УДК 004.451.2, 004.82, 004.021, 519.6, 004.42

Операционная семантика выражений в языке Rust на языке ABML

Бодин Е.В. (Институт систем информатики СО РАН)

Ануреев И.С. (Институт систем информатики СО РАН)

В статье рассматривается формальное описание операционной семантики выражений языка программирования Rust с использованием предметно-ориентированного языка моделирования ABML. Основное внимание уделяется динамическим аспектам вычислений, включая управление памятью, владение, заимствование и проверку конфликтов доступа на этапе выполнения.

Предлагаемый подход опирается на онтологическое представление синтаксических и семантических сущностей Rust, что позволяет единообразно описывать выражения, блоки и структуры данных как элементы единой вычислительной модели. В отличие от традиционных формализаций, модель явно включает метаданные безопасности, необходимые для воспроизведения механизмов ownership и borrow checking.

Особенностью работы является использование иерархической модели памяти, позволяющей корректно описывать частичное заимствование структур и доступ к их полям. Это обеспечивает более точную динамическую семантику по сравнению с плоскими моделями памяти и демонстрирует соответствие формальных правил реальному поведению программ на Rust.

Полученная операционная семантика является исполняемой и может служить основой для анализа программ, прототипирования интерпретаторов и дальнейших исследований в области формальной верификации языков с управляемой безопасностью памяти.

Ключевые слова: операционные семантики, онтологии языков программирования, модели языков программирования, атрибутные замыкания, ABML, Rust, управление памятью, онтологическое моделирование

1. Введение

Современные языки программирования системного уровня все чаще ориентируются на строгие гарантии безопасности памяти, которые должны обеспечиваться без значительного снижения производительности. Язык Rust представляет собой один из наиболее успешных примеров такого подхода, сочетая низкоуровневый контроль над ресурсами с

формально заданными правилами владения и заимствования. Эти правила традиционно проверяются на этапе компиляции, однако их точная семантическая интерпретация требует аккуратного формального описания.

Формальная операционная семантика играет ключевую роль в понимании поведения программ, анализе корректности и построении инструментов верификации. Для языков с нетривиальной моделью памяти, таких как Rust, стандартные подходы, основанные на простых состояниях вида «память—окружение», оказываются недостаточными. В частности, они не позволяют напрямую выразить ограничения, связанные с одновременным доступом к данным, жизненным циклом значений и частичным заимствованием составных объектов.

В данной работе предлагается использовать онтологический подход к заданию операционной семантики, реализованный с помощью языка ABML [21]. В рамках этого подхода вычислительное состояние рассматривается как совокупность объектов и их атрибутов, а динамика исполнения описывается через вычисление и обновление этих атрибутов. Такой взгляд позволяет естественным образом интегрировать в модель дополнительные семантические слои, в том числе метаданные, отвечающие за безопасность памяти.

ABML ранее применялся для формального описания семантики языковых конструкций, включая операторы передачи управления в языке C [20]. Эти исследования показали, что онтологическое моделирование обеспечивает модульность, расширяемость и исполняемость семантики. Настоящая статья развивает данный подход применительно к языку Rust, фокусируясь не на управляющих конструкциях, а на выражениях и связанных с ними механизмах работы с памятью.

В статье формализуются базовые синтаксические сущности Rust, включая выражения, объявления переменных и блоки, а также вводится иерархическая модель локаций памяти. Особое внимание уделяется операционной семантике заимствования, разыменования и присваивания, а также алгоритму динамической проверки конфликтов, моделирующему поведение borrow checker. Рассматриваемые примеры демонстрируют, как предложенная модель воспроизводит как корректные сценарии доступа к данным, так и ситуации, приводящие к ошибкам выполнения.

Таким образом, целью работы является построение исполняемой операционной семантики выражений Rust на основе ABML, которая не только отражает ключевые свойства языка, но и может служить фундаментом для дальнейших исследований в области фор-

мальных методов, анализа программ и разработки инструментов поддержки Rust.

Статья организована следующим образом. В разделе 2 вводится онтология выражений и типов данных языка Rust, описываются базовые синтаксические сущности и их семантические роли в вычислительной модели. В разделе 3 рассматриваются модели агентов и окружения, используемые для представления состояния вычислений, включая иерархическую модель памяти и метаданные безопасности. Раздел 4 посвящен формальному заданию операционной семантики выражений Rust на языке ABML, включая объявления переменных, вычисление базовых выражений, присваивание и доступ к полям структур. Также в нем подробно рассматривается операционная семантика заимствования, разыменования и универсальное правило проверки заимствований, моделирующее поведение borrow checker. В разделе 5 приводятся иллюстративные и комплексные примеры выполнения программ, демонстрирующие работу иерархической проверки конфликтов и частичного заимствования структур. Далее следует раздел «Родственные работы», в котором проводится сопоставление предложенного подхода с существующими формализациями семантики Rust и других языков программирования. В заключении подводятся итоги работы и обсуждаются направления дальнейших исследований.

2. Онтология выражений и типов данных языка Rust

В данном разделе вводится онтология выражений, констант и типов данных. В онтологическом подходе к спецификации операционной семантики языков программирования [20] онтология конструкций языка Rust описывается набором типов языка ABML, полное описание которого можно найти в [21].

2.1. Имена

В этом подразделе описываются типы для разных видов имен, используемых в Rust-программах:

```
(typedef "name" (uniont symbol string))
(typedef "variable" "name")
(typedef "field name" "name")
(typedef "struct name" "name")
```

Таким образом, все имена моделируются лисповскими строками.

