УДК 1(16)

МОНАДНЫЕ ВЫЧИСЛЕНИЯ В ТЕОРЕТИКО-ТИПОВОЙ СЕМАНТИКЕ

О. А. Доманов
Институт философии и права СО РАН (г. Новосибирск)
odomanov@gmail.com

Аннотация. Монада является теоретико-категорной конструкцией, которую возможно понимать как абстракцию вычисления [Moggi, 1991]. Это вычисление, результатом которого является не просто вычисляемая величина, а эта величина с дополнительной структурой. Так понятая, монада успешно применяется в последние годы в семантике естественного языка в рамках формализма Монтегю. Данная статья посвящена применению монадных вычислений в теоретико-типовой семантике [Ranta, 1994]. На нескольких примерах демонстрируется применение монад, а также аппликативных функторов, для учёта различных контекстов, управления порядком применения кванторов, формализации представления мнений (belief reports). Формализация проводится в функциональном языке Agda.

Ключевые слова: семантика, естественный язык, теория типов, монады, вычисления, Agda.

Для цитирования: Доманов, О. А. (2023). Монадные вычисления в теоретико-типовой семантике. *Respublica Literaria*. Т. 4. № 4. С. 70-81. DOI: 10.47850/RL.2023.4.4.70-81.

MONAD COMPUTATIONS IN TYPE THEORETICAL SEMANTICS

O. A. Domanov
Institute of Philosophy and Law SB RAS (Novosibirsk)
odomanov@gmail.com

Abstract. Monad is a construction from the category theory, which could be understood as an abstraction of computation [Moggi, 1991]. This is a computation resulting not simply in the computed value but in this value with an additional structure. In recent years, monad in this sense has been successfully applied to natural language semantics in the framework of Montague's formalism. This article deals with monad computation applications in type theoretical semantics [Ranta, 1994]. Several examples illustrate the usage of monads as well as applicative functors for contexts accounting in various situations, controlling the order of quantifiers, belief reports formalization. The language of formalization is Agda.

Keywords: semantics, natural language, type theory, monads, computations, Agda.

For citation: Domanov, O. A. (2023). Monad Computations in Type Theoretical Semantics. *Respublica Literaria*. Vol. 4. no. 4. pp. 70-81. DOI: 10.47850/RL.2023.4.4.70-81.

Идея монады как понятия вычисления восходит к работам Е. Могги [Moggi, 1991]. Монада – это понятие из теории категорий, обозначающее отображения особого рода (см. ниже). Эти отображения формализуют вычисления, результатами которых являются не просто вычисляемые значения, а значения с дополнительной структурой. Например, параллельно собственно вычислению может сохраняться информация о состоянии программы, выполняться какие-то побочные действия и т. д. Формально, если простое вычисление производится с величинами типа T, то монадное – с величинами типа MT, представляющими собой T с дополнительной структурой. В теории категорий T представляет собой категорию,

т. е. набор объектов с морфизмами между ними, а M – функтор, отображающий категорию T в обогащённую категорию MT. На функтор M накладываются требования сохранения исходной структуры T, что фактически делает его эндофунктором, т. е. отображением некоторой категории в себя.

В формальной семантике естественного языка в качестве категории выступают выражения, на которых интерпретируется язык, вместе с их трансформациями, соответствующими синтаксическим трансформациям. Как правило, следуя Монтегю [Montague, 1974], как язык интерпретации используют λ-исчисление с двумя основными типами e и t [см., например: Asudeh, 2014; Asudeh and Giorgolo, 2020]. В отличие от этих подходов, я ниже использую теоретико-типовую семантику [Ranta, 1994], в которой выражения естественного языка интерпретируются на теории типов, восходящей к работам Мартин-Лёфа [Martin-Löf, 1984] (более конкретно, используемый ниже язык Agda основан на Единой теории типов UTT [Luo, 1994]). Во многих случаях этот подход оказывается более выразительным. Изложение основано на нескольких примерах, демонстрирующих возможности подхода, причём помимо монады используется также более простая структура, которая называется аппликативным функтором.

В качестве языка формализации используется функциональный язык Агда [Agda Documentation; Norell, 2009]. В статье излагаются лишь основные идеи и подходы, полную формализацию можно найти по адресу https://github.com/odomanov/ttsemantics/tree/v3/Agda/Functors.

