Языки программирования. Семантика и системы типов

Николай Кудасов

17 февраля 2024

Проект. Этап 1

1.1 Кратко о проекте

На этом этапе проекта вам необходимо реализовать программу, осуществляющую проверку типов в исходном коде на простом функциональном типизированном фрагменте языка Stella[1]. А именно, ваша реализация должна поддерживать следующее:

- ядро языка Stella (логические типы, натуральные числа, функции)
- `let`-связывания
- приписывание типа (аннотация)
- единичный тип
- пары и записи
- типы-суммы и варианты
- рекурсия общего вида и оператор неподвижной точки
- встроенные списки

[1] https://fizruk.github.io/stella/

1.2 Требования к реализации

Основная цель проекта — реализовать *Тайпчекер*, программу реализующую проверку типов для модельного языка Stella. Синтаксический разбор (парсинг) и структура синтаксического дерева может быть переиспользована, однако сам алгоритм проверки типов и вспомогательные определения должны быть реализованы каждым студентом индивидуально.

Реализация проекта допускается на любом языке программирования, по предварительному согласованию с преподавателем. Тем не менее, рекомендуется использовать языки, поддерживаемые инструментом BNF Converter[2] или ANTLR[3], поскольку для этим инструментов существует готовая грамматика, по которой можно сгенерировать необходимую инфраструктуру проекта.

1.2.1 Формат ввода и вывода

Тайпчекер должен принимать исходный код программы на языке Stella из стандартного потока ввода (`stdin`) и выводить результат проверки типов в стандартные потоки вывода (`stdout`) и ошибок (`stderr`). Если исходный код не содержит ошибок типизации, программа должа завершаться с нулевым кодом выхода. Иначе — с любым ненулевым.

При наличии ошибок типизации, первая такая ошибка должна быть напечатана в стандартном потоке ошибок (`stderr`). Сообщение об ошибке должно содержать человеко-читаемый текст с описанием ошибки, а также код типа ошибки. Ниже приведён пример программы с ошибой типизации и пример текста ошибки:

```
// программа на Stella
language core;

fn increment_twice(n : Nat) -> Nat {
  return succ(succ(n))
}

fn main(n : Nat) -> Nat {
  return increment_twice( if Nat::iszero(n) then false else true )
}

// сообщение об ошибке
ERROR_UNEXPECTED_TYPE_FOR_EXPRESSION:
  ожидается тип
    Nat
  но получен тип
    Bool
  для выражения
    if Nat::iszero (n) then false else true
```

1.2.2 Лексический и синтаксический разбор

Для реализации лексического и синтаксического разбора предлагается использовать готовые грамматики языка Stella вместе с генераторами парсеров BNFC[4] или ANTLR[5].

BNFC поддерживает генерацию для Haskell, Agda, C, C++, Java (через ANTLR) и OCaml. Экспериментальные генераторы существуют для TypeScript и Dart. BNFC является надстройкой, использует другие генераторы парсеров внутри и предоставляет также более качественное абстрактное синтаксическое дерево и методы для преобразования синтаксиса в текст (претти-принтинг).

ANTLR поддерживает генерацию для Java, C#, Python 3, JavaScript, TypeScript, Go, C++, Swift, PHP и Dart.

[2] https://bnfc.digitalgrammars.com
[3] https://www.antlr.org
[4] https://bnfc.digitalgrammars.com
[5] https://www.antlr.org

1.2.3 Требования к проверке типов

Реализация *Тайпчекера* **должна** поддерживать следующие синтаксические конструкции языка Stella:

1. для ядра языка:
   (a) Программа (модуль): `AProgram`
   (b) Объявление функции (с ровно одним параметром): `DeclFun`, `AParamDecl`, `SomeReturnType`
   (c) Логические выражения: `TypeBool`, `ConstTrue`, `ConstFalse`, `If`
   (d) Выражения с натуральными числами: `TypeNat`, `ConstInt(0)`, `Succ`, `IsZero`, `NatRec`
   (e) Функции как значения первого класса (только с одним параметром): `TypeFun`, `Abstraction`, `AParamDecl`, `Application`
   (f) Переменные (неизменяемые): `Var`
2. для расширения `#unit-type`: `TypeUnit`, `ConstUnit`
3. для расширений `#pairs` и `#tuples`: `TypeTuple`, `Tuple`, `DotTuple`
4. для расширения `#records`: `TypeRecord`, `Record`, `DotRecord`
5. для расширения `#let-bindings`: `Let`, `APatternBinding`, `PatternVar`
6. для расширения `#type-ascriptions`: `TypeAsc`
7. для расширения `#sum-types`: `TypeSum`, `Inl`, `Inr`, `Match`, `AMatchCase`, `PatternInl`, `PatternInr`, `PatternVar`
8. для расширения `#lists`: `TypeList`, `List`, `ConsList`, `Head`, `Tail`, `IsEmpty`
9. для расширения `#variants`: `TypeVariant`, `AVariantFieldType`, `SomeTyping`, `Variant`, `SomeExprData`, `PatternVariant`, `SomePatternData`
10. для расширения `#fixpoint-combinator`: `Fix`

При возникновении ошибки типизации, *Тайпчекер* должен завершиться с ненулевым кодом выхода и распечатать в стандартном потоке ошибок сообщение, содержащее описание проблемы и код ошибки. Для данного задания необходимо использовать один из следующих кодов ошибки:

1. `ERROR_MISSING_MAIN` — в программе отсутствует функция `main`;
2. `ERROR_UNDEFINED_VARIABLE` — в выражении содержится необъявленная переменная;
3. `ERROR_UNEXPECTED_TYPE_FOR_EXPRESSION` — тип выражения отличается от ожидаемого; эта ошибка должна возникать только если ни одна из более точных ошибок (ниже) не возникла раньше;
4. `ERROR_NOT_A_FUNCTION` — при попытке применить (`Application`) выражение к аргументу или передать в комбинатор неподвижной точки (`Fix`), выражение оказывается не функцией; ошибка должна возникать до проверки типа аргумента;
5. `ERROR_NOT_A_TUPLE` — при попытке извлечь компонент кортежа (`DotTuple`) из выражения, выражение оказывается не кортежем (`TypeTuple`);
6. `ERROR_NOT_A_RECORD` — при попытке извлечь поле записи (`DotRecord`) из выражения, выражение оказывается не записью (`TypeRecord`);
7. `ERROR_NOT_A_LIST` — при попытке извлечь голову (`Head`), извлечь хвост (`Tail`) или проверить список на наличие элементов (`IsEmpty`), соответствующий аргумент оказывается не списком (`TypeList`);

8. `ERROR_UNEXPECTED_LAMBDA` — в процессе проверки типов анонимная функция (`Abstraction`) проверяется с не функциональным типом (`TypeFun`); ошибка должна возникать до проверки типа самой анонимной функции;
9. `ERROR_UNEXPECTED_TYPE_FOR_PARAMETER` — в процессе проверки параметра анонимной функции (`AParamDecl`) указанный тип параметра отличается от ожидаемого; ошибка должно возникать до проверки тела анонимной функции;
10. `ERROR_UNEXPECTED_TUPLE` — в процессе проверки типов кортеж (`Tuple`) проверяется с типом отличным от типа кортежа (`TypeTuple`); ошибка должна возникать до проверки типа самого кортежа;
11. `ERROR_UNEXPECTED_RECORD` — в процессе проверки типов запись (`Record`) проверяется с типом отличным от типа записи (`TypeRecord`); ошибка должна возникать до проверки типа самой записи;
12. `ERROR_UNEXPECTED_LIST` — в процессе проверки типов список (`List` или `ConsList`) проверяется с типом отличным от типа списка (`TypeList`); ошибка должна возникать до проверки типа самого списка;
13. `ERROR_UNEXPECTED_INJECTION` — в процессе проверки типов инъекция (`Inl` или `Inr`) проверяется с типом отличным от типа-суммы (`TypeSum`); ошибка должна возникать до проверки типа самой инъекции;
14. `ERROR_MISSING_RECORD_FIELDS` — в записи (`Record`) отсутствуют ожидаемые поля;
15. `ERROR_UNEXPECTED_RECORD_FIELDS` — в записи (`Record`) присутствуют поля, которых нет в ожидаемом типе записи;
16. `ERROR_UNEXPECTED_FIELD_ACCESS` — попытка извлечь отсутствующее поле записи (`DotRecord`);
17. `ERROR_TUPLE_INDEX_OUT_OF_BOUNDS` — попытка извлечь отсутствующий компонент кортежа (`DotTuple`);
18. `ERROR_UNEXPECTED_TUPLE_LENGTH` — длина кортежа (`Tuple`) не соответствует ожидаемой длине;
19. `ERROR_AMBIGUOUS_SUM_TYPE` — тип инъекции (`Inl` или `Inr`) невозможно определить (в данном контексте отсутсвует ожидаемый тип-сумма);
20. `ERROR_AMBIGUOUS_LIST` — тип списка (`List` или `ConsList`) невозможно определить (в данном контексте отсутсвует ожидаемый тип списка);
21. `ERROR_ILLEGAL_EMPTY_MATCHING` — `match`-выражение с пустым списком альтернатив;
22. `ERROR_NONEXHAUSTIVE_MATCH_PATTERNS` — не все образцы перечислены в `match`-выражении (`inl` и `inr` для типа-суммы, все возможные теги для типа варианта)
23. `ERROR_UNEXPECTED_PATTERN_FOR_TYPE` — образец в `match`-выражении не соответствует типу разбираемого выражения;