2.2. Константы и значения

В этом подразделе описываются типы для разных видов констант языка Rust, а также значений, которые могут возвращать выражения на этом языке:

```
(typedef "constant" (uniont "i32 value" "bool value"))

(typedef "i32 value" int)
(typedef "bool value" (enumt "true" "false"))
(typedef "()" (enumt "()"))

(typedef "value" (uniont "constant" "()" "reference"))

(mot "reference" :at "location" "location"
  :at "lifetime" "lifetime")

(typedef "location" (uniont "simple location" "struct location"
  "field location"))
(mot "simple location")
(mot "struct location" :amap "field name" "location")
(mot "field location")

(mot "lifetime")
```

Экземпляры типов `"i32 value"` и `"bool value"` являются моделями значений типов `i32` и `bool` языка Rust. Язык Rust имеет и другие примитивные типы, но мы для простоты в этой статье ограничиваемся только этими двумя. Остальные типы моделируются аналогичным образом.

Тип `"()"` моделирует значение `()` в языке Rust типа `unit`.

Тип `"location"` моделирует локации (адреса, ячейки памяти) в языке Rust. Мы выделяем 3 подтипа локаций – локации, связанные с переменными примитивных типов; локации, связанные с переменными типа структуры и локации, связанные с полями структуры. В языке Rust имеются и другие составные типы помимо структур, например, кортежи, но мы для простоты ограничиваемся только структурами, поскольку моделирование значе-

ний других составных типов делается аналогичным образом.

Тип "lifetime" моделирует ссылки, которые связаны с локациями и имеют время жизни.

Тип "lifetime" определяет значения, которые описывают время жизни для ссылок.

2.3. Типы данных

В этом подразделе собраны типы языка ABML, моделирующие типы языка Rust. Для простоты мы ограничиваемся небольшим набором типов, но это набор несложно расширить:

```
(typedef "type" (uniont "i32" "bool" "unit" "struct name"
   "struct type"
  "&T1" "&mutT1"))

(typedef "i32" (enumt "i32"))
(typedef "bool" (enumt "bool"))
(typedef "unit" (enumt "unit"))
(cot "struct type" :amap "field name" "type")
(cot "&T" :at "type" "type")
(cot "&mutT" :at "type" "type")
```

Типы "i32" и "bool" моделируют типы i32 и bool языка Rust.

Тип "unit" моделирует тип unit языка Rust.

Тип "struct type" моделирует типы структур, задавая их поля и типы этих полей.

Типы "&T1" и "&mutT1" моделируют типы для обычных и мутабельных ссылок.

2.4. Выражения

Модели выражений языка Rust, рассматриваемые в этой статье, на языке ABML определяются следующим набором типов:

```
(typedef "expression" (uniont "variable" "constant" "1.2" "&1"
  "&mut1" "*1" "1+2"))

(mot "1.2" :at 1 "expression" :at 2 "field name")
(mot "&1" :at 1 "expression")
```

```
(mot "&mut1" :at 1 "expression")
(mot "*1" :at 1 "expression")
(mot "1+2" :at 1 "expression" :at 2 "expression")

(mot "1=2" :at 1 "identifier" :at 2 "expression"
  :at "type" "type")
(mot "let1:=2" :at 1 "identifier" :at 2 "expression"
  :at "type" "type")
(mot "let mut1:=2" :at 1 "identifier" :at 2 "expression"
  :at "type" "type")
(mot "{1}" :at 1 (listt "expression"))
```

Они моделируют небольшой, но достаточный набор выражений для описания основных концепций операционной семантики языка Rust. Заметим, что для моделей выражений `assign`, `let` и `let mut` языка Rust, мы считаем, что тип атрибута `1` известен и хранится в атрибуте `"type"`.

3. Модели агентов и окружения

Состояние программы в ABML моделируется агентом, который оперирует знаниями о памяти и окружении. Для языка Rust множество агентов определяется следующим типом:

```
(mot "agent"
  :at "location" (cot :amap "variable" "location")
  :at "mutability" (cot :amap "variable"
    (enumt "mutable" "immutable"))
  :at "location value" (cot :amap "location" "value")
  :at "location type" (cot :amap "location" "type")
  :at "borrows" (cot :amap "location" (listt "borrow"))
  :at "lifetimes" (cot :amap "lifetime" (listt "location"))
  :at "value" "value"
```

Атрибут `"location"` связывает переменные программы с локациями.

Атрибут `"mutability"` определяет, является ли эта связь мутабельной или нет.

Атрибуты `"location value"` и `"location type"` задают значения, хранящиеся в лока-

циях и типы этих значений, соответственно.

Атрибут "borrows" определяет стеки заимствований для локаций.

Атрибут "lifetimes" описывает, как локации распределяются по времени жизни.

Атрибут "value" – это стандартный атрибут языка ABML, который хранит последнее вычисленное значение.

Заимствования и время жизни определяются следующим образом:

```
(mo "borrow" :at "lifetime" "lifetime"
  :at "kind" (enumt "free", "shared", "unique"))

(mo "lifetime")
```

4. Операционная семантика моделей выражений

В данном разделе задается исполняемая семантика конструкций Rust в виде атрибутных замыканий [21]. В отличие от традиционных интерпретаторов, семантика на языке ABML описывает не просто изменение значений, а трансформацию базы знаний агента, включая обновление метаданных безопасности.

Семантика Rust разбивается на семантику мест (возвращает локацию), семантику r-значений (возвращает значение) и семантику операторов (изменяет содержимое агента), которые задаются атрибутными замыканиями для атрибутов "place", "rvalue" и "statement", соответственно.

4.1. Семантика мест

Семантика мест определяется для выражений типов "variable", "1.2" и "*1".

