \cdot q + r) \wedge (r < y)).$$

If the precondition of the `DIV` program is always considered as true, then the above expression could be written as:

$$(x, y \in int) \wedge (x \geq 0) \wedge (y > 0) \wedge (q, r \in int) \wedge (q, r \geq 0) \wedge (x = y \cdot q + r) \wedge (r < y).$$

The program `DIV` transformation according the forward tracing based on the theorem (3) is shown in Fig. 4b. As the result we got the triple with an "empty" program. It, in turn, could be transformed according to the rule (2) into the following formula:

$$(x, y, q, r \in int) \wedge (x, q, r \geq 0) \wedge (y > 0) \wedge (x = y \cdot q + r) \wedge (r < y) \Rightarrow (x = y \cdot q + r) \wedge (r < y).$$

Obviously this formula is identically true, which means that the program `DIV` is correct.

Let us prove the correctness of the recursive function `div_rec` with the Hoare triple (5). For the sake of simplicity we omit the assignment of identifiers to the elements of the input argument list and argument *arg* is represented as a datalist (x, y, q_1, p_1).

Let the input argument of the function, written in the list form `(x, y, q1, p1)`, be called "the current argument". And the expression for the recursive call `(x, y, (q1, 1) :+, (r1, y):-)` be called "the argument of the recursive call". Let "`(y, r1) :<=`" be called "the condition of recurring". If the expression is true then the branch of the data-flow graph with recursive calls starts executing. Let "`(y, r1) :>`" be called "the condition of leaving" the recurrent function. When this condition is true, the branch of the data-flow graph, that has no recursive call, starts executing. The last condition is true if and only if "the condition of recurring" is false. So the function definitely finishes its execution.

When the execution of the function `div_rec` starts, the input arguments `y` and `r1` form the list `(y, r1)`. Then functions "<=" and ">" are applied to this list. These are built-in functions with the same axioms which differ from each other only in the sign of the operation. For instance, axioms of the function "<=" are the following:

$$\boxed{\begin{array}{l}(p_1, p_2 \in \{int, float\}) \vee \\ (p_1, p_2 \in char) \vee \\ (p_1, p_2 \in bool)\end{array}} \; \texttt{(p1,p2):<=} \rightarrow r_1 \boxed{\begin{array}{l}(r_1 \in bool) \wedge \\ (r_1 = p_1 \leq p_2)\end{array}},$$

$$\boxed{\begin{array}{l}\neg((p_1, p_2 \in \{int, float\}) \vee \\ (p_1, p_2 \in char) \vee \\ (p_1, p_2 \in bool)\end{array}} \; \texttt{(p1,p2):<=} \rightarrow r_1 \boxed{\begin{array}{l}(r_1 \in error) \wedge \\ (r_1 = \texttt{BASEFUNCERROR})\end{array}},$$

The condition (3) holds only for the first axiom. After applying the rule of forward tracing based on the first axiom we get formulas for "the condition of recurring" and "the condition of leaving" the recurrent function. They are $(y \leq r_1)$ and $(y > r_1)$ respectively. The output values of these formulas are boolean constants, that form the datalist of two elements, which is an input argument of the function "`?`". According to the semantics, the function "`?`" returns the sequence number of *true* elements of the list. As "the condition of recurring" and "the condition of leaving" the recurrent function are mutualexcluding, the function "`?`" returns "1" or "2". The given constant is used as a "select list element" function being applied to the list

`({(x, y,(q1, 1):+, (r1, y):-): div_rec}, (q1, r1)).`

This is a way the conditional choice in the Pifagor language is implemented, determining whether the branch of the data-flow graph that has or does not have a recursive call starts executing.

The distinction between "the condition of recurring" and "the condition of leaving" the recurrent function enables us to "divide" the initial triple into two triples, namely, the initial triple (5) is true if the following two triples are also true:

$$\boxed{\begin{array}{l}(x, y, q_1, r_1 \in int)\wedge\\(x \geq 0) \wedge (y > 0)\wedge\\(q_1 \geq 0) \wedge (r_1 \geq 0)\wedge\\(x = y \cdot q_1 + r_1) \wedge (y > r_1)\end{array}} \quad \texttt{(q1,r2):.} \to (q, r) \quad \boxed{\begin{array}{l}(q, r \in int)\wedge\\(q \geq 0) \wedge (r \geq 0)\wedge\\(x = y \cdot q + r)\wedge\\(r < y)\end{array}}, \qquad (6)$$

$$\boxed{\begin{array}{l}(x, y, q_1, r_1 \in int)\wedge\\(x \geq 0) \wedge (y > 0)\wedge\\(q_1 \geq 0) \wedge (r_1 \geq 0)\wedge\\(x = y \cdot q_1 + r_1) \wedge (y \leq r_1)\end{array}} \quad \texttt{prog} \to (q, r) \quad \boxed{\begin{array}{l}(q, r \in int)\wedge\\(q \geq 0) \wedge (r \geq 0)\wedge\\(x = y \cdot q + r) \wedge (r < y)\end{array}}, \qquad (7)$$

where `prog` corresponds to the code "`{(x, y, (q1, 1):+, (r1, y):-):div_rec}:.`". The precondition of the first triple is the conjunction of the program `div_rec` precondition and "the condition of leaving" the recurrent function, and in the code of the triple the "branch point" is replaced by the branch of the data-flow graph without a recursion. The second triple precondition is the conjunction of the program precondition and "the condition of recurring", and in the code the "branch point" is replaced by the branch of the data-flow graph with a recursive call.

