Corrections to the program verification rules

Tibor Gregorics
gt@inf.elte.hu

Zsolt Borsi
bzsr@inf.elte.hu

ELTE Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary

Abstract

The subject of this paper is a program verification method that takes into account abortion caused by partial functions in program statements. In particular, boolean expressions of various statements will be investigated that are not well-defined. For example, a loop aborts if its execution begins in a state for which the loop condition is undefined. This work considers the program constructs of nondeterministic sequential programs and also deals with the synchronization statement of parallel programs introduced by Owicki and Gries [Owi76]. The syntax of program constructs will be reviewed and their semantics will be formally defined in such a way that they suit the relational model of programming developed at Eötvös Loránd University [Fot88, Fot95]. This relational model defines the program as a set of its possible executions and also provides definition for other important programming notions, like problem and solution. The proof rules of total correctness [Dij76, Fot05, Gri81, Hoa69, Owi76] will be extended by treating abortion caused by partial functions. The use of these rules will be demonstrated by means of a verification case study.

1 Introduction

In mathematics, a partial function is a binary relation from X to Y that does not map every element of X to an element of Y. It is well-known that the expression a/b is not defined if the divisor b is zero. But even the subtraction $x - y$ may have no defined value, it occurs if x and y are natural numbers and x is less than y, and the expected value also should be a natural number. More precisely, $f\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ (where $f(x - y) = x - y$) is a partial function, because not every element of the set $\mathbb{N} \times \mathbb{N}$ is in the domain of f. The value of a partial function is undefined when its argument is out of the domain of the partial function.

In programming, an attempt to divide by zero is handled in various ways depending on the programming environment. It either leads to a compile-time error or produces catchable runtime error if it happens at runtime. In general, evaluation to an undefined value may lead to exception or undefined behaviour. Programmers encounter with expressions on a daily basis, when they are constructing statements. The most commonly used form of the assignment statement sets the value of an expression to a variable. Particularly, boolean expressions are concerned when one writes loop condition or conditions of an alternative command. The value of these expressions is not certainly defined mathematically.

In: E. Vatai (ed.): Proceedings of the 11th Joint Conference on Mathematics and Computer Science, Eger, Hungary, 20th – 22nd of May, 2016, published at http://ceur-ws.org

There are programs that have to work without any error. To be able to reason about the correctness of such programs we need to have rigorous definition of the semantics of the language of these programs. We also need methods that allow us to verify the correctness of programs. When one provides the semantics of a program construct, the functions that are used to build the programming statement (loop condition for example) are assumed to be total functions. The verification methods also consider program descriptions where these functions are well-defined.

This paper focuses on partial functions in program descriptions. In particular, not well defined boolean expressions of various statements will be considered. They will be taken into account when providing the semantics of statements. The semantics of program constructs are given in such a way that they suit an existing relational programming model. This model defines the basic concepts of programming, for example defines the program as a set of its possible executions. A verification method will be presented as well, that handles statements containing partial logical functions. Some rules of the verification method are well known, the new rules will be given along with their proof.

The rest of this paper is organized as follows. Section 2 reviews how partial functions are handled in the literature. Section 3 introduces keywords that are allowed to use to build programs we want to investigate. Next, the semantics of these elementary programs and construct are provided. The mentioned relational programming model is also presented here shortly. Then we provide verification rule for each statement that can be formed from our keywords. Section 4 presents a verification example and illustrates the use of the verification rules given in the previous section. Section 5 summarizes our approach and work.

2 Related work

Z. Manna in [Man74] presents a verification method for flowchart programs. A flowchart program is a diagram constructed by edges and nodes, where the nodes denote statements. $y \leftarrow g(x,y)$ stands for an assignment statement where $g(x,y)$ is a total function mapping $D_x \times D_y$ into D_y.

C. A. Hoare in [Hoa69] did not mention that the conditions of the alternative or loop constructions were total logical functions but his examples showed this.

K. R. Apt and E.-R. Oderog use total logical functions namely Boolean expressions in their work. For example, to ensure that the expressions x div y and x mod y are total they additionally stipulate $x\ div\ y = 0$ and $x\ mod\ y = x$ for the special case of $y = 0$ [Apt97].

D. Gries in his fundamental work on investigating program correctness states that the guards $\beta_1, \ldots \beta_n$ have to be well-defined boolean expressions to make sure that the alternative command avoids abortion. However, in his verification rules, by assuming that all boolean expressions used in program descriptions (i.e. loop conditions, guards of guarded commands) are well-defined, he eliminates this condition to make the verification rules simpler [Gri81].

Williem-Paul de Roever et al. extend Floyd's inductive assertion method for proving sequential transition systems. In sequential transition systems edges are labelled by commands in the form of $c \to f$. In their work c has to be a total boolean condition, but partial state transformations that might lead to runtime error (for example $c \to x := 1/y$ where c is the guard of the command $x := 1/y$) are allowed to use. They present a method for proving that the execution of a given program will not apply undefined operations [Roe01].