Аппликативные функторы и монады

Пусть имеется отображение `F : Set ℓ → Set ℓ`, которое мы будем понимать как конструктор типов, т. е. функцию, переводящую типы `A` в типы `F A` с дополнительной структурой. Приведём несколько примеров таких отображений:

```
F₁ A = W → A
F₂ A = A × S
F₃ A = Either E A
```

Если понимать `W` как множество возможных миров, то `F₁` переводит тип в его интенсионал; `F₂` добавляет к типу информацию `S`; `F₃` конструирует тип для результатов вычислений, которые могут привести к ошибке – таким результатом может быть либо значение `A`, либо ошибка `E`. Для дальнейшего часто полезно неформально понимать `F A` как «контейнер», в который помещено значение типа `A` и из которого последнее можно «извлекать», причём способ извлечения различается для разных `F`.

Аппликативным функтором называется следующая структура:

```
record Applicative {ℓ} (F : Set ℓ → Set ℓ) : Set _ where
  field
    pure  : ∀{A : Set ℓ} → A → F A
    _<*>_ : ∀{A B : Set ℓ} → F (A → B) → F A → F B
```

Она содержит две операции. Функция `pure` «поднимает» тип `A` до типа `F A` (или помещает его в контейнер `F A`). Функция `_<*>_` это композиция «внутри» типов `F A`, которая, фактически, является применением функции `F (A → B)` к аргументу `F A`. Эти функции подчиняются определённым аксиомам, которые, например, обеспечивают, что `pure` играет роль единицы (см. примеры), а `_<*>_` оказывается ассоциативной. В частности, это делает отображение `F` функтором и позволяет определить функцию

```
_<$>_ : ∀{A B : Set ℓ} → (A → B) → (F A → F B)
f <$> fa = pure f <*> fa
```

Монада является аппликативным функтором с дополнительным морфизмом $FF \to F$. Он позволяет проводить композицию монадных операций. Однако монада обычно определяется независимо от аппликативного функтора как эндофунктор с двумя операциями (приведённое ниже определение является псевдокодом, фактическое определение сложнее):

```
record Monad {ℓ} (M : Set ℓ → Set ℓ) : Set _ where
  field
    return : ∀{A} → A → M A
    _>>=_  : ∀{A B} → M A → (A → M B) → M B
```

Функция `return` совпадает с `pure` и исполняет ту же роль. Функция `_>>=_` (bind) определяет композицию и позволяет объединять операции в цепочки. Например, мы можем писать `ma >>= f >>= g` и т. д., для `ma : M A`, `f : A → M B`, `g : B → M C` и т. д. Для этого эти две операции должны подчиняться законам, которые, фактически, делают монаду моноидом на эндофункторах. В терминах контейнеров, `ma >>= f` должно быть определено таким образом, чтобы соответствующее вычисление состояло в извлечении значения типа `A` из `ma : M A` и применении к нему функции `f`, в результате чего получается значение типа `M B`. Каким конкретно образом это происходит зависит от устройства функтора `M`. Мы увидим примеры ниже.

Рассмотрим теперь применение аппликативных функторов и монад в семантике естественного языка.

Пример 1. Зависимость от контекста

Определим множество контекстов или возможных миров (для простоты, всего с двумя мирами):

```
data ВозмМир : Set where
  w₁ w₂ : ВозмМир
```

Пусть дополнительной структурой для типа будет его зависимость от возможного мира, т. е. функтор `F` имеет вид:

```
F : ∀{a} → Set a → Set a
F A = ВозмМир → A
```

Другими словами, `F` переводит тип в его интенсионал. «Извлечение из контейнера» для `f : F A` состоит в применении `f` к некоторому миру `w`.

Определим аппликативный функтор, задав его операции:

```
applicative : ∀{ℓ} → Applicative {ℓ} F
applicative = record { pure  = λ x w → x
                     ; _<*>_ = λ f x w → (f w) (x w)
                     }
```

Здесь `pure` переводит в постоянную функцию, со значением, не зависящим от `w` («жёсткий десигнатор»). Что же касается применения интенсионала функции к интенсионалу аргумента `f <*> x`, то оно состоит в том, что для определённого мира `w` мы сначала извлекаем значения из `f` и `x`, а затем применяем одно из них к другому.

