UDC 004.4:510.6
DOI: https://doi.org/10.17721/1812-5409.2026/1.29

Andrii KRYVOLAP, PhD (Phys. & Math.)
ORCID ID: 0009-0002-0493-9587
e-mail: andriikryvolap@knu.ua
Taras Shevchenko National University of Kyiv, Kyiv, Ukraine

Olena SHYSHATSKA, PhD (Phys. & Math.)
ORCID ID: 0000-0001-8791-8989
e-mail: shyshatska@knu.ua
Taras Shevchenko National University of Kyiv, Kyiv, Ukraine

Nataliia RUSINA, PhD (Ped.), Assoc. Prof.
ORCID ID: 0000-0002-5595-9548
e-mail: rusina@knu.ua
Taras Shevchenko National University of Kyiv, Kyiv, Ukraine

FRAME RULE IN MONOTONE FLOYD – HOARE LOGIC EXTENSION

Separation logic is an extension of Floyd – Hoare logic, aimed at the specification and verification of programs with a main focus on the representation of dynamic memory handling. Its main tool is the frame rule, which enables local proofs. Applying the composition-nominative approach to classical Floyd – Hoare logic, its monotone extension was developed. The main advantages it offers are the use of nominative data to describe the state of the program and the consideration of partial predicates. The main purpose of this research is to introduce the frame rule and pointer capabilities for monotone Floyd – Hoare logic. This further extension adds support for local reasoning, which is crucial today when the complexity of software only rises. Local reasoning means that a property of a part of the program can be proved separately and combined with proofs for other components if specific conditions hold. For this purpose, the nominative data was enriched with the notion of an address. New special compositions for storing and loading from dynamic memory were introduced. The logical connective separating conjunction was redefined as a composition in program algebra. This composition was proved to be monotone. To extend the proof of the program's approximations to a complete proof, all compositions have to be monotone. Furthermore, rules for new operators were added to the derivation system. Rules for storing to and loading from dynamic memory are modifications of the rules for the assignment operator. The frame rule states that, with the help of a separating conjunction, a valid Hoare triple can be augmented with an assertion for a part of the memory that was not modified by the program. It was shown that the classic version of the rule, implemented within Separation logic, is not sound for a monotone extension. Conditions for a sound version were presented. Investigation of possible restrictions on the operations with pointers, which will lead to constructive frame rule conditions, is the aim of future research. Complex nominative data with dynamic memory is out of the scope of this paper and should be considered separately.

K e y w o r d s: ***formal methods, software verification, program logic, nominative data, separation logic.***

AMS 2020 classification: 03B70, 68Q60.

Introduction

Relevance of research. Software verification is a crucial part of software engineering as its main goal is ensuring the correctness and quality of program systems. The main challenge within this domain remains the availability of efficient tools supported by formal methods with sufficient expressiveness. Floyd – Hoare logic, after being introduced, became a standard basis for formal verification. Different extensions were studied over the years (Apt, & Olderog, 2019), covering, among others, parallel programs, complex data structures, etc. One such extension is the Separation logic. The main advantages of the suggested approach were the operators introduced to handle assertions over dynamic memory and a frame rule aimed at local reasoning. The importance of local reasoning is reinforced by the growing complexity of modern software. Further development resulted in the introduction of numerous variants of the Separation logic (Reynolds, 2002), covering different aspects of program systems. Among such variants of significant interest are the following:

- Separation Incorrectness logic (Raad et al., 2020), which enriches the Incorrectness logic with tools for handling dynamic memory. Incorrectness logic is a dual approach to Hoare logic, aimed at finding errors rather than proving correctness in all states.
- Exact Separation Logic (Maksimović et al., 2023), which combines in a single formal system over-approximation of Separation logic, and under-approximation of Separation Incorrectness logic.
- Concurrent Separation Logic (O'Hearn, 2007), where disjoint memory areas are treated more in terms of parallel programs, and communication is performed with the help of common memory.
- Dynamic Separation logic (De Boer, Hiep, & de Gouw, 2023), which combines Separation logic with dynamic logic, that is, an approach of defining Floyd – Hoare logic as modal logic.

Monotone Floyd – Hoare logic extension (Nikitchenko et al., 2018) uses nominative data and partial mappings to properly describe the state of execution of a program and account for a possible non-terminating behavior. It is defined within the composition-nominative approach (Shkilnyak, 2025), which also allows definitions in a semantic-syntactical manner. Different types of complex nominative data were broadly studied for the monotone Floyd – Hoare logic extension, also covering such problems as the unambiguity of the complex names for such data. But at the same time, pointers and dynamic

memory pose different challenges, such as aliasing, pointer arithmetics, and proving the exclusiveness of the memory access. To handle such problems, notions of dynamic memory and disjoint areas of dynamic memory need to be adopted for the monotone Floyd – Hoare logic extension. Combined with the frame rule variant, this would allow a program logic that will have local reasoning and nominative data capabilities, together with monotone operators.

