Yugoslav Journal of Operations Research
Volume 20 (2010), Number 2, 229-236
DOI:10.2298/YJOR1002229M

DIJKSTRA'S INTERPRETATION OF THE APPROACH TO SOLVING A PROBLEM OF PROGRAM CORRECTNESS

Branko MARKOSKI
Technical Faculty, "Mihajlo Pupin", University of Novi Sad, Zrenjanin, Serbia
markoni@uns.ac.rs

Petar HOTOMSKI
Technical Faculty "Mihajlo Pupin", University of Novi Sad, Zrenjanin, Serbia

Dušan MALBAŠKI
Faculty of Technical Sciences, Institute of Computing and Control, University of Novi Sad, Serbia

Danilo OBRADOVIĆ
Faculty of Technical Sciences, Institute of Computing and Control, University of Novi Sad, Serbia

Received: June 2006 / Accepted: November 2010

Abstract: Proving the program correctness and designing the correct programs are two connected theoretical problems, which are of great practical importance. The first is solved within program analysis, and the second one in program synthesis, although intertwining of these two processes is often due to connection between the analysis and synthesis of programs. Nevertheless, having in mind the automated methods of proving correctness and methods of automatic program synthesis, the difference is easy to tell. This paper presents *denotative interpretation of programming calculation* explaining *semantics* by formulae φ and ψ, in such a way that they can be used for defining state sets for program P.

Keywords: Dijkstra, denotative interpretation, predicate, terminate, operator.

AMS Subject Classification: 03BXX

1. INTRODUCTION

We are referring to, according to [Čub,1989] the main results based on matemathical-logical approach (Floyd, Manna, Waldinger, Weisman, Ness).

For each program a question of termination and correctness is presented, and for two programs – the question of their equivalence.

Using directed graph, a notion of *abstract (non-interpreted) program* is defined. *Partially interpreted* program is then obtained by using interpretation of functional, predicate and constants symbols. *Realized program* is obtained through the interpretation of free variables within a partially interpreted program. Functioning of realized program may be followed by its *executing sequence*.

According to [Dijkstra, 1988] basic assumptions of *programming logic* are given.

Interesting system is the one which would, starting from initial state, "terminate" in final state (which, as a rule, depends on choice of initial state). We assume that the input value is presented in the choice of initial state and that the output value is presented in the final state. "Condition characterizing a set of all initial states, from which activation surely leads to correct termination of events in such a way that leaves a system in final state satisfying given conclusion is called widest precondition regarding that conclusion" [Dijkstra, 1988].

If a mechanism or machine as a system is noted as S, and desired conclusion as R, than the widest precondition may be noted as follows:

wp(S,R)

where wp is a function of two arguments of S and a predicate R. Semantics of some mechanism is known well enough if we know its predicate transformer, which tells us that for every conclusion R we may derive the widest precondition (noted as wp(S,R)).

We may say that wp is a set of all states, such that execution starts in one of them. If S starts in state satisfying R and if execution terminates, than final state would satisfy R. More harsh condition may be given, that predicate P implies R for all states, i.e.

$P \Rightarrow wp(S,R)$.

If starting state satisfies predicate P, then:

1. S is required to terminate
2. R becomes correct

Since $P \Rightarrow wp(S,R)$ is correct for all states, follows that wp(S,R) is true whenever P is true.

At the start, a mechanism S is placed (identical transformations(, such that no matter conclusion R follows wp(S,R)=R. This mechanism programmers recognize as "empty command". Dijkstra calls it "skip".

If x is defined as a variable substitued by expression E, this command is presented as follows:

x : =E

(where so-called operator "value assignation" Dijkstra calls "acquire value"). Final definition summarizing all beforehand assumptions is:

wp("x:=E",R) = R_E^X for every conclusion R

which may be considered, for every coordinated variable x and for every expression E of appropriate type, as a semantic definition of "value assignation" operator.