1.2.4 Необязательные расширения

Следующие расширения могут быть реализованы за дополнительные баллы:

1. `#natural-literals`: `ConstInt`
2. `#nested-function-declarations`: `DeclFun`
3. `#nullary-functions` и `#multiparameter-functions`:
   - узлы синтаксического дерева: `DeclFun`, `Abstraction`, `Application`
   - коды ошибок:

(a) `ERROR_INCORRECT_ARITY_OF_MAIN` — функция `main` объявлена с n параметрами, где $n \neq 1$;
(b) `ERROR_INCORRECT_NUMBER_OF_ARGUMENTS` — вызов функции (`Application`) происходит с некорректным количеством аргументов;
(c) `ERROR_UNEXPECTED_NUMBER_OF_PARAMETERS_IN_LAMBDA` — количество параметров анонимной функции (`Abstraction`) не совпадает с ожидаемым количеством параметров;

4. `#structural-patterns` (расширенные и вложенные образцы в `let`-связываниях и `match`-выражениях):
   - `PatternTuple`
   - `PatternRecord`, `ALabelledPattern`
   - `PatternList`, `PatternCons`
   - `PatternInt`, `PatternSucc`
   - `PatternFalse`, `PatternTrue`,
   - `PatternUnit`
5. `#nullary-variant-labels` (теги без данных в вариантах):
   - узлы синтаксического дерева: `NoTyping`, `NoExprData`, `NoPatternData`
   - коды ошибок:
     (a) `ERROR_UNEXPECTED_DATA_FOR_NULLARY_LABEL` — вариант (`Variant`) содержит данные (`SomeExprData`), хотя ожидается тег без данных (`NoTyping`);
     (b) `ERROR_MISSING_DATA_FOR_LABEL` — вариант (`Variant`) не содержит данные (`NoExprData`), хотя ожидается тег с данными (`SomeTyping`);
     (c) `ERROR_UNEXPECTED_NON_NULLARY_VARIANT_PATTERN` — образец варианта (`PatternVariant`) содержит тег с данными (`SomePatternData`), хотя в типе разбираемого выражения этот тег указан без данных (`NoTyping`);
     (d) `ERROR_UNEXPECTED_NULLARY_VARIANT_PATTERN` — образец варианта (`PatternVariant`) содержит тег без данных (`NoPatternData`), хотя в типе разбираемого выражения этот тег указан с данными (`SomeTyping`);
6. `#letrec-bindings` и `#letrec-many-bindings`: `LetRec` и `Let`

1.3 Описание возможностей языка Stella

Stella — это язык программирования, созданный специально для практики реализации алгритмов проверки типов. Ядро языка выполнено в минималистичном стиле и семантически соответствует простому типизированному λ-исчислению с логическими и арифметическими выражениями. Поверх ядра, Stella поддерживает множество расширений, позволяющих постепенно добавлять в язык синтаксические и другие возможности.