Lately, there has been a rise in interest in an under-approximating approach to program logic that is dual to an over-approximating approach of the Floyd – Hoare logic. Logics that follow such an approach are aimed at finding the execution that leads to an error, rather than at proving that executions starting in a proper initial state will be correct. Examples of such logics are the Incorrectness logic (O'Hearn, 2019) and its Separation version (Raad et al., 2020). Even though under-approximation is not considered for the monotone Floyd – Hoare logic extension, it is a topic for future investigation. This proves that the frame rule and dynamic memory handling can be beneficial for a broad range of program logics.

The object of research is program logics that allow reasoning about the dynamic memory aspect of the programs and complex data structures.

The aim and objectives of the research are to introduce a nominative data extension with pointers and dynamic memory, provide the implementation of the frame rule of Separation logic within monotone Floyd – Hoare logic, and investigate conditions for its soundness.

1. Main definitions and results

Monotone Floyd – Hoare logic is defined with a semantic-syntactic approach. This means that first semantics is introduced as the following three-sorted program algebra:

$$< Pr^{V,A}, Fn^{V,A}, FPrg^{V,A}; \vee, \neg, \{S_P^{\overline{v}}\}_{\overline{v}\in U(V)}, \{S_F^{\overline{v}}\}_{\overline{v}\in U(V)}, \{`x\}_{x\in V}, \{\exists x\}_{x\in V}, id, \{AS^x\}_{x\in V}, \bullet, IF, WH, FH > .$$

Here $Pr^{V,A} : ND(V,A) \xrightarrow{\mathsf{p}} Bool$ is a set of all partial mappings from nominative data $ND(V,A)$ to a set of boolean values $Bool = \{True, False\}$ that represent semantics of boolean expressions in the program, $Fn^{V,A} : ND(V,A) \xrightarrow{\mathsf{p}} A$ is a set of all partial mappings from nominative data $ND(V,A)$ to a set of values A that represent semantics of arithmetic expressions in the program, $FPrg^{V,A} : ND(V,A) \xrightarrow{\mathsf{p}} ND(V,A)$ is a set of all partial mappings from nominative data $ND(V,A)$ to nominative data $ND(V,A)$ that represent the semantics of program operators where nominative data models the current state of the variables. At the same time, nominative data is defined as the set of all partial mappings from the set of the names of the variables V to a set of all possible values A: $ND(V,A) : V \xrightarrow{\mathsf{p}} A$.

Operators of the algebra are called compositions and denote the main constructs of the programming language. Superposition operators $\{S_P^{\overline{v}}\}_{\overline{v}\in U(V)}, \{S_F^{\overline{v}}\}_{\overline{v}\in U(V)}$ with parameter $\overline{v} \in U(V)$ correspond to the function or predicate calls, where $U(V)$ is the set of all finite sequences of different variable names. $\{`x\}_{x\in V}$ is a parametric denaming operator used to retrieve the value of a variable. Compositions $\vee, \neg, \{\exists x\}_{x\in V}$ represent respective logical connectives. On the program level, compositions $id, \{AS^x\}_{x\in V}, \bullet, IF, WH$ are used to denote program constructs skip (idle operator), assignment operator, sequential execution, if-then-else operator, and while-do operator accordingly. FH is a special composition representing the semantics of the Floyd – Hoare triple.

Nominative data has to be extended to allow reasoning about dynamic memory or the heap. We will use the same assumptions as in classical Separation logic, namely: the set of all possible values is the same set that is used as addresses for pointers; variables can be assigned either a value or an address. This leads us to the following definition.

Definition 1. Dynamic nominative data over a set of variable names V and a set of addresses A is a partial mapping of the type: $DND(V,A) : V \cup A \xrightarrow{\mathsf{p}} A$.

For further convenience, the next notations will be used to denote the stack and heap parts of the dynamic nominative data $st \in DND(V,A)$: $stack(st) = st|_V$ and $heap(st) = st|_A$. Here $st|_V$ marks the restriction of the mapping st to a set V. As dynamic nominative data represents a state of the program, both terms will be further used interchangeably.

Substituting nominative data with dynamic nominative data, we can introduce dynamic versions of the partial mapping types: $Pr_D^{V,A} : DND(V,A) \xrightarrow{\mathsf{p}} Bool$, $Fn_D^{V,A} : DND(V,A) \xrightarrow{\mathsf{p}} A$, $FPrg_D^{V,A} : DND(V,A) \xrightarrow{\mathsf{p}} DND(V,A)$.

Extension of most program algebra compositions on dynamic nominative data is straightforward and is omitted in this paper. Only superpositions require consideration. To describe operations with a heap, new compositions of storing values to and retrieving values from a specific address are introduced. We assume that to operate with an address, it has to be first stored in some variable. This won't limit the expressiveness of a programming language, but it will help to make address arithmetic expressions clearer. Also, as in the case of nominative data, only equitone predicates and functions are considered for dynamic nominative data. Mapping f over dynamic nominative data is considered equitone if for any state st, $f(st)\downarrow \Rightarrow \forall st' \supseteq st : f(st')\downarrow = f(st)$. Here, $f(st)\downarrow$ denotes that mapping f is defined on state st; $f(st)\uparrow$ will be used to indicate that mapping f is undefined on state st. Restriction of $Pr_D^{V,A}$ and $Fn_D^{V,A}$ to only equitone mapping is natural, as it corresponds to the intuition that if information within a current state is sufficient to calculate a value of the expression of a condition, this value should not change for any extension of the state.