3 Theoretical background

3.1 Syntax

A parallel nondeterministic program (let S denote it) can be described with a finite string of symbols including the keywords **skip**, **abort**, **if**, **fi**, **while**, **do**, **od**, [,], **await**, **then**, **ta**, **parbegin**, $\|$ and **parend**, which is generated by the following grammar:

$$
\begin{aligned}
S =&\mathbf{skip} \mid \mathbf{abort} \mid \underline{v} := f(\underline{v}) \mid [S_0] \mid \mathbf{await}\ \beta\ \mathbf{then}\ S_0\ \mathbf{ta} \mid \\
&S_1; S_2 \mid \mathbf{if}\ \pi_1 \to S_1 \ \Box\ \ldots\ \Box\ \pi_n \to S_n\ \mathbf{fi} \mid \mathbf{while}\ \pi\ \mathbf{do}\ S_0\ \mathbf{od} \mid \\
&\mathbf{parbegin}\ S_1 \parallel \cdots \parallel S_n\ \mathbf{parend}
\end{aligned}
$$

where $\underline{v}$ denotes the current variables, $\pi, \pi_1, \ldots, \pi_n, \beta$ are partial logical functions over the current state space, $S_0, S_1, \ldots, S_n$ are programs. The **skip** is the empty program (doing nothing), **abort** is the wrong program (resulting fail), the $\underline{v} := f(\underline{v})$ is the nondeterministic assignment, the $(S_1; S_2)$ is the sequential compisition, the **if** $\pi_1 \rightarrow S_1 \square \ldots \square \pi_n \rightarrow S_n$ **fi** is the nondeterministic conditional statement, the **while** π **do** S_0 **od** is the loop, the $[S_0]$ is the atomic region, the **await** β **then** S_0 **ta** is the await-statement, and **parbegin** $S_1 \parallel \cdots \parallel S_n$ **parend** is the parallel composition.

3.2 Semantics

Before the semantics of the elementary programs and the program constructions described above are shown the concept of the program must be clarified.

All concepts of our programming model as like as the concept of the program are based on the state space. The concept of the state space has already been interpreted in several ways. For many people, the state space is a model of a von Neumann type of computer, others, e.g. Dijkstra [Dij76], associate this notion with the problem to be solved where a state is a compound of the values of the main data types. So, the program is "outside" of the state space operating on it. In our programming model, this second meaning is used. In [Fot88], the notion of the state space is a Cartesian product of the type value set of data types. The only mistake of this obvious definition is that it imposes an order on the components. In [Fot05, Gre12], this mistake has been repaired.

A program is the complex of its executions. An execution is a sequence of states. A program, by definition, can always begin, i.e. at least one execution has to start from each state. The program is nondeterministic because several executions may start from the same state and nobody knows which execution will happen. The first state (start state) of all executions and the last, if the execution is finite, are in the so called base state space. Namely the state space can be permanently changed; the inner states of the executions may have got new components because the program can create and destroy new components (variables) during its execution, so the state space changes dynamically. Two constraints are given: all new components have to be destroyed at the termination, at the very latest, but the base variables should never be removed. The current state always contains the components of the base state space. The variables of the base state space are the base variables; the other variables are the auxiliary variables of the program. Thus the base state space is always a subspace of the current state space. The case when the execution of a program goes wrong will be denoted by a finite sequence of states where the last state is the **fail**.

A sequence can be given by the enumeration of its elements between the signs "<" and ">": $< e_1, e_2, \cdots >$. We will use the interconnection of two sequences if the end of the first sequence is identical to the front of the second one. More precisely, if $\alpha =< a_1, \ldots, a_n >$ and $\beta =< b_1, b_2, \cdots >$ are sequences and $a_n = b_1 \neq$ **fail**, then their interconnection is $\alpha \otimes \beta =< a_1, \ldots, a_n, b_2, \cdots >$.

The formal definition of the program [Gre12] requires some notions. Let H^{**} denote the set of all finite and infinite sequences of the elements of set H. H^{∞} includes the infinite sequences; H^* contains the finite ones. So, $H^{**} = H^* \cup H^{\infty}$ and $H^* \cap H^{\infty} = \emptyset$. The length of the sequence $\alpha \in H^{**}$ is $|\alpha|$, the value of which is ∞ if the sequence is infinite.

Definition 1. *Let A be the so-called base state space and $\bar{A}$ be the set of all states which belong to the state spaces B whose subspace is A, i.e. $\bar{A} = \bigcup_{A \leq B} B$. $\bar{A}$ does not contain the state **fail**. The relation $S \subseteq A \times (\bar{A} \cup \{\textbf{fail}\})^{**}$ is a **program** over A, if*

1. $\mathcal{D}_S = A$
2. $\forall a \in A$ *and* $\forall \alpha \in S(a) : |\alpha| \geq 1$ *and* $\alpha_1 = a_i$
3. $\forall \alpha \in \mathcal{R}_S$ *and* $\forall i (1 \leq i < |\alpha|) : \alpha_i \neq \textbf{fail}$
4. $\forall \alpha \in \mathcal{R}_S : |\alpha| < \infty \rightarrow \alpha_{|\alpha|} \in A \cup \{\textbf{fail}\}$

Now, we are going to give the semantics of the elementary programs and program constructions so that they are treated as programs in the sense of the previous definition. Our aim is to define the set of state-sequences that are mapped to an arbitrary state by a construction.