Let us consider the triple (7). The function "." executes in the first place and releases the delay of the delay list. It does not change precondition and postcondition. By applying the forward tracing rule based on the axioms for function "–" and "+", the expressions `(q, 1):+` and `(r, y):-` are replaced by formulas $q_2 := (q_1 + 1)$ and $r_2 := (r_1 - y)$, respectively and the following Hoare triple is obtained:

$$\boxed{\begin{array}{l}(x, y, q_1, r_1 \in int)\wedge\\(x \geq 0) \wedge (y > 0)\wedge\\(q_1 \geq 0) \wedge (r_1 \geq 0)\wedge\\(x = y \cdot q_1 + r_1) \wedge (y \leq r_1)\wedge\\(q_2 = q_1 + 1) \wedge (r_2 = r_1 - y)\end{array}} \quad \texttt{(x,y,q2,r2):div\_rec} \to (q, r) \quad \boxed{\begin{array}{l}(q, r \in int)\wedge\\(q \geq 0)\wedge\\(r \geq 0)\wedge\\(x = y \cdot q + r)\wedge\\(r < y)\end{array}}, \qquad (8)$$

Further the function `div_rec` is called recursively with the `(x, y, q2, r2)` "argument of the recursive call".

Let us define the bound function for the function `div_rec`:

$$f(x, y, q_1, r_1) = 1 + x - (y \cdot q_1 + res),$$
$$res = x \bmod y,$$

res being a remainder of x divided by y $((x = q \cdot y + res) \wedge (res < y))$, f being an integer function of integer arguments. f is defined for all (x, y, q_1, r_1) that satisfy the function `div_rec` precondition, as $(x, q_1, r_1 \geq 0)$ and $(y > 0)$, $f(x, y, q_1, r_1) \geq 1$.

Let us prove the correctness of the function `div_rec` inductively by the values of the bound function.

The basis. The program terminates if "the condition of leaving" the recurrent function $y > r_1$ is true for the argument (x, y, q_1, r_1). From the precondition (5) it follows that the expression $x = y \cdot q_1 + r_1$ is true. Also, from the definition of *res* it follows that $res = r_1$. Then

$$f(x, y, q_1, r_1) = 1 + x - (y \cdot q_1 + res) = 1 + (y \cdot q_1 + r_1) - (y \cdot q_1 + res) = 1.$$

Obviously, the result of the function execution satisfies the precondition (5). The formal check of this assertion is done by proving the correctness of the triple (6). The function "." does not change its argument, so taking into account that $(q = q_1)$ and $(r = r_1)$ we conclude that the rule of transformation a triple into the first-order logic formula (2) could be used:

$$((x, y, q_1, r_1 \in int) \wedge (x \geq 0) \wedge (y > 0) \wedge (q_1 \geq 0) \wedge (r_1 \geq 0) \wedge (x = y \cdot q_1 + r_1) \wedge (y > r_1))$$
$$\Rightarrow ((q_1, r_1 \in int) \wedge (q_1 \geq 0) \wedge (r_1 \geq 0) \wedge (x = y \cdot q_1 + r_1) \wedge (r_1 < y)).$$

Obviously, this formula is identically true. It should be pointed out that $f(x, y, q_1, r_1) > 1$ is true if "the condition of leaving" the recurrent function fails.

The inductive step. Let the argument (x, y, q_1, r_1) satisfy the precondition of the triple (5), the "condition of recurring" $y \leq r_1$ and let the value of the bound function for this argument be equal to N: $f(x, y, q_1, r_1) = N$. Assume that the function is correct for all arguments for which the bound function values are less than N. Let us show that the value of the bound function for the argument of the recursive call is always less than N:

$$f(x, y, q_2, r_2) = f(x, y, q_1 + 1, r_1 - y) = 1 + x - (y \cdot (q_1 + 1) + res) = 1 + x - (y \cdot q_1 + res) - y$$
$$= f(x, y, q_1, r_1) - y = N - y < N.$$

Then, according to the inductive assumption, the triple for the function `div_rec`, applied to "the argument of the recursive call", is correct (in the precondition of the formula (5) the terms `q1` and `r1` are replaced by `q2` and `r2` respectively):

$$\boxed{\begin{array}{l}(x, y, q_2, r_2 \in int) \wedge \\ (x \geq 0) \wedge (y > 0) \wedge \\ (q_2 \geq 0) \wedge (r_2 \geq 0) \wedge \\ (x = y \cdot q_2 + r_2)\end{array}} \; \texttt{(x,y,q2,r2):div\_rec} \rightarrow (q, r) \; \boxed{\begin{array}{l}(q, r \in int) \wedge \\ (q \geq 0) \wedge (r \geq 0) \wedge \\ (x = y \cdot q + r) \wedge \\ (r < y)\end{array}}. \qquad (9)$$