Definition 2. Parametric store composition ST^x of a type $Fn_D^{V,A} \xrightarrow{\mathsf{t}} FPrg_D^{V,A}$ with a parameter $x \in V$ can be defined on arbitrary state as $ST^x(f)(st) \simeq st\nabla[st(x) \mapsto f(st)]$. Here $[v_1 \mapsto a_1, \ldots, v_n \mapsto a_n, a'_1 \mapsto a''_1, \ldots, a'_m \mapsto a''_m]$ is an explicit form of dynamic nominative data representation for $v_1, \ldots, v_n \in V$ and $a_1, \ldots a_n, a'_1, \ldots, a'_m, a''_1 \ldots a''_m \in A$, where each pair denotes either name-value or address-value in data. Overriding operator ∇ is defined as follows: $st_1\nabla st_2 = [l \mapsto a | l \in$ $\in$ dom $st_2 \wedge st_2(l) = a \vee l \in$ dom $st_1 \setminus$ dom $st_2 \wedge st_1(l) = a]$. In other words, $st = st_1\nabla st_2$, the result of the application of

the overriding operator will be the state (nominative data) in which variable and address mapping that was defined in the right operand st_2 is retained together with all pairs from the left operator st_1, that are not defined in st_2.

Definition 3. Parametric load function LD^x of a type $Fn_D^{V,A}$ with a parameter $x \in V$ can be defined on an arbitrary state in the following way: $LD^x(st) \simeq st(st(x))$. Applying nominative data st to a variable x will yield the address stored in it, but the second application will return the value stored at that address.

Definition 4. Parametric extended predicate superposition composition $\{S_P^{\overline{v},\overline{u'}}\}_{\overline{v},\overline{u}\in U(V)}$ of a type $Pr_D^{V,A} \times (Fn_D^{V,A})^{n+m} \xrightarrow{t} \xrightarrow{t} Pr_D^{V,A}$ with parameters $v_1, v_2, \ldots, v_n, u_1, u_2, \ldots, u_m \in V$ can be defined on arbitrary state in the following way:

$$S_P^{\overline{v},\overline{u'}}(p, f_1, \ldots, f_n, g_1, \ldots, g_m)(st) \simeq p(st\nabla[v_1 \mapsto f_1(st), \ldots, v_n \mapsto f_n(st), st(u_1) \mapsto g_1(st), \ldots, st(u_m) \mapsto g_m(st)]).$$

A common type of property that needs to be verified for programs that manipulate the heap deals with exclusive access of certain program components to their own part of the heap. Separation conjunction is a logical connective introduced in Separation logic to handle such assertions without the need for repeated explicit conditions describing parts of the heap each component works with. Intuitively, it can be defined in the following way: formula $P * Q$ holds for each configuration where there are two disjoint subheaps, on one of which P holds and on another – Q holds. Let us give a formal definition of two states having disjoint heaps with a shared stack.

Definition 5. Two states $st_1, st_2 \in DND(V, A)$ are considered disjoint configurations $st_1||_{V,A}st_2$ if $stack(st_2) = stack(st_2)$ and dom $heap(st_1) \cap$ dom $heap(st_2) = \emptyset$

If two states are disjoint, the overall state can be obtained by applying the override operator. Disjoint union will be used as a name for the resulting operation.

Definition 6. State $st \in DND(V, A)$ is called a disjoint union of two states $st_1, st_2 \in DND(V, A)$ (denoted $st = st_1 \oplus st_2$) if $st_1||_{V,A}st_2$ and $st = st_1\nabla st_2$.

Based on the definition, the next properties of disjoint union can be obtained: $st_1 \oplus st_2 = st_2 \oplus st_1$, $stack(st_1 \oplus st_2) = = stack(st_1) = stack(st_2)$, $heap(st_1 \oplus st_2) = heap(st_1) \cup heap(st_2)$. Additionally, if $st = st_1 \oplus st_2$ we will use $\ominus$ to mark the remainder of the heap: $st_1 = st \ominus st_2$ and $st_2 = st \ominus st_1$.

With a formal means to describe states with the same stack but non-intersecting areas of dynamic memory, separation conjunction composition can be introduced for program algebra monotone Floyd – Hoare logic extension with heap. For convenience, it will be used in infix form.

Definition 7. Separation conjunction composition $*$ of a type $Pr_D^{V,A} \times Pr_D^{V,A} \xrightarrow{t} Pr_D^{V,A}$ can be defined on the arbitrary state in the following way:

$$P * Q(st) = \begin{cases} True, & \text{iff exist } st_1, st_2 \text{ such that } st = st_1 \oplus st_2 \wedge P(st_1) = True \wedge Q(st_2) = True; \\ False, & \text{iff for every } st_1, st_2 \text{ such that } st = st_1 \oplus st_2 \text{ it holds that } P(st_1) = False \vee Q(st_2) = False; \\ \bot, & \text{otherwise.} \end{cases}$$

Let us recall the definition of the monotone operator prior to presenting the results on whether the separation conjunction composition is monotone. Monotonicity is treated in the sense of definedness.