Definition 2. *Let A be a state space and $\sigma \in A$ be the current state.*

skip(σ) ::= $\{< \sigma >\}$

Definition 3. *Let A be a state space and $\sigma \in A$ be the current state.*

$$\textbf{abort}(\sigma) \ ::= \ \{< \sigma, \textbf{fail} >\}$$

Definition 4. *Let A be a state space and $f \subseteq A \times A$ be a relation and $\sigma \in A$ be the current state.*

$$(\underline{v} := f(\underline{v}))(\sigma) \ ::= \begin{cases} \{< \sigma, \sigma' > \mid \sigma' \in f(\sigma)\} & \textit{if } \sigma \in \mathcal{D}_f \\ \{< \sigma, \textbf{fail} >\} & \textit{if } \sigma \notin \mathcal{D}_f \end{cases}$$

Definition 5. *Let A be the common base state space of the program S_1 and S_2. Let $\sigma \in A$ be the current state.*

$$\begin{aligned} (S_1; S_2)(\sigma) \ ::= & \{\alpha \mid \alpha \in S_1(\sigma) \cap \overline{A}^{\infty}\} \cup \\ & \{\alpha \mid \alpha \in S_1(\sigma) \textit{ and } \mid \alpha \mid < \infty \textit{ and } \alpha_{|\alpha|} = \textbf{fail}\} \cup \\ & \{\alpha \otimes \beta \mid \alpha \in S_1(\sigma) \cap \overline{A}^{*} \textit{ and } \beta \in S_2(\alpha_{|\alpha|})\} \end{aligned}$$

Definition 6. *Let A be the common base state space of the program $S_1 \ldots S_n$ and the conditions $\pi_1 \ldots \pi_n$. Let $\sigma \in A$ be the current state.*

$$(\textbf{if } \pi_1 \to S_1 \,\square \ldots \square\, \pi_n \to S_n \textbf{ fi})(\sigma) \ ::= \ \omega(\sigma) \cup \bigcup_{\substack{i=1 \\ \sigma \in \mathcal{D}_{\pi_i} \wedge \pi_i(\sigma)}}^{n} S_i(\sigma)$$

$$\textit{where } \omega(\sigma) = \begin{cases} \{< \sigma, \textbf{fail} >\} & \textit{if } \exists i \in [1..n] : \sigma \notin \mathcal{D}_{\pi_i} \vee \forall i \in [1..n] : \sigma \in \mathcal{D}_{\pi_i} \wedge \neg \pi_i(\sigma) \\ \emptyset & \textit{otherwise} \end{cases}$$

Definition 7. *Let A be the common base state space of the program S_0 and the condition π. Let $\sigma \in A$ be the current state.*

$$(\textbf{while } \pi \textbf{ do } S_0 \textbf{ od})(\sigma) \ ::= \begin{cases} (S_0; \textbf{while } \pi \textbf{ do } S_0 \textbf{ od})(\sigma) & \textit{if} \quad \sigma \in \mathcal{D}_\pi \wedge \pi(\sigma) \\ \{< \sigma >\} & \textit{if} \quad \sigma \in \mathcal{D}_\pi \wedge \neg \pi(\sigma) \\ \{< \sigma, \textbf{fail} >\} & \textit{if} \quad \sigma \notin \mathcal{D}_\pi \end{cases}$$

Definition 8. *Let A be a state space and $f \subseteq A \times A$ be a relation and $\sigma \in A$ be the current state.*
$[S](\sigma) ::= S(\sigma)$

Definition 9. *Let A be the common base state space of the program S_0 and the condition β. Let $\sigma \in A$ be the current state.*

$$(\textbf{await } \beta \textbf{ then } S_0 \textbf{ ta})(\sigma) \ ::= \begin{cases} [S_0](\sigma) & \textit{if} \quad \sigma \in \mathcal{D}_\beta \wedge \beta(\sigma) \\ (\textbf{skip}; \textbf{await } \beta \textbf{ then } S_0 \textbf{ ta})(\sigma) & \textit{if} \quad \sigma \in \mathcal{D}_\beta \wedge \neg \beta(\sigma) \\ \{< \sigma, \textbf{fail} >\} & \textit{if} \quad \sigma \notin \mathcal{D}_\beta \end{cases}$$

Before the definition of the parallel composition the concept of the "uninterrupted" must be intoduced. Let S be a program and σ be an arbitrary state of its base state space. The execution $S(\sigma)$ is uninterrupted if the program S is the **skip**, **abort**, an assignment statement, an atomic region $[P]$ or an await statement **await** β **then** S_0 **ta** where $\sigma \notin \mathcal{D}_\beta$ or $\beta(\sigma)$ is *true*. (In these cases the await statement is the **abort** or the atomic region $[S_0]$.)

We must remark that in case of $S(\sigma)$ is not uninterrupted (where σ is an arbitrary state and S is a program), then S can be always splitted into two programs so that $S(\sigma) = (u; T)(\sigma)$ so that $u(\sigma)$ is uninterrupted. This u is named as the first statement of S and T is the remainder part of S relative to the state σ.

- If $S =$ **await** β **then** S_0 **ta** and $\sigma \in \mathcal{D}_\beta \wedge \neg\beta(\sigma)$, then u is **skip** and T is **await** β **then** S_0 **ta**.
- If $S = (S_1; S_2)$ and $S_1(\sigma)$ is uninterrupted, then the u is S_1 and T is S_2.
- If $S = (S_1; S_2)$ and $S_1(\sigma)$ is not uninterrupted, then u is the first statement of S_1 and T is the sequence $(T_1; S_2)$ where T_1 is the remainder part of S_1.