Для моделей переменных семантика определяется следующим образом:

```
(aclosure ac :attribute "place" :type "variable" :do
  (update-push-aclosure ac :av "stage" "returning value")
  (update-push-aclosure ac :av "stage" "checking location"))

(aclosure ac :attribute "place" :type "variable"
  :stage "checking location" :instance i :agent a
  :ap a (aseq "location" i) loc :match
    :v (not (null loc)) T
```

```
    :do loc
    :exit (co "stop next aclosure" :av "type" "error" :av
      "aclosure" ac))

(aclosure ac :attribute "place" :type "variable"
  :stage "returning value" :value loc :do loc)
```

Таким образом, возвращается локация, связанная с переменной, в том случае, если такая связь есть. В противном случае, выполнение программы останавливается и выдается ошибка.

Семантика операции доступа к полю структуры задаются аналогичным образом:

```
(aclosure ac :attribute "place" :type "1.2" :do
  (update-push-aclosure ac :av "stage" "returning value")
  (update-push-aclosure ac :av "stage" "checking field location")
  (update-push-aclosure ac :av "stage" "evaluating 1")

(aclosure ac :attribute "place" :type "1.2" :stage "evaluating 1"
  :instance i :ap i 1 v1 :do
    (clear-update-eval-aclosure ac :attribute "place"
      :instance v1)))

(aclosure ac :attribute "place" :type "1.2"
  :stage "checking field location" :value v1 :agent a :instance i
  :ap i 2 v2 :ap v1 (aseq "fields" v2) loc :match
    :v (not (null loc)) T
    :do loc
    :exit (co "stop next aclosure" :av "type" "error" :av
      "aclosure" ac))

(aclosure ac :attribute "place" :type "1.2"
  :stage "returning value" :value loc :do loc)
```

Семантика операции * определяется следующими замыканиями:

```
(aclosure ac :attribute "place" :type "*1" :do
  (update-push-aclosure ac :av "stage" "returning value")
  (update-push-aclosure ac :av "stage" "evaluating 1"))

(aclosure ac :attribute "place" :type "*1" :stage "evaluating 1"
  :instance i :ap i 1 v1 :do
    (clear-update-eval-aclosure ac :attribute "rvalue"
      :instance v1))

(aclosure ac :attribute "place" :type "*1"
  :stage "returning value" :agent a :value ref
  :ap ref "location" loc :ap a (aseq "location value" loc) v :do
    v)
```

4.2. Семантика r-значений

Семантика r-значений определяется для выражений типов "value", "place", "&1", "&mut1", "&*1", "&mut*1", "*1" и "1+2".

Тип "place" определяется как объединение следующих типов:

```
(typedef "place" (uniont "variable" "1.2"))
```

Семантика значений задается следующим образом:

```
(aclosure ac :attribute "rvalue" :type "value" :do
  (update-push-aclosure ac :av "stage" "returning value")

(aclosure ac :attribute "rvalue" :type "value"
  :stage "returning value" :instance i :do i)
```

Следующие атрибутные замыкания задают семантику типа "place":

```
(aclosure ac :attribute "rvalue" :type "place" :do
  (update-push-aclosure ac :av "stage" "returning value")
  (update-push-aclosure ac :av "stage" "checking location")
  (update-push-aclosure ac :av "stage" "evaluating location"))
```

```
5
6  (aclosure ac :attribute "rvalue" :type "place"
7    :stage "evaluating location" :do
8      (update-eval-aclosure ac :attribute "place"))
9
10 (aclosure ac :attribute "rvalue" :type "place"
11   :stage "checking location" :agent a :value loc
12   :ap a (aseq "borrows" loc) st :ap (car st) "kind" kd :match
13     :av (or (equal kd "free") (equal kd "shared")
14       (equal kd "unique"))
15     :do loc
16     :exit (co "stop next aclosure" :av "type" "error" :av
         "aclosure" ac))
17
18 (aclosure ac :attribute "rvalue" :type "place"
19   :stage "returning value" :agent a :value loc
20   :ap a (aseq "location value" loc) v :do v)
```

Для операции &p семантика определяется следующим образом:

```
1  (aclosure ac :attribute "rvalue" :type "&1" :do
2    (update-push-aclosure ac :av "stage" "returning value")
3    (update-push-aclosure ac :av "stage" "checking location")
4    (update-push-aclosure ac :av "stage" "evaluating location")
5
6  (aclosure ac :attribute "rvalue" :type "&1"
7    :stage "evaluating location" :instance i :ap i 1 v :do
8      (clear-update-eval-aclosure ac :attribute "place"
9        :instance v)))
10
11 (aclosure ac :attribute "rvalue" :type "&1"
12   :stage "checking location" :agent a :value loc
13   :ap a (aseq "borrows" loc) st :ap (car st) "kind" kd :match
```

```
    :av (or (equal kd "free") (equal kd "shared"))
    :do loc
    :exit (co "stop next aclosure" :av "type" "error" :av
      "aclosure" ac)))

(aclosure ac :attribute "rvalue" :type "&1"
  :stage "returning value" :agent a :value loc
  :v (mo "lifetime") lt :ap a (aseq "borrows" loc) bor :do
    (aset a
      :av (aseq "borrows" loc)
        (cons (mo "borrow" :av "lifetime" lt :av "kind" "shared")
          (aseq a "borrows" loc))
      :av (aseq "lifetimes" lt) (list loc))
    (co "reference" :av "location" loc :av "lifetime" lt))
```

Условием выполнимости этой операции является тот факт, что локация для места `p` не должна быть эксклюзивной. Заметим, что мы не утверждаем, что она должна быть свободной или разделяемой, поскольку хотим иметь расширяемую модель.