Рассмотрим конкретный пример. Пусть есть тип людей `Human`, состоящий из трёх человек:

```
data Human : Set where
  Mary Naomi Alex : Human
```

Определим предикат `run` («бежит»). В теоретико-типовой семантике глаголы интерпретируются зависимыми типами (см. [Ranta, 1994]). Этот предикат, вообще говоря, различается в разных мирах:

```
data run₁ : Human → Set where
  mr : run₁ Mary

data run₂ : Human → Set where
  ar : run₂ Alex

run : F (Human → Set)
run w₁ = run₁
run w₂ = run₂
```

Мы здесь определили его так, что в мире `w₁` имеется доказательство `mr` того, что Мэри бежит, а в мире `w₂` – доказательство `ar` того, что Алекс бежит. Пусть далее местоимение `she` имеет разные интерпретации в разных мирах:

```
she : F Human
she w₁ = Mary
she w₂ = Naomi
```

Интерпретация предложения «Она бежит» выглядит тогда следующим образом:

```
she-runs : F _
she-runs = run <*> she
```

Как можно проверить, эта пропозиция имеет в мире `w₁` доказательство `mr`:

DOI: 10.47850/RL.2023.4.4.70-81

```
_ : she-runs w₁
_ = mr
```

Зависимый тип «отец» также различен в разных мирах. Пусть отец не всегда существует, т. е. будем считать, что его тип равен `Maybe Human` – опциональный тип, значением которого может быть `nothing`.

```
father₁ : Human → Maybe Human
father₁ Mary = just Alex
father₁ _ = nothing

father₂ : Human → Maybe Human
father₂ Naomi = just Alex
father₂ _ = nothing

father : F (Human → Maybe Human)
father w₁ = father₁
father w₂ = father₂
```

Таким образом, `Alex` является отцом `Mary` в первом мире и отцом `Naomi` – во втором. Интерпретация термина «её отец» выглядит следующим образом:

```
her-father : F _
her-father = father <*> she
```

Как и следовало ожидать, Алекс является отцом в обоих случаях:

```
_ : her-father w₁ ≡ just Alex
_ = refl

_ : her-father w₂ ≡ just Alex
_ = refl
```

(здесь ≡ обозначает пропозициональное, т. е. доказываемое равенство, в отличие от равенства по определению =).

Проверим теперь как работает композиция. Сначала определим глагол `run` для `Maybe Human`:

```
runm : F (Maybe Human → Set)
runm w₁ (just x) = run₁ x
runm w₁ nothing = ⊥
runm w₂ (just x) = run₂ x
runm w₂ nothing = ⊥
```

Здесь ⊥ означает пустой тип, т. е. отсутствие пропозиции. Тогда предложение «Её отец бежит» становится

```
her-father-runs : F _
her-father-runs = runm <*> (father <*> she)
```

Как можно проверить, эта пропозиция имеет доказательство в мире `w₂`:

```
_ : her-father-runs w₂
_ = ar
```

В подобных примерах в качестве возможных миров могут выступать любые контексты, от которых зависят значения, с которыми мы работаем. Эта структура, фактически, является монадой и известна как Reader Monad; она позволяет композиционально формализовать зависимость от контекстов в разных смыслах.

Пример 2. Продолжения

Продолжение (continuation) позволяет формализовать контекст выражения или «оставшуюся часть вычисления». Например, для P в выражении $\forall x P$ контекст можно изобразить как $\forall x[\,]$, где $[\,]$ обозначает «дырку», которая должна быть заполнена, чтобы вычисление завершилось. Здесь продолжение вычисления можно не вполне формально изобразить как функцию $\lambda k.\forall x[kx]$, где k – функция продолжения (или просто продолжение), и kx помещается на место дырки в выражении. Для работы с продолжениями все величины представляются как имеющие такие дырки, которые требуется заполнить, чтобы закончить вычисление, т. е. как функции вида $\lambda k.f[kx]$. Это можно рассматривать как обобщение абстрагирования, известного из λ-исчисления. Дырки указывают на элементы в структурах, которые в этом смысле абстрагируются. В простейших случаях продолжение может быть просто предикатом, но в более интересных оно может представлять собой как угодно сложное вычисление. Продолжения позволяют управлять порядком исполнения вычислений. В лингвистике это оказывается важным во многих случаях, самым известным из которых является, вероятно, зависимость смысла от порядка кванторов (см. пример ниже). Продолжения используются достаточно давно в формальной семантике вообще [Wadler, 1994] и в семантике естественного языка в частности [Barker, 2002]. Например, книга Баркера и Шана «Continuations and Natural Language» [Barker and Shan, 2014] полностью посвящена использованию продолжений в анализе естественного языка. Она, однако, использует формализацию Монтегю, а монады не играют в ней практически никакой роли и упоминаются лишь в самом конце книги. Мы рассмотрим, как этот подход можно использовать в теоретико-типовой семантике.