- If $S = \textbf{if } \pi_1 \rightarrow S_1 \,\square \ldots \square\, \pi_n \rightarrow S_n \textbf{ fi}$ and one of its conditions is not defined or all conditions are false in σ $(\exists i \in [1..n] : \sigma \notin \mathcal{D}_{\pi_i} \vee \forall i \in [1..n] : \sigma \in \mathcal{D}_{\pi_i} \wedge \neg\pi_i(\sigma))$, then u is **abort** and T is **abort**.
- If $S = \textbf{if } \pi_1 \rightarrow S_1 \,\square \ldots \square\, \pi_n \rightarrow S_n \textbf{ fi}$ and all conditions are defined and some of them are true in σ $(\forall i \in [1..n] : \sigma \in \mathcal{D}_{\pi_i} \wedge \exists i \in [1..n] : \pi_i(\sigma))$, then u is **skip** and T is S_i assuming the i^{th} branch is selected by the scheduler.
- If $S = \textbf{while } \pi \textbf{ do } S_0 \textbf{ od}$ and its condition is not defined $(\sigma \notin \mathcal{D}_\pi)$, then u is **abort** and T is **abort**.
- If $S = \textbf{while } \pi \textbf{ do } S_0 \textbf{ od}$ and the condition is false in σ $(\sigma \in \mathcal{D}_\pi \wedge \neg\pi(\sigma))$, then u is **skip** and the reminder part of S is **skip**.
- If $S = \textbf{while } \pi \textbf{ do } S_0 \textbf{ od}$ and the condition is true in σ $(\sigma \in \mathcal{D}_\pi \wedge \pi(\sigma))$, then u is **skip** and T is $(S_0; \textbf{while } \pi \textbf{ do } S_0 \textbf{ od})$.
- If $S = \textbf{parbegin } P_1 \parallel \cdots \parallel P_i \parallel \cdots \parallel P_n \textbf{ parend}$ and its i^{th} branch is selected by the scheduler and $P_i(\sigma)$ is uninterrupted, then u is P_i and T is $\textbf{parbegin } P_1 \parallel \cdots \parallel P_{i-1} \parallel P_{i+1} \parallel \cdots \parallel P_n \textbf{ parend}$.
- If $S = \textbf{parbegin } P_1 \parallel \cdots \parallel P_i \parallel \cdots \parallel P_n \textbf{ parend}$ and its i^{th} branch is selected by the scheduler and $P_i(\sigma)$ is not uninterrupted, then u is the first statement of P_i and T is $\textbf{parbegin } P_1 \parallel \cdots \parallel T_i \parallel \cdots \parallel P_n \textbf{ parend}$ where T_i is the remainder part of P_i.

Definition 10. *Let A be the common base state space of the programs $S_1, \ldots, S_n$ and $\sigma \in A$ be the current state.*

$$(\textbf{parbegin } S_1 \parallel \cdots \parallel S_i \parallel \cdots \parallel S_n \textbf{ parend})(\sigma) \ ::= \bigcup_{i=1}^{n} B_i(\sigma)$$

where

$$B_i(\sigma) = \begin{cases} (S_i; \textbf{parbegin } S_1 \parallel \cdots \parallel S_{i-1} \parallel S_{i+1} \parallel \cdots \parallel S_n \textbf{ parend})(\sigma) \\ \qquad \textit{if } S_i(\sigma) \textit{ is uninterrupted} \\ (u_i; \textbf{parbegin } S_1 \parallel \cdots \parallel S_{i-1} \parallel T_i \parallel S_{i+1} \parallel \cdots \parallel S_n \textbf{ parend})(\sigma) \\ \qquad \textit{if } S_i(\sigma) = (u_i; T_i)(\sigma) \textit{ where } u_i(\sigma) \textit{ is uninterrupted} \end{cases}$$

3.3 Verification

Informally, a program is correct if it satisfies the intended input/output relation. Program correctness is expressed by so-called correctness formulas. These are statements of the form

$$\{\{Q\}\}S\{\{R\}\}$$

where S is a program and Q and R are assertions. The assertion Q is the precondition of the correctness formula and R is the postcondition. The precondition describes the set of initial states in which the program S is started and the postcondition describes the set of desirable final states.

More precisely: a correctness formula is *true* if every excecution of S that starts in a state satisfying Q is finite (it terminates) and its final state satisfies R. (This is the concept of the total correctness. The partial correctness is omitted in this paper.)

Reasoning about correctness formulas in terms of semantics is not very convenient. Hoare has introduced a proof system allowing us to prove partial correctness of deterministic programs in a syntax-directed manner, by induction on the program syntax [Hoa69]. Dijkstra, Gries and Owicki have developed this system for nondeterministic and parallel programs [Dij76, Gri81, Owi76]. Now this system is going to be extended.

The first six rules are well-known, they are only shown for the sake of completeness without their proofs.

Theorem 1. *Let Q and R be two assertions.*

$$\frac{Q \implies R}{\{\{Q\}\} \textbf{ skip } \{\{R\}\}}$$

Theorem 2. *Let Q and R be two assertions, $\underline{v} := f(\underline{v})$ be an assignment.*

$$\frac{Q \implies \underline{v} \in \mathcal{D}_f \wedge \forall \underline{e} \in f(\underline{v}) : R^{\underline{v} \leftarrow \underline{e}}}{\{\{Q\}\}\ \underline{v} := f(\underline{v})\ \{\{R\}\}}$$

The $R^{\underline{v}\leftarrow\underline{e}}$ means that the components of $\underline{v}$ must be substituted for the corresponding components of $\underline{e}$. In that case when the relation $f \subseteq A \times A$ of the assingment is a total function, i.e., f is a (deterministic) function mapping from A to A and $\mathcal{D}_f = A$, the rule of assigment can be written in the following form.

$$\frac{Q \implies R^{\underline{v}\leftarrow f(\underline{v})}}{\{\{Q\}\}\ \underline{v} := f(\underline{v})\ \{\{R\}\}}$$