Семантика для операции `&mut p` определяется аналогичным образом:

```
(aclosure ac :attribute "rvalue" :type "&mut1" :do
  (update-push-aclosure ac :av "stage" "returning value")
  (update-push-aclosure ac :av "stage" "checking location")
  (update-push-aclosure ac :av "stage" "evaluating location")

(aclosure ac :attribute "rvalue" :type "&mut1"
  :stage "evaluating location" :instance i :ap i 1 v :do
    (clear-update-eval-aclosure ac :attribute "place"
      :instance v)))

(aclosure ac :attribute "rvalue" :type "&mut1"
  :stage "checking location" :agent a :value loc :instance i
  :ap a (aseq "borrows" loc) st :ap (car st) "kind" kd :match
```

```
    :av (and (or (equal kd "free") (equal kd "unique"))
      (clear-update-eval-aclosure ac :attribute "mutable"
      :instance (aget i 1))
    :do loc
    :exit (co "stop next aclosure" :av "type" "error" :av
       "aclosure" ac)))

(aclosure ac :attribute "rvalue" :type "&mut1"
  :stage "returning value" :agent a :value loc
  :v (mo "lifetime") lt :ap a (aseq "borrows" loc) bor :do
    (aset a
      :av (aseq "borrows" loc)
        (cons (mo "borrow" :av "lifetime" lt :av "kind" "unique")
          (aseq a "borrows" loc))
      :av (aseq "lifetimes" lt) (list loc))
    (co "reference" :av "location" loc :av "lifetime" lt))
```

Только условия выполнимости другие: локация для места `p` должна быть эксклюзивной, а само место мутабельным. Мутабельность места определяется набором атибутных замыканий для атрибута `"mutable"`.

Для операции `&*p` семантика определяется следующим образом:

```
(aclosure ac :attribute "rvalue" :type "&*1" :do
  (update-push-aclosure ac :av "stage" "returning value")
  (update-push-aclosure ac :av "stage" "checking location")
  (update-push-aclosure ac :av "stage" "evaluating location")

(aclosure ac :attribute "rvalue" :type "&*1"
  :stage "evaluating location" :instance i :ap i 1 v :do
    (clear-update-eval-aclosure ac :attribute "place"
      :instance v)))

(aclosure ac :attribute "rvalue" :type "&*1"
```

```
  :stage "checking location" :agent a :value loc :instance i
  :ap a (aseq "borrows" loc) st :ap (car st) "kind" kd :match
    :av (is-instance (aget i 1) "&mutT1")
    :do loc
    :exit (co "stop next aclosure" :av "type" "error" :av
      "aclosure" ac)))

(aclosure ac :attribute "rvalue" :type "&*1"
  :stage "returning value" :agent a :value loc
  :v (mo "lifetime") lt :ap a (aseq "borrows" loc) bor :do
    (aset a
      :av (aseq "borrows" loc)
        (cons (mo "borrow" :av "lifetime" lt :av "kind" "shared")
          (aseq a "borrows" loc))
      :av (aseq "lifetimes" lt) (list loc))
    (co "reference" :av "location" loc :av "lifetime" lt))
```

Для операции `&mut*p` семантика определяется аналогичным образом:

```
(aclosure ac :attribute "rvalue" :type "&mut*1" :do
  (update-push-aclosure ac :av "stage" "returning value")
  (update-push-aclosure ac :av "stage" "checking location")
  (update-push-aclosure ac :av "stage" "evaluating location")

(aclosure ac :attribute "rvalue" :type "&mut*1"
  :stage "evaluating location" :instance i :ap i 1 v :do
    (clear-update-eval-aclosure ac :attribute "place"
      :instance v)))

(aclosure ac :attribute "rvalue" :type "&mut*1"
  :stage "checking location" :agent a :value loc :instance i
  :ap a (aseq "borrows" loc) st :ap (car st) "kind" kd :match
    :av (and (equal kd "unique")
```

```
      (is-instance (aget i 1) "&mutT1"))
    :do loc
    :exit (co "stop next aclosure" :av "type" "error" :av
      "aclosure" ac)))

(aclosure ac :attribute "rvalue" :type "&mut*1"
  :stage "returning value" :agent a :value loc
  :v (mo "lifetime") lt :ap a (aseq "borrows" loc) bor :do
    (aset a
      :av (aseq "borrows" loc)
        (cons (mo "borrow" :av "lifetime" lt :av "kind" "unique")
          (aseq a "borrows" loc))
      :av (aseq "lifetimes" lt) (list loc))
    (co "reference" :av "location" loc :av "lifetime" lt))
```

Семантика операции * определяется следующими замыканиями:

```
(aclosure ac :attribute "rvalue" :type "*1" :do
  (update-push-aclosure ac :av "stage" "returning value")
  (update-push-aclosure ac :av "stage" "evaluating 1"))

(aclosure ac :attribute "rvalue" :type "*1" :stage "evaluating 1"
  :instance i :ap i 1 v1 :do
    (clear-update-eval-aclosure ac :attribute "rvalue"
      :instance v1))

(aclosure ac :attribute "rvalue" :type "*1"
  :stage "returning value" :agent a :value ref
  :ap ref "location" loc :ap a (aseq "location value" loc) v :do
    v)
```

Операция + определяется следующим образом:

```
(aclosure ac :attribute "rvalue" :type "1+2" :do
  (update-push-aclosure ac :av "stage" "returning value")
```

```
  (update-push-aclosure ac :av "stage" "evaluating 2")
  (update-push-aclosure ac :av "stage" "evaluating 1")

(aclosure ac :attribute "rvalue" :type "1+2"
  :stage "evaluating 1" :instance i :ap i 1 v1 :do
    (clear-update-eval-aclosure ac :attribute "rvalue"
      :instance v1)))

(aclosure ac :attribute "rvalue" :type "1+2"
 :stage "evaluating 2" :value v1 :instance i :ap i 2 v2 :do
    (clear-update-eval-aclosure ac :attribute "rvalue"
      :instance v2 :av "v1" v1)))

(aclosure ac :attribute "rvalue" :type "1+2"
  :stage "returning value" :ap "v1" v1 :value v2 :do (+ v1 v2))
```

Подобным образом определяются и другие бинарные операции.