Definition 8. A composition $C : (Pr_D^{V,A})^k \times (Fn_D^{V,A})^l \times (FPrg_D^{V,A})^m \xrightarrow{t} Pr_D^{V,A}$ is called monotone over the first argument if for any $p, p_2, \ldots, p_k, p' \in Pr_D^{V,A}$, $f_1, f_2, \ldots, f_l \in Fn_D^{V,A}$, and $pr_1, pr_2, \ldots, pr_m \in FPrg_D^{V,A}$ it holds that $p \subseteq p'$ implies $C(p, p_2, \ldots, p_k, f_1, f_2, \ldots, f_l, pr_1, pr_2, \ldots, pr_m) \subseteq C(p', p_2, \ldots, p_k, f_1, f_2, \ldots, f_l, pr_1, pr_2, \ldots, pr_m)$. Here $\subseteq$ is considered for mappings as a set-theoretical relation over the graphs of respective mappings. A composition is called monotone if it is monotone over every argument.

Lemma 1. *Separation conjunction composition is monotone.*

Proof. Monotonicity will be proved for the first argument of composition. Proof for another argument is similar.

Assume that property doesn't hold, this means that $\exists P, P', Q \in Pr_D^{V,A}$ such that $P \subseteq P' \wedge P * Q \nsubseteq P' * Q$. The latter is possible in the following cases:

- $\exists st \in DND : P * Q(st) \downarrow = True \wedge (P' * Q(st) \downarrow = False \vee P' * Q(st) \uparrow)$,
- $\exists st \in DND : P * Q(st) \downarrow = False \wedge (P' * Q(st) \downarrow = True \vee P' * Q(st) \uparrow)$.

Let us consider the first case. If $P * Q(st) \downarrow = True$ then by the definition of composition there should exist such st_1 and st_2 that $P(st_1) = True$ and $Q(st_2) = True$ and $st = st_1 \oplus st_2$. From $P(st_1) = True$ and $P \subseteq P'$ we obtain $P'(st_1) = True$ and from definition of composition using $P'(st_1) = True$, $Q(st_2) = True$ and $st = st_1 \oplus st_2$ we can conclude that $P' * Q(st) \downarrow = True = P * Q(st)$ which contradicts with our assumption.

The second case, where $P * Q(st) \downarrow = False$, proof is similar. Using the definition of composition for every st_1 and st_2 such that $st = st_1 \oplus st_2$, either $P(st_1) = False$ or $Q(st_2) = False$. If $P(st_1) = False$, using $P \subseteq P'$ we obtain

$P'(st_1) = False$. Finally, this proves we cannot find a pair of disjoint states such that at least one of $P'(st_1) = False$ and $Q(st_2) = False$ won't hold. Thus, $P' * Q(st) \downarrow = False = P * Q(st)$, and in the second case, we also achieve a contradiction.

Both cases were considered, which means that $P*Q \subseteq P'*Q$ and composition is monotone over the first argument. □

Combining the given definition, we will obtain a three-sorted algebra that will define the semantics of the monotone Floyd – Hoare logic extension with dynamic memory.

$$< Pr_D^{V,A}, Fn_D^{V,A}, FPrg_D^{V,A}; \vee, \neg, *, \{S_P^{\overline{v},\overline{u'}}\}_{\overline{v},\overline{u}\in U(V)}, \{S_F^{\overline{v},\overline{u'}}\}_{\overline{v},\overline{u}\in U(V)}, \{`x\}_{x\in V},$$
$$\{\exists x\}_{x\in V}, id, \{AS^x\}_{x\in V}, \{ST^x\}_{x\in V}, \{LD^x\}_{x\in V}, \bullet, IF, WH, FH > .$$

The next step, after new compositions were defined and their properties were investigated, is to provide sound derivation rules for a monotone Floyd – Hoare logic extension that operates with those compositions. Proof that existing rules remained sound when moving from nominative data to dynamic nominative data is straightforward and also omitted in this paper.

The rule for store composition can be viewed as the modification of the rule for assignment composition:

$$R\_ST \frac{}{\{S_P^{x'}(p,f)\}ST^x(f)\{p\}}.$$

Lemma 2. *Rule R_ST is sound.*

Proof. The rule is sound if for valid premises – valid outcomes are derived. As in the rule, there are no premises – we need to prove that triple $\{S_P^{x'}(p,f)\}ST^x(f)\{p\}$ is valid. Triple $\{p\}r\{q\}$ is considered valid if for every interpretation and for every state st, such that $p(st) \downarrow = True$, either $r(st) \uparrow$ or $q(r(st)) \downarrow = True$. In our case, based on the definition of superposition composition, if $S_P^{x'}(p,f)(st) \downarrow = True$ then $p(st\nabla[st(x) \mapsto f(st)]) \downarrow = True$. If program $ST^x(f)$ terminates for starting state st, $p(ST^x(f)(st)) = p(st\nabla[st(x) \mapsto f(st)])$, which evaluates to True based on the previous assertion. This proves the soundness of the rule. □

In the Separation logic frame, the rule for the derivation system is defined as follows:

$$R\_FRAME\_SEP \frac{\{p\}pr\{r\}}{\{p*q\}pr\{r*q\}}, ch(pr) \cap fv(q) = \emptyset.$$

Where $ch(pr)$ denotes variables that were changed by program pr and is defined inductively based on the structure of pr, $fv(q)$ is the list of free variables for predicate q. Such a variant of the rule is not sound for monotone Floyd – Hoare logic extension. The reason behind this fact is that the introduced condition doesn't account for complex pointer arithmetic available in the general case. We will investigate restrictions in the semantic form that will address this issue. The main idea behind condition $ch(pr) \cap fv(q) = \emptyset$, together with the use of the separation conjunction, was to describe that when there is an assertion about the part of the heap that the program doesn't affect – it can be combined with any valid assertion about the part of the heap that the program modifies.