Theorem 3. *Let Q and R be two assertions, S be a program.*

$$\frac{\{\{Q\}\}\ S\ \{\{R\}\}}{\{\{Q\}\}\ [S]\ \{\{R\}\}}$$

Theorem 4. *Let Q and R be two assertions, S_1 and S_2 be two programs.*

$$\frac{\begin{array}{c}\exists Q' : A \to \mathbb{L} \\ \{\{Q\}\}\ S_1\ \{\{Q'\}\} \\ \{\{Q'\}\}\ S_2\ \{\{R\}\}\end{array}}{\{\{Q\}\}\ (S_1; S_2)\ \{\{R\}\}}$$

Theorem 5. *Let Q and R be two assertions, and S^* stand for the program S annotated with assertions such as preconditions of the assignments or invariants of the loops.*

$$\frac{\{\{Q\}\}\ S^*\ \{\{R\}\}}{\{\{Q\}\}\ S\ \{\{R\}\}}$$

The next three rules take into consideration the cases when some logical functions of the construction is not well-defined. These rules are the extensions of the well-known versions that use well-defined logical functions. An assertion P will be interpreted as the set of states that satisfy P many times in the following rules. From its context it can be decided which interpretation holds. For example, Q and R are assertions in the expression $Q \implies R$ but they are sets in $Q \subseteq R$.

Theorem 6. *Let Q and R be two assertions, and $S_1, \ldots S_n$ be programs and $\pi_1, \ldots \pi_n$ be conditions.*

$$\frac{\begin{array}{c}Q \implies \pi_1 \vee \cdots \vee \pi_n \\ Q \subseteq \mathcal{D}_{\pi_1} \cap \cdots \cap \mathcal{D}_{\pi_n} \\ \forall i \in \{1, \ldots, n\} : \{\{Q \wedge \pi_i\}\}\ S_i\ \{\{R\}\}\end{array}}{\{\{Q\}\}\ \textbf{if}\ \pi_1 \to S_1 \,\square \ldots \square\, \pi_n \to S_n\ \textbf{fi}\ \{\{R\}\}}$$

Proof. We must show that the executions of the conditional statement starting from Q finish at R. Let q be an arbitrary state of Q. Since $Q \subseteq \mathcal{D}_{\pi_1} \cap \cdots \cap \mathcal{D}_{\pi_n}$ and $Q \implies \pi_1 \vee \cdots \vee \pi_n$, according to the definition each execution of the conditional statement starting from q belongs to the executions of the component S_i where its condition π_i is satisfied by q.

These executions terminate in a state satisfying R because $\forall i \in \{1, \ldots, n\} : \{\{Q \wedge \pi_i\}\}S_i\{\{R\}\}$ thus $\{\{Q\}\}$ **if** $\pi_1 \to S_1 \,\square \ldots \square\, \pi_n \to S_n$ **fi** $\{\{R\}\}$ holds. $\square$

Theorem 7. *Let Q and R be two assertions, and S_0 be a program and π be a condition.*

$$\frac{\begin{array}{c}\exists I : A \to \mathbb{L}\ \textit{and}\ \exists t : A \to \mathbb{Z} \\ Q \implies I \\ I \subseteq \mathcal{D}_\pi \\ I \wedge \neg\pi \implies R \\ I \wedge \pi \implies t \geq 0 \\ \{\{I \wedge \pi\}\}\ S_0\ \{\{I\}\} \\ \forall c_0 \in \mathbb{Z} : \{\{I \wedge \pi \wedge t = c_0\}\}\ S_0\ \{\{t < c_0\}\}\end{array}}{\{\{Q\}\}\ \textbf{while}\ \pi\ \textbf{do}\ S_0\ \textbf{od}\ \{\{R\}\}}$$

Proof. We need to prove that the executions of the loop starting from Q are finite and they finish at R. Let q be an arbitrary state of Q. Since $Q \implies I$ and $I \subseteq \mathcal{D}_\pi$ thus $q \in \mathcal{D}_\pi$.

The execution starting from q can be splitted into the sections of the executions generated by the body S_0. Each section is started at a state satisfying $I \wedge \pi$ and terminates at a state satisfying I (see the cond. $\{\{I \wedge \pi\}\}S_0\{\{I\}\}$). If the total execution is finite, its last section finishes at a state satisfying $\neg\pi$ or the state q own satisfies $\neg\pi$ if the loop stops at once. It means that each finite execution strating from q finishes at a state of R since $I \wedge \neg\pi \implies R$.

The proof will be complete if we show that there is no infinite execution from q. If there would be an infinite execution, it should consist of infinite sections. Let us consider the infinite sequence of integers that is mapped from the beginning states of the sections by the function t. Because of the criterion $\forall c_0 \in \mathbb{Z} : \{\{I \wedge \pi \wedge t = c_0\}\}\ S_0\ \{\{t < c_0\}\}$ this sequence should be strictly monotone decreasing thus it should contain negative numbers. However all numbers must be nonnegative because of the criterion $I \wedge \pi \implies t \geq 0$. This is a contradiction. □

Theorem 8. *Let Q and R be two assertions, and S_0 be a program and β be a condition.*

$$\frac{\begin{array}{c} Q \subseteq \mathcal{D}_\beta \\ \{\{Q \wedge \beta\}\}\ S_0\ \{\{R\}\} \end{array}}{\{\{Q\}\}\ \textbf{await}\ \beta\ \textbf{then}\ S_0\ \textbf{ta}\ \{\{R\}\}}$$

Proof. It is enough to show that the executions of the conditional statement starting from Q finish at R. Let q be an arbitrary state of Q. If $q \in \mathcal{D}_\beta$ and q satisfies β, the executions of the await-statement starting from q are identical to the executions of the atomic region S_0 starting from q. These executions finish at a state in R because of $\{\{Q \wedge \beta\}\}\ S_0\ \{\{R\}\}$. □

The last rule is about the parallel composition.