To some programmers, natural broadening of value assignation command is very close, so-called "competitive value assignation". Simply, to certain number of different variables value assignation may be done simultaneously. Competitive value assignation is presented as a set (list) of different variables to which value is changed (separated by commas) on the left-hand side of a value assigning operator and a set (list) of identical number of expressions (separated by commas) on the right-hand side of a value assigning operator.

x1,x2 : = E1,E2

x1,x2, x3 : = E1,E2,E3

2. PROGRAMMING CALCULATION

Programming calculation is a special quantification calculation consisting of:

1. Constants and variables called states and state variables, respectively
2. Relation letters of length 1 marked as A,B,C
3. Relation letters of length 2 marked P,S,... called programs
4. Formulae marked $\vartheta,\phi.\Psi$,
5. Special kinds of formulae marked $\{\phi\}$ P $\{\Psi\}$ with main interpretation - if before execution of program P a formula ϕ was correct, than P must terminate and formula Ψ becomes true
6. *Axioms* (precisely, axiom schemes) of assignation

$\vdash \{\varphi^x_e\}$ x := e $\{\varphi\}^+$

where φ^x_e marks expression formula φ where every occurrence of variable x is (simultaneously) replaced by expression e. Symbol "+" marks that from x := e is required to terminate, which reduced to calculation e.

7. Rules of derivation

$$\frac{\varphi \Rightarrow \theta,\ \{\theta\}\ P\ \{\psi\}}{\{\varphi\}\ P\ \{\psi\}}$$

$$\frac{\{\varphi\}\ P\ \{\theta\},\ \theta \Rightarrow \psi}{\{\varphi\}\ P\ \{\psi\}}$$

$$\frac{\{\varphi\}\ P\ \{\theta\},\ \{\theta\}\ Q\ \{\psi\}}{\{\varphi\}\ P; Q\ \{\psi\}}$$

$$\frac{\{\varphi \wedge B\}\ P\ \{\varphi\}}{\{\varphi\}\ \textbf{while}\ B\ \textbf{do}\ P\ \{\varphi \wedge \neg B\}}$$

The first two rules are named consequence rules. The third rule is sequence, and fourth is iteration.

3. DENOTATIONAL INTERPRETATION

Denotational interpretation of programming calculation [Dijkstra, 1988] gives semantics to formulae φ i ψ, υσ, using them to define sets of states for program P.

Let P be a program with set of states D. Set of states may be further interpreted as a set of values for all variables within a program. For denotational interpretation this further precisement is not necessary, so we will stay at abstract states. We define sets D_φ and D_ψ as follows:

$$D_\varphi = \{d \mid (d \in D) \wedge \varphi(d)\} \subseteq D \tag{1.1}$$

$$D_\psi = \{d \mid (d \in D) \wedge \psi(d)\} \subseteq D \tag{1.2}$$

Additionally, mark $\vdash \alpha$ is interpreted as $\alpha = T$, and $\vdash \neg\, \alpha$ as $\alpha = \bot$.

Position 1.1. applies

$$(\forall d)\ ((d \in D) \Rightarrow \vdash \neg\varphi(d)) \Rightarrow (D_\varphi = \varnothing) \tag{1.3}$$

Proof *Statement* $(\forall d)\ (\ (d \in D) \Rightarrow \not\vdash \neg\varphi(d))\ ) \Rightarrow (D_\varphi = \varnothing)$ *is same as*

$$(\forall d)\ ((d \notin D) \vee \vdash \neg\, \varphi(d)) \Rightarrow (D_\varphi = \varnothing) \tag{1.4}$$

If $d \notin D$ applies, then, according to 1.1, $d \notin D_\varphi$. If $\vdash \neg\varphi(d)$ applies, then, also according to 1.1, $d \notin D_\varphi$.

Position 1.2. applies

$$(\forall d)\ ((d \in D) \Rightarrow \vdash \neg\, \psi(d)) \Rightarrow (D_\psi = \varnothing) \tag{1.5}$$

Proof. *Same as in previous position*.

Position 1.3. If φ is defined, applies

$$(\forall d)\ ((d \in D) \Rightarrow \neg \vdash \varphi(d)) \Leftrightarrow (D_\varphi = \varnothing) \tag{1.6}$$

Proof. *Implication from right to left must be proven. We will use contraposition, i.e.* $(p \Rightarrow q) \Leftrightarrow (\neg q \Rightarrow \neg p)$*. Let left side as false. Its negation is*

$$(\exists d')\ ((d' \in D) \wedge \varphi(d'))$$

so, according to (1.1) it means $d' \in D$, i.e. $D_\varphi \neq \varnothing$.