Let us introduce tools to describe such aspects of the predicates and programs as the part of the heap affected by them. As pointer arithmetic is not limited in our case, part of the heap affected by the program in terms of updating stored values is a function over the state. It cannot be described by only considering the variables that the program updates. In the general case, the definition of the required function can be seen below.

Definition 9. Memory footprint function for a program $pr \in FPrg_D^{V,A}$ is a function of a type $DND(V,A) \xrightarrow{t} 2^A$ that for an arbitrary state st where $pr(st) \downarrow$ is equal $mem(pr)(st) = \{a | a \in A \wedge st(a) \downarrow \wedge pr(st)(a) \downarrow \neq st(a)\}$, and $mem(pr)(st) = \emptyset$ otherwise if program is not terminating on st.

The intuition behind the memory footprint function is the maximal heap area affected and changed by the program. It doesn't account for addresses that were updated in the process with the same value. In this case, a strict version of the function can be introduced that considers not only the addresses that were changed by the program but also the addresses that are required by the program to terminate.

Definition 10. A strict memory footprint function for a program $pr \in FPrg_D^{V,A}$ is such a function of a type $DND(V,A) \xrightarrow{t} 2^A$ that for an arbitrary state st, where $pr(st) \downarrow$, it can be defined as $mem_s(pr)(st) = \bigcup\limits_i A_i$, where A_i are all such sets of addresses that $retain(pr, st, A_i) \wedge \neg\exists A_i' \subset A_i : retain(pr, st, A_i')$ and $retain(pr, st, A_i)$ is a special notion for $pr(st)|_{V\cup A_i} = pr(st|_{V\cup A_i})|_{V\cup A_i} \wedge heap(st|_{V\cup(A\setminus A_i)}) = heap(pr(st)|_{V\cup(A\setminus A_i)})$, and $mem_s(pr)(st) = \emptyset$ if program is not terminating on st.

Remark 1. Based on the monotonicity of compositions it can be shown that the condition for A_i also holds for their union and any superset up to st, which means $retain(pr, st, mem_s(pr)st)$ and for all A such that $mem_s(pr)st \subset A \subset$ dom $heap(st)$ it holds that $retain(pr, st, A)$.

Remark 2. It is obvious that for arbitrary $st \in DND(V,A)$ and arbitrary $pr \in Prg_D^{V,A}$ $mem(pr)(st) \subseteq mem_s(pr)(st)$.

Restrictions on the memory modification capabilities for programs that are available within program algebra impact whether the memory footprint function can be defined inductively based on the program structure.

Similarly to the programs, a general definition of a memory function for predicates can be given based on the intuitive notion of an area of memory required for the predicate to be defined.

Definition 11. Memory footprint function for a predicate $p \in Pr_D^{V,A}$ is a function of a type $DND(V,A) \xrightarrow{t} 2^A$ that for an arbitrary state st where $p(st) \downarrow$ it is equal $mem_s(p)(st) = \bigcup\limits_i A_i$, here A_i are all such sets of addresses such that $p(st|_{V \cup A_i}) \downarrow = p(st) \wedge \neg \exists A_i' \subset A_i : p(st|_{V \cup A_i}) \downarrow = p(st)$, and $mem(p)(st) = \emptyset$ otherwise if predicate is not defined on st.

Using the notion of the memory footprint frame rule, modified semantical conditions can be introduced:

$$R\_FRAME \frac{\{p\}pr\{r\}}{\{p * q\}pr\{r * q\}}, \forall st \in DND(V,A) : p(st) \to (mem_s(pr)(st) \cap mem(q)(st) = \emptyset), ch(pr) \cap fv(q) = \emptyset.$$

Theorem 1. *Rule R_FRAME is sound.*

Proof. Assume that triple $\{p * q\}pr\{r * q\}$ is not valid. This means that there exists such a state st that $p * q(st) = True$, $pr(st)$ is defined, and $r * q(pr(st)) = False$. If $p * q(st) = True$ then from the definition of separating conjunction exist st_1 and st_2 such that $st = st_1 \oplus st_2$, $p(st_1) = True$, $q(st_2) = True$.