Theorem 9. *Let Q, Q_1, … Q_n and R, R_1, … R_n be assertions, S_1, … S_n be programs, and S_1^*, … S_n^* be the annotations of the corresponded programs.*

$$\frac{\begin{array}{c} Q \implies Q_1 \wedge \ldots \wedge Q_n \\ \forall i \in \{1, \ldots, n\} : \{\{Q_i\}\}S_i^*\{\{R_i\}\} \\ \textit{and they are interference free} \\ R_1 \wedge \ldots \wedge R_n \implies R \end{array}}{\{\{Q\}\}\ \textbf{parbegin}\ S_1 \parallel \cdots \parallel S_n\ \textbf{parend}\ \{\{R\}\}}$$

where standard proof outlines $\{\{Q_i\}\}S_i^*\{\{R_i\}\}$, $i \in \{1, \ldots, n\}$, are called interference free if no normal assignment or atomic region u of a component program S_i interferes with the proof outline $\{\{Q_j\}\}S_j^*\{\{R_j\}\}$ of another component program S_j where $i \neq j$. We say that u does not interfere with $\{\{Q\}\}S^*\{\{R\}\}$ if the following conditions are satisfied:

1. *for all assertions r in $\{\{Q\}\}S^*\{\{R\}\}$ the formula $\{\{r \wedge pre(u)\}\}u\{\{r\}\}$ holds, where $pre(u)$ is the precondition of u in the annotation S^*,*
2. *for all termination function $t : \bar{A} \to \mathbb{Z}$ in $\{\{Q\}\}S^*\{\{R\}\}$ where A is the base state space of the parallel composition the formula $\{\{pre(u) \wedge t = c_0\}\}\ u\ \{\{t \leq c_0\}\}$ holds, where c_0 is an arbitrary integer.*

4 Case Study

Consider the following problem: given an array x of n integer numbers and an integer number k. Count the elements of x that are divisors of k.
Specification of the problem can be given in the following way:
$A = (x\colon \mathbb{Z}^n, k\colon \mathbb{Z}, count\colon \mathbb{N})$

$Pre = (x = x')$

$Post = (Pre \wedge count = \sum\limits_{j=1}^{n} \chi(x[j] \mid k))$
where $\chi\colon \mathbb{L} \to \{0,1\}$ and $\chi(true) = 1$ and $\chi(false) = 0$

Let S denote the following program:

```
i, count := 1, 0
while  i ≤ n  do
   if
      x[i] | k → count := count + 1   □
      x[i] ∤ k → skip
   fi ;
   i := i+1
od
```

Let Q' denote the intermediate assertion of the sequence S, between the initialisation and the loop, Q'' the intermediate assertion that holds before executing the assignment $i := i + 1$, Inv the invariant and t the variant function of the loop.

Now we shall to prove that $\{\{Pre\}\}\ S\ \{\{Post\}\}$ holds. Since S is a sequence, due to Theorem 4. it is sufficient to prove that

1. $\{\{Pre\}\}\ i, count := 1, 0\ \{\{Q'\}\}$ and
2. $\{\{Q'\}\}\ DO\ \{\{Post\}\}$, where DO denotes the loop:
   while $i \le n$ **do if** $x[i] \mid k \to count := count + 1 \quad \Box\ x[i] \nmid k \to$ **skip fi**; i:=i+1 **od**
   and $Q' = (Pre \wedge count = 0 \wedge i = 1)$ is given.

Let us prove the two conditions:

1. $\{\{Pre\}\}\ i, count := 1, 0\ \{\{Q'\}\}$
   By replacing i with 1 and $count$ with 0 in Q' we obtain $(Pre \wedge 0 = 0 \wedge 1 = 1)$, that is Pre. Obviously $Pre \implies Pre$ holds. We proved that $Pre \implies Q'^{i\leftarrow 1, count\leftarrow 0}$ holds. Now, by the remark of Theorem 2. $\{\{Pre\}\}\ i, count := 1, 0\ \{\{Q'\}\}$ is deduced.
2. $\{\{Q'\}\}\ DO\ \{\{Post\}\}$
   Instead of proving this verification condition, due to Theorem 7. it is sufficient to prove that
   (a) $Q' \implies Inv$ and
   (b) $Inv \subseteq \mathcal{D}_{i \le n}$ and
   (c) $Inv \wedge \neg(i \le n) \implies Post$ and
   (d) $Inv \wedge i \le n \implies n - i \ge 0$ and
   (e) $\forall c_0 \in \mathbb{Z}$: $\{\{Inv \wedge i \le n \wedge n - i = c_0\}\}\ S_0\ \{\{Inv \wedge n - i < c_0\}\}$

   where S_0 denotes the loop body **if** $x[i] \mid k \to count := count + 1 \Box\ x[i] \nmid k \to$ **skip fi**; i:=i+1.
   and the loop invariant Inv and the variant function t are given as follows:
   $Inv = (Pre \wedge i \in [1..n + 1] \wedge count = \sum_{j=1}^{i-1} \chi(x[j] \mid k))$
   $t = n - i$
   Let us prove the conditions separately:
   (a) $Q' \implies Inv$
   We have to prove that Q' implies
      i. Pre
      This holds since Q' contains Pre.
      ii. $i \in [1..n + 1]$
      Since $i = 1$, i is an element of the not empty set $[1..n + 1]$. The interval $[1..n + 1]$ is not empty, because n, that is the length of the array x, is zero or a positive integer number.
      iii. $count = \sum_{j=1}^{i-1} \chi(x[j] \mid k)$
      Due to condition Q', $count = 0$ and $i = 1$. Thus the sum is empty and its value is also 0.