4.3. Семантика операторов

Семантика операторов определяется для выражений типов "1=2", "let1=2", "letmut1=2" и "{1}". При определении семантики для этих выражений характерными являются стадии "updating agent" и "checking location". Первая определяет, как модифицируется агент, а вторая – условия выполнимости операции.

Операция "1=2" определяется следующим образом:

```
(aclosure ac :attribute "statement" :type "1=2" :do
  (update-push-aclosure ac :av "stage" "updating agent")
  (update-push-aclosure ac :av "stage" "evaluating 2")
  (update-push-aclosure ac :av "stage" "checking location")
  (update-push-aclosure ac :av "stage" "evaluating 1")

(aclosure ac :attribute "statement" :type "1=2"
  :stage "evaluating 1" :instance i :ap i 1 v1 :do
```

```
    (clear-update-eval-aclosure ac :attribute "place"
      :instance v1)))

(aclosure ac :attribute "statement" :type "1=2"
  :stage "checking location" :value loc :agent a
  :ap a (aseq "borrows" loc) st :ap (car st) "kind" kd :match
    :av (or (equal kd "free") (equal kd "unique"))
    :do loc
    :exit (co "stop next aclosure" :av "type" "error" :av
       "aclosure" ac))

(aclosure ac :attribute "statement" :type "1=2"
 :stage "evaluating 2" :value v1 :instance i :ap i 2 v2 :do
    (clear-update-eval-aclosure ac :attribute "rvalue"
      :instance v2 :av "v1" v1)))

(aclosure ac :attribute "statement" :type "1=2"
  :stage "updating agent" :ap "v1" v1 :value v2 :do
    (aset a "location value" v1 v2))
```

Условие выполнимости операции требует, чтобы локация, являющаяся результатом вычисления значения атрибута 1, была или свободной, или эксклюзивной, а место 1 мутабельным.

Семантика операции "let1=2" имеет вид:

```
(aclosure ac :attribute "statement" :type "let1=2" :do
  (update-push-aclosure ac :av "stage" "updating agent")
  (update-push-aclosure ac :av "stage" "evaluating 2"))

(aclosure ac :attribute "statement" :type "let1=2"
 :stage "evaluating 2" :value v1 :instance i :ap i 2 v2 :do
    (clear-update-eval-aclosure ac :attribute "rvalue"
      :instance v2))
```

```
(aclosure ac :attribute "statement" :type "let1=2"
  :stage "updating agent" :instance i :ap i 1 x :value v2
  :v (mo "simple location") loc :do
    (aset a :av (aseq "location" x) loc
      :av (aseq "mutability" x) "immutable"
      :av (aseq "location value" loc) v2
      :av (aseq "borrows" loc)
        (list (mo "borrow" :av "kind" "free"))))
```

Для простоты, мы рассмотрели только случай, когда переменная, являющаяся значением атрибута 1 имеет примитивный тип. Случай структуры определяется аналогичным образом.

Операция "letmut1=2" определяется аналогичным образом:

```
(aclosure ac :attribute "statement" :type "letmut1=2" :do
  (update-push-aclosure ac :av "stage" "updating agent")
  (update-push-aclosure ac :av "stage" "evaluating 2"))

(aclosure ac :attribute "statement" :type "letmut1=2"
 :stage "evaluating 2" :value v1 :instance i :ap i 2 v2 :do
    (clear-update-eval-aclosure ac :attribute "rvalue"
      :instance v2))

(aclosure ac :attribute "statement" :type "letmut1=2"
  :stage "updating agent" :instance i :ap i 1 x :value v2
  :v (mo "simple location") loc :do
    (aset a :av (aseq "location" x) loc
      :av (aseq "mutability" x) "mutable"
      :av (aseq "location value" loc) v2
      :av (aseq "borrows" loc)
        (list (mo "borrow" :av "kind" "free"))))
```

Отличается только модификация агента.

Особенностью семантики блока является его возможность определять локальные времена жизни и исключать их при выходе из блока:

```
(aclosure ac :attribute "statement" :type "{1}" :agent a
  :ap a "lifetimes" lfs :do
    (update-push-aclosure ac :av "stage" "restoring lifetime"
      :av "lifetime list" (attributes lfs))
    (update-push-aclosure ac :av "stage" "evaluating 1"))

(aclosure ac :attribute "statement" :type "{1}"
  :stage "evaluating 1" :instance i :ap i 1 stl :do
    (update-eval-aclosure ac :stage "body iteration"
      :av "current" 0 :av "statements" stl
      :av "bound" (length stl)))

(aclosure ac :attribute "statement" :type "{1}"
  :stage "body iteration" :ap "current" k :ap "statements" stl
  :ap "bound" n :match
    :v (< k n) T
    :do
      (update-eval-aclosure ac :av "current" (+ k 1))
      (clear-update-eval-aclosure ac :instance (nth k stl)))

(aclosure ac :attribute "statement" :type "{1}"
  :stage "restoring lifetime" :agent a :ap "lifetime list" lfl
  :match :v (mo) lfs
    (dolist (lf lfl) (aset lfs lf (aget a "lifetimes" lf)))
    (aset a "lifetimes" lfs)
```

5. Верификация правил безопасности на примерах

В данном разделе рассматривается работа интерпретатора ABML на классических сценариях языка Rust. Особое внимание уделяется тому, как динамическая семантика агента воспроизводит статические проверки компилятора, обеспечивая безопасность работы с па-

мятью.