The above triple could be used as a theorem for proving the truth of the Hoare triple (8) with the help of the forward tracing rule, when the function `div_rec` is considered as nonrecursive. Firstly let us check the condition (3). It is sufficient to show that the `div_rec` precondition holds for "the argument of the recursive call". We now show that the truth of the precondition of (8) leads to the truth of the precondition of (9). It is equivalent to the following formula:

$$(x, y, q_1, r_1 \in int) \wedge (x \geq 0) \wedge (y > 0) \wedge (q_1 \geq 0) \wedge (r_1 \geq 0) \wedge (x = y \cdot q_1 + r_1) \wedge (y \leq r_1) \wedge (q_2 = q_1 + 1)$$
$$\wedge (r_2 = r_1 - y) \Rightarrow (x, y, q_2, r_2 \in int) \wedge (x \geq 0) \wedge (y > 0) \wedge (q_2 \geq 0) \wedge (r_2 \geq 0) \wedge (x = y \cdot q_2 + r_2).$$

The above formula is true, so the precondition of the function `div_rec` holds for the "the argument of the recursive call". Hence the forward tracing rule based on the theorem (9) could be applied to the triple (7). This transformation results in the Hoare triple with an "empty" program. It is transformed according to the rule (2) into the following formula:

$$((x, y, q_1, r_1 \in int) \wedge (x \geq 0) \wedge (y > 0) \wedge (q_1 \geq 0) \wedge (r_1 \geq 0) \wedge (x = y \cdot q_1 + r_1) \wedge (y \leq r_1) \wedge (q \geq 0)$$
$$\wedge (r \geq 0) \wedge (x = y \cdot q + r) \wedge (r < y)) \Rightarrow ((q, r \in int) \wedge (q \geq 0) \wedge (r \geq 0) \wedge (x = y \cdot q + r) \wedge (r < y)).$$

This formula is identically true, which implies the correctness of the program `div_rec`.

6. CONCLUSIONS

This paper describes an alternative approach to verification of parallel programs written in functional data-flow programming paradigm. A formal system, sufficient for proving the correctness of a program written in the Pifagor language, is considered. The peculiarities of the Pifagor language allows to represent a program as a data-flow graph, which simplifies the process of their debugging and verification. Due to the fact that Pifagor language does not restrict program parallelism, it could be used as an abstract specification for parallel programs. This allows easier debugging and verification with further formal (automatic or manual) transferring the program to the system with specific architecture. Further this method could be used as a base of a toolkit to support program correctness proving, since this method could be made automatic at many stages.

ACKNOWLEDGMENTS

This work was supported by the Federal Targeted Program "Scientific and Scientific-Pedagogical Personnel of Innovative Russia" under the Grant no. 14.A18.21.0396.

REFERENCES

1. Nepomnyashiy, V.A. and Ryakin, O.M., *Applied Methods for Programs Verification*, Moscow: Radio i svyaz, 1988.
2. Hoare, C.A.R., An axiomatic basis for computer programming, *Communications of the ACM*, 1969, vol. 10, no. 12, pp. 576–585.
3. Anureev, I.S., Maryasov, I.V., and Nepomniaschy, V.A., The mixed axiomatic semantics method for C-program verification*, MAIS*, 2010, vol. 17, no 3, pp. 5–28.
4. Legalov, A.I., The functional programming language for creating architecture-independent parallel programs, *Computational Technologies*, vol. 10, no, 1, pp. 71–89; Novosibirsk: Institute of Computational Technologies SB RAS, 2005.
5. Udalova, U.V., Legalov, A.I., and Sirotinina, N.U., Debug and verification of function-stream parallel programs, *Journal of Siberian Federal University. Engineering and Technologies,* 2011, vol. 4, no. 2, pp. 213–224.
6. Legalov, A.I., Kazakov, F.A., Kuzmin, D.A., and Privalihin, D.V., Functional model of parallel computations and the programming language Pifagor, http://www.softcraft.ru/fppp.shtml.
7. Legalov, A.I., The usage of asynchronous lists within the data-flow model of computations, in *The Third Siberian School-Seminar on Parallel Computations*, Tomsk: Tomsk University Press, 2006, pp. 113–120.
8. Kropacheva, M. and Legalov, A., Formal verification of programs in the Pifagor language, *Parallel Computing Technologies, 12th International Confernce PACT September–October, 2013, St. Petersburg, Russia, LNCS 7979*, pp. 80-89.
9. Chang, C. and Lee, R.C., *Symbolic Logic and Mechenical Theorem Proving*, New York: Academic press, 1973.
10. Harrison, J., *Handbook Of Practical Logic And Automated Reasoning*, New York: Cambridge University Press, 2009.