Position 1.4. Position 1.3. applies if in (1.6) φ is replaced by ψ.

1.4.1. Execution function

Let P is a program and D is a set of its states. Then *execution function*, marked [P], is copying

$$[P]: D \to D.$$

Statement formulae φ and ψ are interpreted as copying

$$\varphi, \psi: D \to \{\perp, T\}.$$

All three functions are partial in general case. With this interpretation, predicate

$$\{\varphi\}\ P\ \{\psi\}$$

has meaning: if a program P was in initial state d_0 for which $\varphi(d_0)$ is applied and if it terminates, it will end in a state $d_f \equiv [P](d_0)$ for which $\psi(d_f)$.

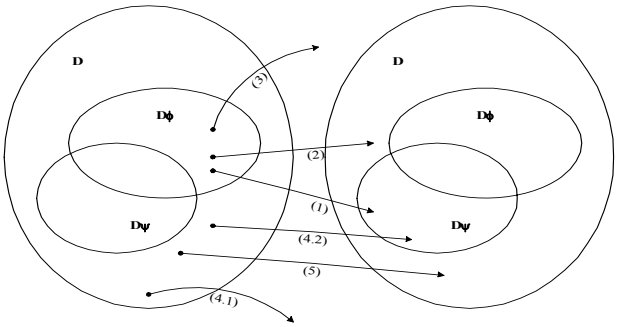

Figure 1

We will consider some special cases connected to the behavior of programming function [P],. assuming that in all the following expressions states belong to set D of program states (i.e. states not connected to program P are excluded).

1. P starts in state $d_0 \in D_\varphi$ and terminates in state $d_f \in D_\psi$ (regular case). Then applies $\{\vdash\!\!\!-\varphi(d)_0\}\ P\ \{\vdash\!\!\!-\ \psi([P](d_0))\}$ where $\psi([P](d_0)) \equiv d_f$.
2. P starts in state $d_0 \in D_\varphi$ and terminates in state $d_f \notin \tilde{D_\psi}$ This means that if functions φ and ψ are well defined (in specification sense), the situation responds to the wrong reaction to good input. In this case applies $\{\vdash\!\!\!-\ \varphi(d)_0\}\ P\ \{\vdash\!\!\!-\ \neg\ \psi([P](d_0))\}$.
3. P starts in state $d_0 \in D_\varphi$ but $[P](d_0)$ is not defined, which means that P from state d_0 does not terminate. Let *termin(s, d)* is a polymorphism with co-domain $\{\perp, T\}$ where s is so-called syntax unit (program, subprogram, program segment, command, even a part of command). Function *termin* has value T if and only if s terminates. Otherwise *termin* gets value ⊥. the fact that P is not terminating from state d_0 means $\vdash\!\!\!-.(d \in D)\ ((d_0, d) \notin [P])$ or, which is the same, $\vdash\!\!\!-\ \neg\ \text{termin}(P, d_0)$.
4. P starts in state $d' \notin D_\varphi$ (i.e. wrong input if φ is well defined according to specification). Then it is possible

   4.1. $\vdash\!\!\!-\ \neg\ \text{termin}(P, d')$ which is all right since it means that program does not terminate for wrong input so it is not robust.

   4.2. $\vdash\!\!\!-\ \text{termin}(P, d') \wedge [P](d') \in D_\psi$, which marks correct reaction to wrong input.
5. P starts in state $d' \notin D_\varphi$ and terminates in state $d'' \notin D_\psi$ which represents wrong reaction to wrong input, i.e. $d' \in D_\varphi \wedge d'' \equiv [P](d') \notin D_\psi$.