5.1. Конфликты заимствования

Основная концепция безопасности Rust заключается в запрете одновременного существования разделяемого доступа (чтения) и возможности модификации данных. Это предотвращает неопределенное поведение и гонки данных. Рассмотрим пример, нарушающий эти правила:

```
let mut x = 5;
let y = &x;        // Shared borrow (разделяемое заимствование)
*x = 10;           // ОШИБКА: x уже заимствован переменной y
```

Выполним интерпретацию этих операций в модели на ABML.

Первая инструкция устанавливает следующие связи в атрибутах агента:

```
loc1 := (mo "simple location")

(aget a "location" x) = loc1
(aget a "mutability" x) = "mutable"
(aget a "location value" loc1) = 5
(aget a "borrows" loc1) = (list (mo "borrow" :av "kind" "free"))
```

где `loc1` – новая локация.

Вторая инструкция модифицирует агента следующим образом:

```
loc1 := (mo "simple location")
loc2 := (mo "simple location")
lt1 := (mo "lifetime")

(aget a "location" x) = loc1
(aget a "location" y) = loc2
(aget a "mutability" x) = "mutable"
(aget a "mutability" y) = "immutable"
(aget a "location value" loc1) = 5
(aget a "location value" loc2) =
  (co "reference" :av "location" loc1 :av "lifetime" lt1)
```

```
(aget a "borrows" loc1) =
  (list (mo "borrow" :av "lifetime" lt1 :av "kind" "shared")
    (mo "borrow" :av "kind" "free"))
(aget a "borrows" loc2) = (list (mo "borrow" :av "kind" "free"))
(aget a "lifetimes" lt1) = (list loc1)
```

где `loc2` – новая локация, и `lt1` – новое время жизни.

Третья инструкция начинает с вычисления `*x`. В этом случае `x` вычисляется как `rvalue` и возвращает число `5`, а вычисление `*x` требует ссылки. Поэтому вычисление операционной семантики завершается с ошибкой.

5.2. Частичное заимствование структур

Одним из ключевых преимуществ онтологического подхода является возможность точного моделирования частичного доступа к компонентам сложных данных. В отличие от систем с «плоской» памятью, блокирующих объект целиком, иерархическая модель ABML позволяет агенту анализировать доступ на уровне отдельных полей.

```
let mut point = Point { x: 1, y: 2 };
let r = &point.x;   // Заимствуем только поле .x
point.y = 10;       // OK: поле .y доступно для записи
```

В данном примере поле `x` заимствовано для чтения, что запрещает его изменение. Однако поле `y` остается свободным и может быть изменено.

Выполним интерпретацию этих операций в модели на ABML.

Первая инструкция устанавливает следующие связи в атрибутах агента:

```
locx := (mo "field location")
locy := (mo "field location")
locpoint := (mo "struct location" :av "x" locx :av "y" locy)

(aget a "location" point) = locpoint
(aget a "mutability" point) = "mutable"
(aget a "location value" locx) = 1
(aget a "location value" locy) = 2
(aget a "location value" loc_point) =
```

```
  (mo "struct location" :av "x" locx :av "y" locy)
(aget a "borrows" locx) = (list (mo "borrow" :av "kind" "free"))
(aget a "borrows" locy) = (list (mo "borrow" :av "kind" "free"))
(aget a "borrows" locpoint) = (list (mo "borrow" :av "kind"
  "free"))
```

Вторая инструкция модифицирует агента следующим образом:

```
locx := (mo "field location")
locy := (mo "field location")
loc_point := (mo "struct location" :av "x" locx :av "y" locy)
locr := (mo "field location")
lt1 := (mo "lifetime")

(aget a "location" point) = locpoint
(aget a "location" r) = locr
(aget a "mutability" point) = "mutable"
(aget a "mutability" r) = "immutable"
(aget a "location value" locx) = 1
(aget a "location value" locy) = 2
(aget a "location value" loc_point) =
  (mo "struct location" :av "x" locx :av "y" locy)
(aget a "location value" locr) =
  (co "reference" :av "location" locx :av "lifetime" lt1)
(aget a "borrows" locx) =
  (list (mo "borrow" :av "lifetime" lt1 :av "kind" "shared")
    (mo "borrow" :av "kind" "free"))
(aget a "borrows" locy) = (list (mo "borrow" :av "kind" "free"))
(aget a "borrows" locpoint) = (list (mo "borrow" :av "kind"
  "free"))
(aget a "lifetimes" lt1) = (list locx)
```

Третья инструкция также выполняется, так как заимствование наложено только на `locx`, а не на `locpoint` целиком:

```
locx := (mo "field location")
locy := (mo "field location")
loc_point := (mo "struct location" :av "x" locx :av "y" locy)
locr := (mo "field location")
lt1 := (mo "lifetime")

(aget a "location" point) = locpoint
(aget a "location" r) = locr
(aget a "mutability" point) = "mutable"
(aget a "mutability" r) = "immutable"
(aget a "location value" locx) = 1
(aget a "location value" locy) = 10
(aget a "location value" loc_point) =
  (mo "struct location" :av "x" locx :av "y" locy)
(aget a "location value" locr) =
  (co "reference" :av "location" locx :av "lifetime" lt1)
(aget a "borrows" locx) =
  (list (mo "borrow" :av "lifetime" lt1 :av "kind" "shared")
    (mo "borrow" :av "kind" "free"))
(aget a "borrows" locy) = (list (mo "borrow" :av "kind" "free"))
(aget a "borrows" locpoint) = (list (mo "borrow" :av "kind"
  "free"))
(aget a "lifetimes" lt1) = (list locx)
```

Этот пример наглядно демонстрирует точность онтологического моделирования: агент «понимает», что заимствование части объекта не эквивалентно блокировке всего объекта, что полностью соответствует семантике разделенного заимствования (split borrowing) в Rust.