1.3.1 Ядро языка Stella

Ядро языка Stella — это простой типизированный функциональный язык программирования с Rust-подобным синтаксисом. Например, рассмотрим следующую программу:

```
// пример программы на ядре Stella
language core;

fn increment_twice(n : Nat) -> Nat {
  return succ(succ(n))
}

fn main(n : Nat) -> Nat {
  return increment_twice(n)
}
```

Построчное объяснение программы:

1. комментарий;
2. объявление о том, что мы используем ядро языка;
3. пустая строка;
4. объявление функции `increment_twice` с параметром `n` типа `Nat` и возвращаемым типом `Nat`; в ядре Stella все функции имеют ровно один параметр;
5. тело функции `increment_twice` (всегда выглядит как `return <выражение>`), где мы возвращаем выражение `succ(succ(n))`; `succ{n}` означает $n + 1$;
6. завершение объявления функции;
7. пустая строка;
8. объявление функции `main` с параметром `n` типа `Nat` и возвращаемым типом `Nat`;
9. тело функции `main`, где мы возвращаем выражение `increment_twice(n)`;
10. завершение объявления функции;

В общем случае, программа на ядре Stella состоит из последовательности объявлений функций одного аргумента, одна из которых должна быть функцией `main`:

```
language core;

fn function_1(x : <тип аргумента>) -> <тип результата> { return <выражение> }
fn function_2(y : <тип аргумента>) -> <тип результата> { return <выражение> }
...
fn main(arg : <тип аргумента>) -> <тип результата> { return <выражение> }
```

Логические выражения Логические выражения представлены в Stella типом `Bool` и следующими выражениями:

- `true` — значение ИСТИНА;
- `false` — значение ЛОЖЬ;
- `if` $\texttt{e}_1$ `then` $\texttt{e}_2$ `else` $\texttt{e}_3$ — условное выражение.

Семантика и правила типизации следуют традиционному определению [1, §8].

Натуральные числа Натуральные числа представлены в Stella типом `Nat` и следующими выражениями:

- `0` — константа НОЛЬ;
- `succ(e)` — инкремент (конструктор числа $e + 1$);
- `Nat::pred(e)` — декремент;
- `Nat::iszerp(e)` — проверка на НОЛЬ;
- `Nat::rec(n, z, s)` — примитивная рекурсия для натуральных чисел:
  - `n` — натуральное число, определяющее кол-во итераций
  - `z` — начальное значение (любого типа)
  - `s` — функция шага рекурсии

Семантика и правила типизации (кроме `Nat::rec(n, z, s)`) следуют традиционному определению [1, §8]. Для примитивной рекурсии, правило типизации следующее:

$$\frac{\Gamma \vdash \texttt{n : Nat} \quad \Gamma \vdash \texttt{z : T} \quad \Gamma \vdash \texttt{s : Nat -> (T -> T)}}{\Gamma \vdash \texttt{Nat::rec(n, z, s) : T}}\ \text{T-NatRec}$$

Функции Функции в ядре являются значениями первого класса, т.е. могут выступать в качестве аргументов и возвращаемых значений других функций. Синтаксически функции представлены

1. типом функций: `fn(T1) -> T2`;
2. именованными определениями:

   ```
   fn <имя>(<параметр> : <тип параметра>) -> <тип результата> { return <выражение> }
   ```

3. анонимными функциями:

   ```
   fn(<параметр> : <тип параметра>) { return <выражение> }
   ```

4. применением функции к аргументу: `e1(e2)` (скобки вокруг аргумента обязательны);

Пример функции над натуральными числами:

```
fn plus2(n : Nat) -> Nat {
  return succ(succ(n))
}
```

Пример логической функции:

```
fn Bool::not(b : Bool) -> Bool {
  return
    if b then false else true
}
```

Пример функции высшего порядка, использующей анонимную функции в теле:

```
fn twice(f : fn(Bool) -> Bool) -> (fn(Bool) -> Bool) {
  return fn(x : Bool) {
    return f(f(x))
  }
}
```

Пример использования функции высшего порядка:

```
fn main(b : Bool) -> Bool {
  return twice(Bool::not)(b)
}
```

Семантика и правила типизации для функций следуют традиционному определению [1, §9].

1.3.2 Единичный тип (`#unit-type`)

Единичный тип представлен в Stella типом `Unit` и константным выражением `unit`. Если действует расширение `#structural-patterns`, то также имеется образец `unit`.

Семантика и правила типизации для функций следуют традиционному определению [1, §11.2].