(b) $Inv \subseteq \mathcal{D}_{i \leq n}$
Since $i \leq n$ is a well-defined logical function, its domain contains all states of the statespace, including those that satisfy Inv.

(c) $Inv \wedge \neg(i \leq n) \implies Post$

i. Pre
The invariant contains the precondition, therefore Inv implies Pre.

ii. $count = \sum_{j=1}^{n} \chi(x[j] \mid k)$
Since $i \in [1..n+1]$ and $\neg(i \leq n)$, therefore we get $i = n+1$.
This, together with $count = \sum_{j=1}^{i-1} \chi(j \mid k))$ we know from Inv, yields the desired condition.

(d) $Inv \wedge i \leq n \implies n - i \geq 0$
This holds because due to the loop condition $i \leq n$ is true.

(e) $\forall c_0 \in \mathbb{Z}$: $\{\{Inv \wedge i \leq n \wedge n - i = c_0\}\}\ S_0\ \{\{Inv \wedge n - i < c_0\}\}$
Let c_0 be an arbitrary integer number. Since the loop body S_0 is a sequence, we use Theorem 4. with $Inv \wedge i \leq n \wedge n - i = c_0$ as Q and with $Inv \wedge n - i < c_0$ as R. It is sufficient to prove the following two conditions:

i. $\{\{Inv \wedge i \leq n \wedge n - i = c_0\}\}\ IF\ \{\{Q''\}\}$ and
ii. $\{\{Q''\}\}\ i := i + 1\ \{\{P \wedge n - i < c_0\}\}$

where IF denotes the conditional statement **if** $x[i] \mid k \rightarrow count := count + 1 \square\ x[i] \nmid k \rightarrow$ **skip fi** and the intermediate assertion of the loop body is Q'' is given: $Q'' = Inv^{i \leftarrow i+1} \wedge n - i = c_0$

i. $\{\{Inv \wedge i \leq n \wedge n - i = c_0\}\}\ IF\ \{\{Q''\}\}$
Due to Theorem 6., the following conditions are sufficient to prove:

A. $Inv \wedge i \leq n \wedge n - i = c_0 \implies (x[i] \mid k \vee x[i] \nmid k)$ and
B. $Inv \wedge i \leq n \wedge n - i = c_0 \subseteq \mathcal{D}_{x[i] \mid k} \cap \mathcal{D}_{x[i] \nmid k}$ and
C. $\{\{Inv \wedge i \leq n \wedge n - i = c_0 \wedge x[i] \mid k \wedge n - i = c_0\}\}\ count := count + 1\ \{\{Q''\}\}$ and
D. $\{\{Inv \wedge i \leq n \wedge n - i = c_0 \wedge x[i] \nmid k \wedge n - i = c_0\}\}$ **skip** $\{\{Q''\}\}$

Let us prove the conditions separately:

A. $Inv \wedge i \leq n \wedge n - i = c_0 \implies (x[i] \mid k \vee x[i] \nmid k)$
For each state of the statespace for which $Inv \wedge i \leq n \wedge n - i = c_0$ holds, either $x[i]$ is a divisor of k or $x[i]$ is not a divisor of k.

B. $Inv \wedge i \leq n \wedge n - i = c_0 \subseteq \mathcal{D}_{x[i] \mid k} \cap \mathcal{D}_{x[i] \nmid k}$
In order to ensure that $x[i] \mid k$ and $x[i] \nmid k$ are well-defined functions, we have to take into account not only that the divisibility $x[i] \mid k$ can be answered only if $x[i]$ is not zero, but the index i has to be inside the bounds of the array x. More precisely, we want to prove that $Inv \wedge i \leq n \wedge n - i = c_0 \implies i \in [1..n] \wedge x[i] \neq 0 \wedge i \in [1..n] \wedge x[i] \neq 0$.
Although $i \in [1..n+1]$ (due to the invariant) and the loop condition $i \leq n$ together allow us to deduce that $i \in [1..n]$ holds, we cannot guarantee that each state of the statespace for which $Inv \wedge i \leq n \wedge n - i = c_0$ holds, $x[i]$ is not 0. The reason is, that there is no assumption for the elements of x, except that they are integer numbers. The case when $x[i] = 0$, is not excluded by any condition we know and are allowed to use. If $x[i]$ equals 0, the expressions $x[i] \mid k$ and $x[i] \nmid k$ have no defined value. The current condition cannot be proven. We provide the remaining part of the proof for the sake of completeness.

C. $\{\{Inv \wedge i \leq n \wedge n - i = c_0 \wedge x[i] \mid k \wedge n - i = c_0\}\}\ count := count + 1\ \{\{Q''\}\}$
Let us recall that Q'' is $(Inv^{i \leftarrow i+1} \wedge n - i = c_0)$.
$Q''^{count \leftarrow count+1} = (Pre \wedge i + 1 \in [1..n+1] \wedge count + 1 = \sum_{j=1}^{i-1} \chi(x[j] \mid k)) + \chi(x[i] \mid k))$. By Theorem 2. it is sufficient to prove that
$(Inv \wedge i \leq n \wedge n - i = c_0 \wedge x[i] \mid k \wedge n - i = c_0) \implies Q''^{count \leftarrow count+1}$.