6. Родственные работы

Исследования в области формальной семантики языков программирования, и в особенности языка Rust, в последние годы привлекают значительное внимание научного сооб-

щества. Это связано с нетривиальной моделью памяти Rust, основанной на концепциях владения, заимствования и строгих гарантиях отсутствия гонок данных. В результате появилось множество работ, направленных на формализацию как статических, так и динамических аспектов языка.

Одной из первых работ, целенаправленно описывающих динамическое поведение Rust, является исполняемая операционная семантика RustSEM [3]. В данной работе авторы предлагают формальную модель исполнения программ Rust, в которой явно представлены механизмы владения и заимствования. Семантика задается в виде системы переходов состояний и ориентирована на воспроизведение поведения реальных программ, включая ситуации, приводящие к ошибкам доступа к памяти. Подход RustSEM демонстрирует возможность динамической проверки корректности работы с заимствованиями, однако модель в значительной степени опирается на плоское представление памяти.

Схожую цель преследует работа KRust [11], в которой формальная семантика Rust реализована в рамках K-фреймворка. Использование K позволяет автоматически получать исполняемый интерпретатор и инструменты анализа на основе формального описания семантики. Авторы показывают, что предложенная модель корректно воспроизводит основные элементы языка, включая перенос владения (move), заимствования и мутабельность. Семантика KRust была сопоставлена с тестами официального компилятора Rust, что подтверждает ее практическую применимость.

Отдельное направление исследований связано с формализацией заимствований через символические модели. В работах по проверке корректности заимствований посредством символьной семантики [6, 7] предлагается формализм LLBC (Low-Level Borrow Calculus), который служит промежуточным уровнем между высокоуровневым Rust и низкоуровневыми моделями памяти. Авторы доказывают корректность символьной семантики по отношению к операционной, что позволяет использовать модель для формальной верификации свойств программ, связанных с безопасностью памяти.

Дальнейшим развитием этого направления является инструмент Aeneas [8], ориентированный на верификацию программ Rust путем перевода в функциональные представления. В данной работе операционная семантика Rust редуцируется к чистой семантике, в которой управление памятью и адресами заменяется абстрактными понятиями владения и займов. Такой подход облегчает доказательство функциональной корректности, но абстрагируется от многих деталей реального исполнения.

Фундаментальной работой в области формального обоснования Rust является проект RustBelt [15]. В нем авторы вводят формальный язык λRust и задают его операционную семантику, что позволяет строго доказать безопасность ключевых механизмов Rust, включая отсутствие использования после освобождения (use-after-free) и гонок данных. В отличие от исполняемых семантик, RustBelt ориентирован прежде всего на доказательство теоретических свойств языка, а не на моделирование конкретных сценариев выполнения программ. В RustBelt предлагается логическая семантика для подмножества Rust и доказываются ключевые свойства безопасности памяти [10, 12, 15]. Хотя исходная версия RustBelt была опубликована ранее, последующие работы существенно расширили этот подход, включая поддержку relaxed memory и unsafe-кода [1, 12, 13].

Развитие RustBelt привело к созданию RustHornBelt и RefinedRust, которые объединяют логические и типовые методы верификации и обеспечивают высокую степень автоматизации доказательств [9, 14, 16]. Эти работы демонстрируют, как операционная семантика Rust может быть связана с логическими моделями и доказательными системами.

Параллельно развиваются практико-ориентированные средства верификации, такие как Verus и Flux, использующие расширенные типовые системы и SMT-решатели для доказательства свойств программ [5, 18, 19]. Эти инструменты опираются на формальные семантические модели Rust, хотя и не всегда задают их явно в операционной форме.

Отдельный класс исследований посвящен aliasing-моделям и динамической семантике заимствований, включая Stacked Borrows и его расширения, которые уточняют допустимые сценарии доступа к памяти во время выполнения [4, 17]. Эти модели оказывают существенное влияние на современные формализации Rust и интерпретацию unsafe-кода.

Существуют также работы, фокусирующиеся на концептуальном и когнитивном анализе модели владения Rust. Так, в [2] предлагается концептуальная модель ownership-типов, которая формализует интуитивные представления о владении и заимствовании и сопоставляет их с реальными правилами языка. Хотя данная работа не задает операционную семантику напрямую, она вносит существенный вклад в понимание связи между статическими и динамическими аспектами Rust.

На этом фоне предложенная в настоящей статье операционная семантика выражений Rust на языке ABML занимает промежуточное положение между теоретическими и практико-ориентированными подходами. В отличие от большинства существующих моделей, она использует онтологическое представление вычислительного состояния и иерар-

хическую модель памяти, что позволяет естественно выразить частичное заимствование и динамическую проверку конфликтов доступа. Таким образом, работа дополняет существующие исследования, предлагая альтернативный, онтологически обогащенный способ формализации семантики Rust.

7. Заключение

В статье была представлена операционная семантика выражений языка Rust, формализованная с использованием онтологического и атрибутно-ориентированного подхода, реализованного в языке ABML. Предложенная модель ориентирована на точное воспроизведение динамического поведения программ с учетом ключевых гарантий безопасности памяти, характерных для Rust.

Одним из основных результатов работы является введение иерархической модели памяти, в которой локации могут представлять как целые структуры, так и их отдельные поля. Такая организация памяти позволяет корректно моделировать частичное заимствование и отражает реальные правила доступа к данным, применяемые в языке Rust. В сочетании с метаданными живучести, мутабельности и активных заимствований эта модель обеспечивает строгую динамическую проверку конфликтов.

Разработанная операционная семантика описана в виде исполняемых правил, что отличает ее от чисто декларативных формализаций. Это делает возможным использование модели не только для теоретического анализа, но и для экспериментального исполнения программ, трассировки вычислений и исследования граничных случаев поведения механизмов ownership и borrow checking.