Consider $st_2' = st_2|_{V \cup mem(q)(st)}$, in this case we can construct $st_1' \supseteq st_1$ such that $st_1' = st \ominus st_2'$. This state will extend the heap of st_1 with the part of the heap of st_2, not in st_2'. Based on the properties of compositions of the predicate level $p(st_1') = True$. $q(st_2') = True$ holds based on the definition of memory footprint of a predicate if we show that $st_2|_{V \cup mem(q)(st)} = st_2|_{V \cup mem(q)(st_2)}$. The latter is equal to $mem(q)(st) \cap$ dom $heap(st_2) = mem(q)(st_2)$ as heap part of the state can be restricted only up to the list of its addresses. This statement is true if we consider that $q(st_2) = True$, $st_2 = st|_{V \cup \text{dom } heap(st_2)}$, $q(st) = True$, and the definition of the memory footprint.

Next we shall prove that for any states st and st_1'' such that $st_1'' \subseteq st$, $stack(st) = stack(st_1'')$, $pr(st) \downarrow$ and also dom $heap(st_1'') \cap mem_s(pr)(st) = \emptyset$ there exist the states st_2'', st_1''', such that $st = st_1'' \oplus st_2''$, $pr(st) = pr(st_2'') \oplus st_1'''$, $stack(st_1''') = stack(pr(st))$ and $heap(st_1''') = heap(st_1'')$. In this case, $st_2'' \supseteq st|_{V \cup mem_s(pr)(st)}$, which means that conditions from the definition of the strict memory footprint also hold for st_2'' and $pr(st) = pr(st_2'') \oplus st_1'''$.

It is easy to see that if $st_1'' = st_2'$ then all the required conditions hold and $st_2'' = st_1'$ as $st = st_1' \oplus st_2'$. Thus, $pr(st) = pr(st_1') \oplus st_1'''$, $stack(st_1''') = stack(pr(st))$, and $heap(st_1''') = heap(st_2')$. Based on the definition of $ch(pr)$, $stack(st_1''')|_{V \setminus ch(pr)} = stack(st)|_{V \setminus ch(pr)} = stack(st_2')|_{V \setminus ch(pr)}$. As $ch(pr) \cap fv(q) = \emptyset$, we have $fv(q) \subseteq V \setminus ch(pr)$, together with $heap(st_1''') = heap(st_2')$ and $stack(st_1''')|_{V \setminus ch(pr)} = stack(st_2')|_{V \setminus ch(pr)}$ this gives us $q(st_1''') = q(st_2') = True$. To show that $r(pr(st_1')) = True$ we need to use the fact that $\{p\}pr\{r\}$ holds for any state, and for state st_1', $p(st_1') = True$ and $pr(st_1') \downarrow$. From $pr(st) = pr(st_1') \oplus st_1'''$, $q(st_1''') = True$ and $r(pr(st_1')) = True$ we can obtain $r * q(pr(st)) = True$, this contradicts with assumption, which proves that rule R_FRAME is sound. □

2. Examples

To showcase the frame rule for monotone Floyd – Hoare logic extension, we will use a simple example with a specification of the stack data structure. This is a first-in-last-out data structure, assertions for two main operations – put a value on top of the stack and retrieve a value from the top of the stack. Memory allocation and deallocation are not considered in this example. The second stack and its properties are used to illustrate the frame rule and give another example of assertions about data structures.

Example. Without any restriction on generality, we will use the set of lowercase letters of the Roman alphabet as variable names $V = \{a, b, \dots, z\}$ and integer numbers as addresses $A = Int = \{\dots, -1, 0, 1, \dots\}$. Basic predicates and functions will include parametric predicates for relations greater, equals, and less, and parametric functions increment and decrement, denoted respectively $>_{x,y}$, $=_{x,y}$, $<_{x,y}$, $x++$, and $x--$. x and y are names of the variables that are used in mappings. The stack data structure is presented by its top and bottom addresses; all addresses that are greater than the bottom and less than the top are considered a part of the stack. To put a value on top of the stack – value is stored in the top address, and the variable holding the top is incremented. Retrieving value is exactly opposite – if top is greater than bottom, return the value at the top and decrement top. The stack is empty if the top address is equal to the bottom address. A stack is considered invalid if its top address is less than the bottom address.

The subprograms and predicates described above can be defined semantically with the first parameter being a variable storing the top address, and the second parameter being a variable storing the bottom address. $push_{x,y}(f) \simeq ST^x(f) \bullet$ $\bullet\ AS^x(x++)$ pushes the value of the expression that represents function f to the top of the stack. $pop_{x,y,v} \simeq IF(>_{x,y}, AS^v(LD^x) \bullet AS^x(x--), id)$ pops the top value of the stack to the variable v; if the stack is empty or invalid, it does nothing. $empty_{x,y} \simeq =_{x,y}$, $invalid_{x,y} \simeq <_{x,y}$.

Here are the properties of the stack data structure that can be proved: $\{empty_{x,y}\}push_{x,y}(f)\{\neg empty_{x,y}\}$, $\{empty_{x,y} \wedge$ $\wedge\ =_{v,w}\}pop_{x,y,v}\{empty_{x,y} \wedge\ =_{v,w}\}$ and $\{empty_{x,y}\}push_{x,y}(f) \bullet pop_{x,y,v}\{empty_{x,y} \wedge S^w(=_{v,w}, f)\}$. It is easy to define memory footprints for $push$ and pop: $mem_s(push_{x,y}(f))(st) = \{st(x--(st))\}$, $mem_s(pop_{x,y}(f))(st) = \{st(x++(st))\}$. Nominative data $[x \mapsto 3, y \mapsto 1, 1 \mapsto 4, 2 \mapsto 2]$ representing a simple stack with two values in it.