В нотации языка Агда при переходе к продолжениям мы заменяем тип `A` на тип функций вида `(A → R) → R`, где `(A → R)` обозначает функцию продолжения. Таким образом, вместо `A` мы имеем выражение, которое при добавлении к нему продолжения вычисляет значение типа `R`. Рассмотрим в качестве примера универсальный квантор. Его можно представить как функцию `λ k → ∀(x : A) → k x`, где `k` обозначает продолжение, т. е. функцию типа `A → Set`. Применение выражения выше к продолжению `k` даёт результатом выражение `∀(x : A) → k x`, которое имеет тип `Set`.

Для построения монады для продолжений мы воспользуемся определением монады из стандартной библиотеки Агды. В ней определена монада `RawMonad`, которая содержит

только операции `return` и `_>>=_`, но не доказательства монадных законов. Тогда монада для продолжений выглядит следующим образом:

```
Cont : ∀{a b}(R : Set a) → Set (a ⊔ b) → Set (a ⊔ b)
Cont R A = (A → R) → R

MonadCont : ∀{a b} (R : Set a) → RawMonad (Cont {a} {b} R)
MonadCont R = record { return = λ a k → k a
                     ; _>>=_ = λ ma f k → ma (λ x → f x k)
                     }
```

Здесь мы сначала определяем монадное значение `Cont`, т. е. функтор, преобразующий тип `A` в тип `(A → R) → R`. Последний является значением, которому требуется продолжение `(A → R)` для получения результата `R`. Затем мы определяем монаду продолжения `MonadCont` со значениями `Cont`. Функция `return` просто применяет продолжение `k` к объекту `a`. Значение `ma >>= f` вычисляется следующим образом. `ma : Cont R A` требует продолжения, т. е. функции типа `A → Cont R B`. Чтобы его получить, мы применяем `f : A → Cont R B` к `x : A`, а затем добавляем продолжение `k`. Если изобразить это в виде дырок, то `ma` будет иметь форму $\lambda k.a[kx]$, а функция `f` – форму $\lambda x.\lambda k.b[kx]$. Тогда композиция `ma >>= f` оказывается функцией $\lambda k.a[b[kx]]$. «Извлечение из контейнера» для величин типа `Cont` выполняется путём применения их к тождественной функции `id`.

Как можно проверить (см. указанный файл), эти определения подчиняются монадным законам.

Рассмотрим конкретный пример. Пусть у нас есть тип людей `Human` и два предиката, соответствующих непереходному глаголу «бежать» и переходному глаголу «любить»:

```
Human   : Set
_runs   : Human → Set
_loves_ : Human → Human → Set
```

Определим интерпретацию для «someone» и «everyone»:

```
someone : Cont Set Human
someone = λ k → Σ[ x ∈ Human ] k x

everyone : Cont Set Human
everyone = λ k → ∀(x : Human) → k x
```

Они представляют собой выражения типа `Cont Set Human`, которые при применении к продолжению `k` дают формулы с экзистенциальным и универсальным квантором, соответственно.

Монадные операции позволяют нам интерпретировать предложение «Someone runs»:

```
someone-runs₁ : Cont Set Set
someone-runs₁ = someone >>= (λ x → return (x runs))
```

Применив его к тождественной функции `id` в качестве продолжения, получим:

```
someone-runs₁ id ≡ Σ[ x ∈ Human ] x runs
```

Как видим, мы получили экзистенциальное утверждение, как и могли бы ожидать.

Для работы с монадами Агда определяет так называемую do-нотацию, позволяющую сделать запись вычислений более наглядной. Она является чисто синтаксическим переопределением. Например, выражение выше мы могли бы переписать так:

```
someone-runs₁ = someone >>= (λ x →
                              return (x runs))
```

В do-нотации то же самое переписывается следующим образом:

```
someone-runs₂ = do
  x ← someone
  return (x runs)
```

Здесь сразу видна последовательность действий: сначала вычисляется `someone`, причём извлечённое из результата значение обозначается `x`, а затем выводится в качестве конечного результата `x runs`.