1.3.3 Пары и кортежи (`#pairs`, `#tuples`)

Пары и кортежи представлены в Stella типами `{<тип1>, <тип2>, ..., <типN>}` и следующими выражениями:

1. `{<выражение1>, ..., <выражениеN>}` — кортеж длины N ($N \geq 0$)
2. `e.`i — доступ к компоненте кортежа с индексом i (компоненты кортежа индексируются с 1, т.е. $1 \leq i \leq n$, где n — длина кортежа)

Если действует расширение `#structural-patterns`, то также имеются образцы

```
{<образец1>, ..., <образецN>}
```

Семантика и правила типизации для пар и кортежей следуют традиционному определению [1, §11.6–11.7].

1.3.4 Записи (`#records`)

Записи представлены в Stella типами `{<тег1> : <тип1>, ..., <тегN> : <типN>}` и следующими выражениями:

1. `{<тег1> = <выражение1>, ..., <тегN> = <выражениеN>}` — запись с N полями ($N \geq 0$)
2. `e.<тег>` — доступ к полю `<тег>` записи

Если действует расширение `#structural-patterns`, то также имеются образцы

`{<тег1> = <образец1>, ..., <тегN> = <образецN>}`

Семантика и правила типизации для записей следуют традиционному определению [1, §11.8].

1.3.5 `let`-связывания

`let`-связывания представлены в Stella выражениями

`let <переменная> = <выражение> in <выражение>`

Если действует расширение `#let-patterns`, то вместо выражение обобщается до

`let <образец> = <выражение> in <выражение>`

Семантика и правила типизации для `let`-связываний следуют традиционному определению [1, §11.5].

1.3.6 Приписывание типа (`#type-ascriptions`)

Приписывание типа представлено в Stella выражениями

`<выражение> as <тип>`

Семантика и правила типизации для приписывания типа следуют традиционному определению [1, §11.4].

1.3.7 Типы-суммы (`#sum-types`)

Типы-суммы представлены в Stella типами `<тип> + <тип>` и следующими выражениями:

1. `inl(<выражение>)` — левая инъекция в тип-сумму;
2. `inr(<выражение>)` — правая инъекция в тип-сумму;
3. `match <выражение> { inl(x) => <выражение> | inr(y) => <выражение> }` — разбор выражения типа-суммы по двум альтернативам (левая и правая инъекция);

Семантика и правила типизации для типов-сумм следуют традиционному определению [1, §11.9].

1.3.8 Списки (`#lists`)

Списки представлены в Stella типами однородных списков `[<тип>]` и следующими выражениями:

1. `[<выражение>, ..., <выражение>]` — список выражений;
2. `cons(<выражение>, <выражение>)` — конструктор списка из головы и хвоста;
3. `List::head(<выражение>)` — голова списка;
4. `List::tail(<выражение>)` — хвост списка;
5. `List::isempty(<выражение>)` — проверка списка на наличие элементов;

Если действует расширение `#structural-patterns`, то также имеются образцы

1. `[<образец>, ..., <образец>]` — образец списка фиксированной длины;
2. `cons(<образец>, <образец>)` — образец непустого списка;

Семантика и правила типизации для списков следуют традиционному определению [1, §11.12].

1.3.9 Варианты (`#variants`)

Варианты представлены в Stella типами `< <тег1> : <тип1>, ..., <тегN> : <типN> >` и следующими выражениями:

1. `< <тег> = <выражение> >` — инъекция в вариант;
2. разбор выражения по альтернативам типа-варианта:

```
match <выражение> {
    < <тег1> = <переменнаяN> > => <выражение1>
  | < <тег2> = <переменнаяN> > => <выражение2>
  | ...
  | < <тегN> = <переменнаяN> > => <выражениеN>
}
```

Семантика и правила типизации для вариантов следуют традиционному определению [1, §11.10].

1.3.10 Комбинатор неподвижной точки (`#fixpoint-combinator`)

Комбинатор неподвижной точки представлен в Stella выражением `fix(<выражение>)`.

Семантика и правила типизации для вариантов следуют традиционному определению [1, §11.11].

Список литературы

[1] Б. Пирс. *Типы в языках программирования: пер. с англ.* Лямбда пресс, 2012. ISBN: 9785791300829. URL: https://books.google.ru/books?id=HJJCkgEACAAJ.