If we have another stack with top a and bottom b condition that this stack contains values in the ascending order can be defined as $q = \forall e \forall f(>_{e,f} \wedge \geqslant_{a,e} \wedge \geqslant_{f,b} \to S^{x,y}(>_{x,y}, LD^e, LD^f)$. The frame rule cannot be used to prove that

$\{empty_{x,y} * q\} push_{x,y}(f) \bullet pop_{x,y,v}\{(empty_{x,y} \wedge S^w(=_{v,w}, f)) * q\}$. The memory footprint of a predicate q is $mem(q)(st) = = \{i \in Int | b'(st) \leqslant i \leqslant a'(st)\}$. There is no guarantee that operations on the initial stack won't overwrite values for addresses of the second stack. One example is $[x \mapsto 3, y \mapsto 1, 1 \mapsto 4, 2 \mapsto 2, a \mapsto 5, b \mapsto 3, 3 \mapsto 1, 4 \mapsto 5]$. This issue can be handled by adding to the precondition assertion that the top address of the first stack is less than the bottom address of the second, or vice versa.

The final Floyd – Hoare triple will be $\{(empty_{x,y} \wedge >_{b,x}) * q\} push_{x,y}(f) \bullet pop_{x,y,v}\{(empty_{x,y} \wedge S^w(=_{v,w}, f)) * q\}$. Shared variable values within the separating conjunction and the absence of the restriction on the free variables of p in R_FRAME made it possible to specify the needed condition. If we have another stack with top a and bottom b condition that the stack contains values in ascending order can be defined as $q = \forall e \forall f (>_{e,f} \wedge \geqslant_{a,e} \wedge \geqslant_{f,b} \rightarrow S^{x,y}(>_{x,y}, LD^e, LD^f)$. Frame rule cannot be used to prove $\{empty_{x,y} * q\} push_{x,y}(f) \bullet pop_{x,y,v}\{(empty_{x,y} \wedge S^w(=_{v,w}, f)) * q\}$. The memory footprint of the predicate q is $mem(q)(st) = \{i \in Int | b'(st) \leqslant i \leqslant a'(st)\}$. There is no guarantee that operations on the initial stack won't overwrite values for addresses of the second stack. The example for such a state is $[x \mapsto 3, y \mapsto 1, 1 \mapsto 4, 2 \mapsto 2, a \mapsto 5, b \mapsto 3, 3 \mapsto 1, 4 \mapsto 5]$. This issue can be handled by adding to the precondition assertion that the top address of the first stack is less than the bottom address of the second, or vice versa. The final Floyd – Hoare triple will be $\{(empty_{x,y} \wedge >_{b,x}) * q\}$ $push_{x,y}(f) \bullet pop_{x,y,v}\{(empty_{x,y} \wedge S^w(=_{v,w}, f)) * q\}$. Shared variable values within the separating conjunction and the absence of the restriction on the free variables of p in R_FRAME made it possible to specify the needed condition.

Discussion and conclusions

Dynamic nominative data was introduced, making it possible to define properties of the programs with pointers within the composition-nominative approach. To represent basic operations with the heap, new compositions were defined, and their monotonicity was proved. Similar results were achieved for the separation conjunction connective, which is the basis of the frame rule and assertions over the heap. Overall, the conditions of the frame rule were investigated in terms of monotone Floyd – Hoare logic extension, and a simple example of its application is provided. The updated conditions required new means that describe a part of the heap affected by the program and a part of the heap needed for the predicate to be defined. Basic properties of such memory mappings were presented.

Similar to what was achieved in already published papers, the introduction of the frame rule extended the capabilities of monotone Floyd – Hoare logic extension for local reasoning. But it is worth mentioning that the construction of memory footprints for special cases of logic, where address operations are restricted, is one of the directions of further research, together with a weaker version of the frame rule, which has a less complex condition. Other open questions are working with memory allocation and concurrency.

The general case of predicates over dynamic nominative data was considered. Further research is required to define the properties of predicates needed for efficient computation of the memory footprint of the predicate. This could also lead to the introduction of new logical connectives similar to the separating conjunction.

Authors' contribution: Olena Shyshatska — conceptualization; Nataliia Rusina — methodology; Andrii Kryvolap — analysis of sources, theoretical foundations of research.

Sources of funding. The study was partially financially supported by Taras Shevchenko National University of Kyiv.