Let P be a program with set of states D and executing function [P]. Program P is totally correct considering predicates φ and ψ if and only if it applies

$$\vdash\!\!\!-\ \{\varphi\}\ P\ \{\psi\}^+$$

which means that P must terminate. Program P is partially correct considering predicates φ and ψ if and only if it applies

$$\vdash \{\varphi \wedge (\text{P terminates})\}\ P\ \{\psi\},$$

i.e. if termination is placed within precondition of expression $\{\varphi\}P\{\psi\}$. Notation for partial correctness is

$$\vdash \{\varphi\}\ P\ \{\psi\}$$

Denotational interpretations of total and partial correctness are consequently

$$\vdash \{\varphi\}\ P\ \{\psi\}^{+} \Leftrightarrow \vdash (\forall\, d \in D)\ (\vdash \varphi(d) \Rightarrow \vdash \text{termin}(P,d) \wedge \vdash \psi([P](d)))$$

is total correctness (termination is also required)
and expression

$$\vdash \{\varphi\}\ P\ \{\psi\} \Leftrightarrow\ \vdash (\forall d \in D)(\vdash \varphi(d) \wedge \text{termin}(P,d)\ \Rightarrow \vdash \psi([P](d)))$$

is partial correctness, not requiring termination.

This means that $\vdash \{\varphi\}\ P\ \{\psi\}$may be generated even when program does not terminate.

4. CONCLUSION

The problem of programs analysis and synthesis, being solved by using resolution procedure of proving and deduction of answers, was published by Z. Mann [Mann, 1969-1974]. The other approach to solving problem of program correctness is application of axiomatic definitions of semantics for Pascal programming language, as a special rule of programming logic [Floyd, 1967], [Hoare, 1969], [Hoare, Wirth, 1972-1973]. Comparing these two approaches in solving problem of program correctness it may be concluded that they are significantly different in conception, but with one common feature: deductive system in predicate language. This is explained by derivation methods in definite predicate calculation, based on the formal theory deduction.. In this way, the problem of program correctness is brought into close relationship with automated checkup of existing proofs of mathematical theorems. Realized (deterministic, and these are the only ones considered here) program has only one, or none, executing sequence (when there is no one existing). Partially interpreted program may have several different executing sequences (within it, for every interpreted predicate is known whether it is correct or not, depending on output variables different execution paths are possible). Abstract program always has only one executing sequence (here is not known whether predicate P or its negation ~P is correct).

5. REFERENCES

[1] Floyd, R.W., "Assigning meanings to programs"*, In: Proc. Sym. in Applied Math., Mathematical Aspects of Computer Science, American Mathematical Society*, 19, 1980, 19-32.

[2] Hoare, C.A.R., "An axiomatic basis for computer programming*"*, *Communications of the ACM,* 12 (1980) 576-583.

[3] Hoare, C.A.R., "Notes on data structuring*" In: Dahl, Dijkstra, Hoare Structured programing, Academic Press*, (1980) 83-174.

[4] Hoare, C.A.R., and Wirth, N., *"*An axiomatic definition of the programming language Pascal", *Acta Informatica,* 2,1983, 335-355.

[5] Manna, Z., "The correctness of programs*", Journal of Computer and System Science,* 3 (2) 1979.

[6] Dijkstra, E.W., *A Discipline of Programming*, Prentice –Hall, 1993.

[7] Paul, C., and Jorgensen, *Software Testing*, CRC Press, 1995.

[8] Dijkstra, E.W., *A Discipline of Programming*, Prentice Hall PTR, Upper Saddle River,NJ, 1997.

[9] Sands, D., "Total correctness by local inprovement in program transformation", *Proceedings of the 22nd ACM SIGPLAN –SIGAST symposium on Principles of programming languages*, S. Francisko, California, United States, 1995.

[10] De Boer, F.S., Gabbrielli, and Meo, M.C., "A timed concurrent constraint language", *Information and Computation*, August 25, 2000.

[11] Valencia, F.D., "Reactive constraint programming"*,* Tech.rep., Centre for Basic Research in Computer Science (Brics), Denmark, 2000

[12] Lacey, D., Jones, D.N., Wyk, E.V., and Frederiksen, C.C., "Proving correctness of complier optimations by temporal logic", *Proceedings of the 29th ACM SIGPLAN –SIGAST Symposium on Principles of programming languages*, Portland, United States, 2002.

[13] Harrison-Gegg, T.S, Buce, G.R., Ganetyky, D., Rebecca, Olson, C.M., and Wilson, J.D., "Studing program correctness*", Proceedings of the 8th Annual Coference on Innovation and Technology in Computer Science Education*, Thessaloniki, Greece ,2003.

[14] Scott, E., Zadirov, A., Feinberg, S., and Jayakody, R., *Proceedings of Informing Science Educational Technology Education Joint Conference*, Pori, Finland, 2003.