Пропозиция с универсальным квантором в do-нотации записывается следующим образом:

```
everyone-runs : Cont Set Set
everyone-runs = do
  x ← everyone
  return (x runs)


everyone-runs id ≡ ∀(x : Human) → x runs
```

Рассмотрим, как мы можем регулировать последовательность вычислений. Как известно, предложение «Everyone loves someone (Каждый человек любит некоторого человека)» имеет две интерпретации, различающиеся порядком применения кванторов. Наш формализм позволяет их различить. Действительно, рассмотрим первую интерпретацию:

```
everyone-loves-someone₁ : Cont Set Set
everyone-loves-someone₁ = do
  x  ← everyone
  y  ← someone
  return (x loves y)
```

При вычислении получаем

```
everyone-loves-someone₁ id ≡
            ∀(x : Human) → Σ[ y ∈ Human ] x loves y
```

Эта интерпретация означает, что каждый человек любит своего человека. Если же мы поменяем порядок операций, то получим:

```
everyone-loves-someone₂ : Cont Set Set
everyone-loves-someone₂ = do
  x ← someone
  y ← everyone
  return (y loves x)

everyone-loves-someone₂ id ≡
            Σ[ x ∈ Human ] (∀(y : Human) → y loves x)
```

Эта интерпретация говорит, что все люди любят одного и того же человека.

Монады, как мы видим, позволяют нам не только учитывать разные продолжения, но и регулировать их порядок.

В качестве второго примера рассмотрим контексты мнения. Предложение «Ralph believes someone is a spy (Ральф верит, что кто-то шпион)» можно интерпретировать двояко. Либо Ральф верит, что существует какой-то человек, являющийся шпионом, либо он верит, что некоторый определённый человек шпион. Если обозначить через $RBel(s)$ пропозицию «Ральф верит, что s», то эти две интерпретации записываются следующим образом:

$$RBel(\exists x.x\ spy)$$
$$\exists x.RBel(x\ spy).$$

Посмотрим, можем ли мы получить эти две интерпретации в нашем формализме как некоторые композиции.

Пусть имеются `Ralph` типа `Human` и два предиката

```
Ralph      : Human
_is-spy    : Human → Set
_believes_ : Human → Set → Set
```

Прямая интерпретация

```
Ralph-believes-some-spy₁ : Cont Set Set
Ralph-believes-some-spy₁ = do
  x ← someone
  return (Ralph believes (x is-spy))

Ralph-believes-some-spy₁ id ≡ Σ[ x ∈ Human ] Ralph believes (x is-spy)
```

позволяет нам получить только один вариант, поскольку неясно, как мы можем здесь поменять порядок операций. Поэтому определим следующий тип:

```
record RB {a} (A : Set a) (P : A → Set) : Set a where
  field
    s1 : A
    s2 : P s1
```

Он позволяет нам учесть область действия (scope) предиката «Ральф верит, что…». Запись `RB A P` означает, что Ральф верит о некотором объекте `s1` типа `A`, что он `P s1`. Агда позволяет ввести синтаксическую замену, в которой нужные нам выражения будут представлены более наглядно:

```
RB A (λ x → P) = RB[ x ∈ A ] P
```

Таким образом, `RB[ x ∈ A ] P` читается как «Ральф верит о некотором `x : A`, что `P`». Я намеренно сделал это определение сходным с экзистенциальным квантором `Σ[ x ∈ A ] P`. Действительно, речь здесь идёт об одном и том же явлении, об области действия, в одном случае – предиката, в другом – глагола «believe».

Тип `RB` позволяет нам определить монадное значение

```
RBel : Cont Set Human
RBel = λ k → RB[ x ∈ Human ] k x
```

Оно означает: «Ральф верит о некотором человеке `x`, что `k x`».

Определив далее монадное значение для предиката `_is-spy`:

```
_*is-spy : Human → Cont Set Set
x *is-spy = return (x is-spy)
```

мы можем интерпретировать наше предложение. Рассмотрим первый вариант:

```
RBel-some-spy₁ : Cont Set Set
RBel-some-spy₁ = do
  x ← RBel
  x *is-spy

RBel-some-spy₁ id ≡ RB[ x ∈ Human ] x is-spy
```

Словами это можно передать как: «Ральф верит, что существует человек, который шпион».