References

Apt, K. R., & Olderog, E.-R. (2019). Fifty years of Hoare's logic. *Formal Aspects of Computing*, 31(6), 751–807. https://doi.org/10.1007/s00165-019-00501-3

De Boer, F. S., Hiep, H.-D. A., & de Gouw, S. (2023). Dynamic separation logic. *Electronic Notes in Theoretical Informatics and Computer Science*, 3, 5:1–5:19. https://doi.org/10.46298/entics.12297

Maksimović, P., Cronjäger, C., Lööw, A., Sutherland, J., & Gardner, P. (2023). Exact separation logic: Towards bridging the gap between verification and bug-finding. In Ali & Salvaneschi (Eds.), *37th European Conference on Object-Oriented Programming (ECOOP 2023)* (pp. 19:1–19:27). Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPICS.ECOOP.2023.19

Nikitchenko, M., Ivanov, I., Kornilowicz, A., & Kryvolap, A. (2018). Extended Floyd-Hoare logic over relational nominative data. *Communications in Computer and Information Science*, 826, 41–64. https://doi.org/10.1007/978-3-319-76168-8_3

O'Hearn, P. W. (2019). Incorrectness logic. *Proceedings of the ACM on Programming Languages*, 4(POPL), 1–32.https://doi.org/10.1145/3371078

O'Hearn, P. W. (2007). Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

Raad, A., Berdine, J., Dang, H.-H., Dreyer, D., O'Hearn, P., & Villard, J. (2020). Local reasoning about the presence of bugs: Incorrectness separation logic. In Lahiri & Wang (Eds.), *Computer Aided Verification* (pp. 225–252). Springer International Publishing.https://doi.org/10.1007/978-3-030-53291-8_14

Reynolds, J. (2002). Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science* (pp. 55–74). IEEE. https://doi.org/10.1109/LICS.2002.1029817

Shkilnyak, S. (2025). Varieties of pure first-order logics of partial quasiary predicates. *Bulletin of Taras Shevchenko National University of Kyiv. Physical and Mathematical Sciences*, 79(2), 80–88. https://doi.org/10.17721/1812-5409.2024/2.13

Отримано редакцією журналу / Received: 10.04.25
Прорецензовано / Revised: 06.11.25
Схвалено до друку / Accepted: 16.04.26
Опубліковано / Published: 05.06.26

Андрій КРИВОЛАП, канд. фіз.-мат. наук
ORCID ID: 0000-0003-4567-8901
e-mail: andriikryvolap@knu.ua
Київський національний університет імені Тараса Шевченка, Київ, Україна

Олена ШИШАЦЬКА, канд. фіз.-мат. наук
ORCID ID: 0000-0001-8791-8989
e-mail: shyshatska@knu.ua
Київський національний університет імені Тараса Шевченка, Київ, Україна

Наталія РУСІНА, канд. пед. наук, доц.
ORCID ID: 0000-0002-5595-9548
e-mail: rusina@knu.ua
Київський національний університет імені Тараса Шевченка, Київ, Україна

РАМКОВЕ ПРАВИЛО В МОНОТОННОМУ РОЗШИРЕННІ ЛОГІКИ ФЛОЙДА – ХОАРА

Логіка розділення є розширенням логіки Флойда – Хоара, спрямованим на специфікацію та верифікацію програмних систем, в яких використовується динамічна пам'ять. Її основним засобом є рамкове правило, що дає змогу працювати з локальними доведеннями. Монотонне розширення логіки Флойда – Хоара було одержано, використовуючи композиційно-номінативний підхід. Основні переваги отриманої логіки полягають у застосуванні номінативних даних для опису стану програми та у роботі з частковими предикатами. Основна мета пропонованого дослідження – розроблення рамкового правила та засобів роботи з вказівниками для монотонної логіки Флойда – Хоара. З огляду на це, вона була збагачена інструментами для підтримки локальних доведень, що надзвичайно важливо сьогодні, коли складність програмного забезпечення лише зростає. Локальність означає, що властивість частини програми може бути доведена окремо та поєднана з властивостями інших компонентів, якщо визначені умови виконуються. Для цього номінативні дані було розширено поняттям адреси. Представлено нові спеціальні композиції збереження в динамічній пам'яті і завантаження з неї. Логічна зв'язка відокремлювальна кон'юнкція була перевизначена як композиція в програмній алгебрі. Доведено, що така композиція є монотонною. Щоб мати можливість розширити доведення для апроксимацій програми до доведення для програми загалом, усі композиції мають бути монотонними. Крім того, додано правила системи виводу для нових операторів. Правила для зберігання в динамічну пам'ять та завантаження з неї є модифікацією правила для оператора присвоєння. Рамкове правило – це правило, яке за допомогою відокремлювальної кон'юнкції описує, як можна розширити істинну трійку Хоара за допомогою твердження для частини пам'яті, яка не була змінена програмою. Було показано, що варіант правила, реалізований у логіці розділення, вже не є коректним для монотонного розширення. Були представлені модифіковані умови, за яких досягається коректність. Темою майбутніх досліджень є можливі обмеження на операції з вказівниками, які дають можливість запропонувати конструктивні умови для рамкового правила. Іншим важливим питанням, що виходить за межі цієї статті, є розгляд використання складних номінативних даних разом з динамічною пам'яттю.

К л ю ч о в і с л о в а: ***формальні методи, верифікація ПС, програмні логіки, номінативні дані, логіка розділення.***

Автори заявляють про відсутність конфлікту інтересів. Спонсори не брали участі в розробленні дослідження; у зборі, аналізі чи інтерпретації даних; у написанні рукопису; в рішенні про публікацію результатів.

The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses or interpretation of data; in the writing of the manuscript; in the decision to publish the results.