Сравнение с предыдущими работами, использующими ABML для описания семантики других языков и конструкций, показывает универсальность выбранного подхода. Онтологическое представление синтаксических и семантических сущностей позволяет постепенно расширять модель, добавляя новые конструкции языка Rust без радикального пересмотра уже существующих определений.

В перспективе предложенная семантика может быть расширена для поддержки более сложных элементов языка, таких как функции, замыкания, обобщенные типы и параллельные вычисления. Кроме того, она может служить основой для построения формальных инструментов анализа и верификации программ на Rust, а также для сопоставления динамической семантики с результатами статического анализа компилятора.

В целом, работа демонстрирует, что онтологический и атрибутно-ориентированный подход в сочетании с ABML является эффективным средством формализации языков программирования с развитой моделью безопасности памяти и открывает новые возможности для исследований в области формальных семантик.

Список литературы

1. Batty M. J. The C11 and C++ 11 concurrency model : Ph. D. thesis ; University of Cambridge, UK. — 2015.
2. Crichton W., Gray G., Krishnamurthi S. A grounded conceptual model for ownership types in rust // Proceedings of the ACM on Programming Languages. — 2023. — Vol. 7, no. OOPSLA2. — P. 1224–1252.
3. An Executable Operational Semantics for Rust with the Formalization of Ownership and Borrowing / Kan S., Chen Z., Sanan D., Lin S.-W., and Liu Y. // arXiv preprint arXiv:1804.07608. — 2018.
4. Exploring C semantics and pointer provenance / Memarian K., Gomes V. B., Davis B., Kell S., Richardson A., Watson R. N., and Sewell P. // Proceedings of the ACM on Programming Languages. — 2019. — Vol. 3, no. POPL. — P. 1–32.
5. Flux: Liquid types for rust / Lehmann N., Geller A. T., Vazou N., and Jhala R. // Proceedings of the ACM on Programming Languages. — 2023. — Vol. 7, no. PLDI. — P. 1533–1557.
6. Ho S., Fromherz A., Protzenko J. Sound Borrow-Checking for Rust via Symbolic Semantics // Proceedings of the ACM on Programming Languages. — 2024. — Vol. 8, no. ICFP. — P. 426–454.
7. Ho S., Fromherz A., Protzenko J. Sound Borrow-Checking for Rust via Symbolic Semantics (Long Version) // arXiv preprint arXiv:2404.02680. — 2024.
8. Ho S., Protzenko J. Aeneas: Rust verification by functional translation // Proceedings of the ACM on Programming Languages. — 2022. — Vol. 6, no. ICFP. — P. 711–741.
9. Iris from the ground up / Jung R., Krebbers R., Jourdan J.-H., Bizjak A., Birkedal L., and Dreyer D. // Submitted to JFP. — 2017.
10. Jung R. Understanding and evolving the Rust programming language : Phd dissertation ; Saarländische Universitäts-und Landesbibliothek. — 2020.

11. Krust: A formal executable semantics of rust / Wang F., Song F., Zhang M., Zhu X., and Zhang J. // 2018 International Symposium on Theoretical Aspects of Software Engineering (TASE) / IEEE. — 2018. — P. 44–51.
12. MoSeL: A general, extensible modal framework for interactive proofs in separation logic / Krebbers R., Jourdan J.-H., Jung R., Tassarotti J., Kaiser J.-O., Timany A., Charguéraud A., and Dreyer D. // Proceedings of the ACM on Programming Languages. — 2018. — Vol. 2, no. ICFP. — P. 1–30.
13. A promising semantics for relaxed-memory concurrency / Kang J., Hur C.-K., Lahav O., Vafeiadis V., and Dreyer D. // ACM SIGPLAN Notices. — 2017. — Vol. 52, no. 1. — P. 175–189.
14. Refinedrust: A type system for high-assurance verification of Rust programs / Gäher L., Sammler M., Jung R., Krebbers R., and Dreyer D. // Proceedings of the ACM on Programming Languages. — 2024. — Vol. 8, no. PLDI. — P. 1115–1139.
15. RustBelt: Securing the foundations of the Rust programming language / Jung R., Jourdan J.-H., Krebbers R., and Dreyer D. // Proceedings of the ACM on Programming Languages. — 2017. — Vol. 2, no. POPL. — P. 1–34.
16. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code / Matsushita Y., Denis X., Jourdan J.-H., and Dreyer D. // Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. — 2022. — P. 841–856.
17. Stacked borrows: an aliasing model for Rust / Jung R., Dang H.-H., Kang J., and Dreyer D. // Proceedings of the ACM on Programming Languages. — 2019. — Vol. 4, no. POPL. — P. 1–32.
18. Verus: Verifying rust programs using linear ghost types / Lattuada A., Hance T., Cho C., Brun M., Subasinghe I., Zhou Y., Howell J., Parno B., and Hawblitzel C. // Proceedings of the ACM on Programming Languages. — 2023. — Vol. 7, no. OOPSLA1. — P. 286–315.
19. Verus: verifying Rust programs using linear ghost types (extended version) / Lattuada A., Hance T., Cho C., Brun M., Subasinghe I., Zhou Y., Howell J., Parno B., and Hawblitzel C. // arXiv preprint arXiv:2303.05491. — 2023.
20. Ануреев И.С. Операционная семантика операторов передачи управления в языке C на языке ABML // Системная информатика. — 2025. — no. 29. — P. 159–188.
21. Ануреев И.С. Язык спецификации дискретных динамических систем, ориентирован-

ных на знания, структурированные в онтологиях // Системная информатика. — 2025. — no. 29. — P. 137–158.