Добавив `someone` впереди, получим:

```
RBel-some-spy₂ : Cont Set Set
RBel-some-spy₂ = do
  x ← someone
  RBel
  x *is-spy

RBel-some-spy₂ id ≡ Σ[ x ∈ Human ] RB[ _ ∈ Human ] x is-spy
```

Эта интерпретация выражается словами как «Существует человек, о котором Ральф верит, что он шпион».

Заметим, что во втором случае в записи `RB[ x ∈ Human ] P` тип `P` на самом деле не зависит от `x`. Поэтому, в частности, нам не понадобилось значение `RBel` в определении `Rbel-some-spy₂`. Введём для этого случая сокращение:

```
RBelp : ∀{a}{A : Set a}(P : Set) → Set a
RBelp {a}{A} P = RB A (λ _ → P)
```

Таким образом, `RBel P` означает просто «Ральф верит, что `P`». Тогда можно записать:

```
RBel-some-spy₂ id ≡ Σ[ x ∈ Human ] RBelp (x is-spy)
```

Как мы видим, изменение порядка вычислений в монаде позволяет нам формализовать различные интерпретации выражений естественного языка. Монады позволяют учитывать контекст и область действия кванторов, предикатов мнения и пр. Существенно при этом, что получающаяся семантика оказывается композициональной.

Список литературы / References

Agda Documentation (2023). Available at: https://agda.readthedocs.io/ (Accessed: 10 July 2023).

Asudeh, A. (2014). *Monads: Some Linguistic Applications*. Available at: http://www.sas.rochester.edu/lin/sites/asudeh/handouts/asudeh-se-lfg13.pdf (Accessed: 10 July 2023).

Asudeh, A. and Giorgolo, G. (2020). *Enriched Meanings. Natural Language Semantics with Category Theory*. Oxford. OUP. 179 p.

Barker, C. (2002). Continuations and the nature of quantification. *Natural Language Semantics*. Vol. 10. no. 3. pp. 211-242.

Barker, C. and Shan, C.-c. (2014). *Continuations and Natural Language*. Oxford. OUP. 230 p.

Luo, Z. (1994). *Computation and Reasoning. A Type Theory for Computer Science*. Oxford. OUP. 228 p.

Martin-Löf, P. (1984). *An Intuitionistic Type Theory. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980*. Napoli. Bibliopolis. 91 p.

Moggi, E. (1991). Notions of computation and monads. *Information and Computation*. Vol. 93. no. 1. pp. 55-92. DOI: 10.1016/0890-5401(91)90052-4.

Montague, R. (1974). *Formal Philosophy. Selected Papers of Richard Montague*. Thomason, R. H. (ed.). With an intro. by Thomason, R. H. New Haven and London. Yale University Press. 369 p.

Norell, U. (2009). Dependently Typed Programming in Agda. In Koopman, P., Plasmeijer, R., and Swierstra, D. (eds.). *Advanced Functional Programming: 6th International School, AFP 2008*. Berlin, Heidelberg. Springer-Verlag. pp. 230-266. DOI: 10.1007/978-3-642-04652-0_5.

Ranta, A. (1994). *Type-theoretical grammar*. Clarendon Press. 226 p.

Wadler, P. L. (1994). Monads and composable continuations. *Lisp and Symbolic Computation*. Vol. 7. no. 1. pp. 39-56.

Сведения об авторе / Information about the author

Доманов Олег Анатольевич — кандидат философских наук, доцент, старший научный сотрудник Института философии и права СО РАН, г. Новосибирск, Николаева, 8, e-mail: odomanov@gmail.com, http://orcid.org/0000-0003-0057-3901.

Статья поступила в редакцию 20.04.2023
После доработки 10.10.2023
Принята к публикации 20.11.2023

Oleg Domanov — Candidate of Philosophical Sciences, associate Professor, Senior Researcher of the Insitute of Philosophy and Law of the Siberian Branch of the Russian Academy of Sciences, Novosibirsk, Nikolaeva Str., 8, e-mail: odomanov@gmail.com, http://orcid.org/0000-0003-0057-3901.

The paper was submitted 20.04.2023
Received after reworking 10.10.2023
Accepted for publication 20.11.2023