- Pre
  Pre in included in Inv.
- $i+1 \in [1..n+1]$
  Due to the invariant $i \in [1..n+1]$ holds. This, combined with the loop condition $i \leq n$ implies $i+1 \in [1..n+1]$. $i=1$ and $i \leq n$.
- $count + 1 = \sum_{j=1}^{i-1} \chi(x[j] \mid k)) + \chi(x[i] \mid k))$
  By the loop invariant Inv, $count = \sum_{j=1}^{i-1} \chi(x[j] \mid k))$. Since in this case $x[i]$ is a divisor of k, $\chi(x[i] \mid k) = 1$, we added 1 to both sides of the previous equation.

D. $\{\{Inv \wedge i \leq n \wedge n-i = c_0 \wedge x[i] \nmid k \wedge n-i = c_0\}\}$ **skip** $\{\{Q''\}\}$
Let us recall that Q'' is $(Inv^{i \leftarrow i+1} \wedge n-i = c_0)$ that is $(Pre \wedge i+1 \in [1..n+1] \wedge count = \sum_{j=1}^{i-1} \chi(x[j] \mid k)) + \chi(x[i] \mid k) \wedge n-i = c_0)$. By Theorem 1. it is sufficient to prove that
$(Inv \wedge i \leq n \wedge n-i = c_0 \wedge x[i] \nmid k \wedge n-i = c_0) \implies Q''$.

- Pre
  Pre in included in Inv.
- $i+1 \in [1..n+1]$
  We prove this in the same way as we did in the previous case.
- $count = \sum_{j=1}^{i-1} \chi(x[j] \mid k)) + \chi(x[i] \mid k))$
  By the loop invariant Inv, $count = \sum_{j=1}^{i-1} \chi(x[j] \mid k))$. Since in this case $x[j]$ is not a divisor of k, $\chi(x[i] \mid k)$ evaluates to zero. The desired condition holds because both sides of the equation $count = \sum_{j=1}^{i-1} \chi(x[j] \mid k))$ in Inv remained the same.
- $n-i = c_0$
  It is obviously true, since it is on the left side.

ii. $\{\{Q''\}\}\ i := i+1\ \{\{Inv \wedge n-i < c_0\}\}$
$Q'' = (Inv^{i \leftarrow i+1} \wedge n-i = c_0$. It is obvious that $(Inv^{i \leftarrow i+1} \wedge n-i = c_0) \implies (Inv \wedge n-i < c_0)^{i \leftarrow i+1}$, therefore by Theorem 2 we get the expected correctness formula.

To prove the correctness formula $\{\{Q\}\}\ S\ \{\{R\}\}$, all the verification conditions generated by the verification rules have to be satisfied. Let us remember that the following condition was not proven:
$Inv \wedge i \leq n \wedge n-i = c_0 \subseteq \mathcal{D}_{x[i]|k} \cap \mathcal{D}_{x[i]\nmid k}$
We could not guarantee that both of the logical functions $x[i] \mid k$ and $x[i] \nmid k$ are well-defined functions. Evaluating these functions of the program might lead to abortion. The rest of the conditions were unnecessary to prove, their proof was given for the sake of completeness.

5 Summarization

The main idea behind this work is to take into account abortion caused by partial functions in programs and extend verification rules to be able to ensure that such programs are total correct. To reason about correctness, we provided the formal definition of the semantics of programs under our investigation. One of the contributions of this paper is, that the semantics of program constructs are defined in such a way that they suit an existing relational model of programming. This relational model defines the program as a set of its possible executions and also provides definition for other important programming notions like problem and solution. Then, for each class of statements we provide a verification rule. The first six rules are well-known. Three rules are new, they are presented along with their proofs. The use of the rules is demonstrated by means of a verification case study.

References

[Apt97] K. R. Apt, E.-R. Olderog. *Verification of Sequential and Concurrent Program.* Springer-Verlag, 1997.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New York, 1976.

[Fot88] Á. Fóthi. *Mathematical Approach to Programming.* Ann. Univ. Sci. Budapest. Sect. Comput. 9, 105-114. 1988.

[Fot95] Á. Fóthi et al. *Some concepts of a Relational Model of Programming.* Varga L., ed., Proc. 4th Symposium on Programming Language and Software Tools, 434-446, Visegrád, Hungary, June 8-14, 1995.

[Fot05] Á. Fóthi. *Bevezetés a programozáshoz.* ELTE Eötvös Kiadó. 2005. (in Hungarian).

[Gre12] T. Gregorics. *Concept of abstract program.* Acta Universitatis Sapientiae, Informatica, 4, 1, 7-16. 2012.

[Gri81] D. Gries. *The Science of Programming.* Springer, Berlin, 1981.

[Hoa69] C. A. Hoare. *An axiomatic basis for computer programming.* Comm. ACM 12, pp. 576-580, 1969.

[Man74] Z. Manna. *Mathematical theory of computation.* McGraw Hill, 1974.

[Owi76] S. Owicki, D. Gries. *An axiomatic proof technique for parallel programs.* Acta Inf., 6, pp. 319-340, 1976.

[Roe01] W.-P. de Roever et al. *Concurrency Verification.* Cambridge University Press, 2001.