Содержание

Проект. Этап 2

Содержание

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], поддержав ссылки, исключения и подтипы. А именно, ваша реализация должна поддерживать следующее:

- все обязательные части Этапа 1,
- последовательное исполнение (`#sequencing`),
- ссылки (`#references`),
- ошибки (`#panic`),
- исключения со значениями (`#exceptions` и `#exception-type-declaration`),
- структурные подтипы (`#structural-subtyping`),
- приведение типа (`#type-cast`),
- верхний и нижний типы (`#top-type`, `#bottom-type`),
- устранение неоднозначных типов за счёт `Bot` (`#ambiguous-type-as-bottom`).

[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;

extend with
  #structural-subtyping,
  #references,
  #top-type,
  #unit-type,
  #type-ascriptions;

fn main(x : &Nat) -> Unit {
  return x := (0 as Top)
}
```

```
// сообщение об ошибке
ERROR_UNEXPECTED_SUBTYPE:
ожидается подтип типа
  Nat
но получен тип
  Top
для выражения
  0 as Top
```

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. все обязательные конструкции Этапа 1;
2. для расширения `#sequencing`: `Sequence`
3. для расширений `#references`: `TypeRef`, `Ref`, `Deref`, `Assign`, `ConstMemory`
4. для расширения `#panic`: `Panic`
5. для расширения `#exceptions` и `#exception-type-annotation`: `DeclExceptionType`, `Throw`, `TryWith`, `TryCatch`
6. для расширения `#structural-subtyping`: нет новых конструкций (но требуется проверка наличия расширения);
7. для расширения `#ambiguous-type-as-bottom`: нет новых конструкций (но требуется проверка наличия расширения);

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. коды ошибки Этапа 1;
2. `ERROR_EXCEPTION_TYPE_NOT_DECLARED` — в программе используются исключения, но не объявлен их тип;
3. `ERROR_AMBIGUOUS_THROW_TYPE` — неоднозначный тип `throw`-выражения (`Throw`);
4. `ERROR_AMBIGUOUS_REFERENCE_TYPE` — неоднозначный тип адреса памяти (`ConstMemory`);
5. `ERROR_AMBIGUOUS_PANIC_TYPE` — неоднозначный тип ошибки (`Panic`);
6. `ERROR_NOT_A_REFERENCE` — попытка разыменовать (`Deref`) или присвоить значение (`Assign`) выражению не ссылочного типа;
7. `ERROR_UNEXPECTED_MEMORY_ADDRESS` — адрес памяти (`ConstMemory`) используется там, где ожидается тип, отличный от типа-ссылки (`TypeRef`);
8. `ERROR_UNEXPECTED_SUBTYPE` — тип выражения не является подтипом ожидаемого; эта ошибка должна возникать только если ни одна из более точных ошибок (выше) не возникла раньше.

1.2.4. Необязательные расширения

Следующие расширения могут быть реализованы за дополнительные баллы:

1. `#open-variant-exceptions`: `DeclExceptionVariant`
2. `#try-cast-as`: `TryCastAs`
3. `#type-cast-patterns`: `PatternCastAs`

1.3. Описание возможностей языка Stella

Stella — это язык программирования, созданный специально для практики реализации алгоритмов 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.

Ниже описаны возможности языка, отличные от тех, что описаны в Этапе 1.

1.3.1. Последовательное исполнение (`#sequencing`)

Последовательное исполнение представлено в Stella выражениями

```
<выражение> ; <выражение>
```

Семантика и правила типизации для последовательного исполнения следуют традиционному определению [1, §11.3].

1.3.2. Ссылки (`#references`)

Ссылки представлены в Stella типами `&<тип>` и выражениями

```
new(<выражение>)             // создание ссылки
*<выражение>                 // разыменование
<выражение> := <выражение>   // присваивание
<адрес>                      // ссылка как явный адрес в памяти
```

Семантика и правила типизации для ссылок следуют традиционному определению [1, §13].
Пример хорошо типизированной программы со ссылками:

```
language core;
extend with #unit-type, #references, #let-bindings, #sequencing;

fn inc_ref(ref : &Nat) -> Unit {
  return
    ref := succ(*ref)
}

fn inc3(ref : &Nat) -> Nat {
  return
    inc_ref(ref);
    inc_ref(ref);
    inc_ref(ref);
    *ref
}

fn main(n : Nat) -> Nat {
  return let ref = new(n) in inc3(ref)
}
```

1.3.3. Ошибки (`#panic`)

Ошибки представлены в Stella выражением

```
panic!           // (невосстановимая) ошибка
```

Семантика и правила типизации для ссылок следуют традиционному определению [1, §14.1].
Ошибки `panic!` не могут быть пойманы конструкциями `try-with` или `try-catch`.
Пример хорошо типизированной программы:

```
language core;
extend with #panic, #pairs, #fixpoint-combinator;

// декремент
fn dec(n : Nat) -> Nat {
  return Nat::rec(n, {0, 0},
    fn(k : Nat) {
      return fn(p : {Nat, Nat}) {
        return { succ(p.1), p.1 }
      }
    }).2
}

// вычитание
fn sub(n : Nat) -> fn(Nat) -> Nat {
```

```
  return fn(m : Nat) {
    return Nat::rec(m, n, fn(k : Nat) { return dec })
  }
}

// деление (с явным параметром для рекурсивного вызова)
fn mkdiv(div : fn(Nat) -> fn(Nat) -> Nat) -> fn(Nat) -> fn(Nat) -> Nat {
  return fn(n : Nat) {
    return fn(m : Nat) {
      return if Nat::iszero(n) then 0 else
        succ(div(sub(n)(m))(m))
    }
  }
}

// деление
fn div(n : Nat) -> fn(Nat) -> Nat {
  return fn(m : Nat) {
    return
      if Nat::iszero(m)
        then panic!^^I// ОШИБКА: деление на НОЛЬ!
        else fix(mkdiv)(n)(m)
  }
}

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

1.3.4. Исключения (`#exceptions`)

Исключения со значениями представлены в Stella типами выражениями

```
// выброс исключения
throw <выражение>

// попытка исполнения выражения с восстановлением
try { <выражение> } with { <выражение> }

// попытка исполнения выражения с восстановлением
// <образец> связывает переменные со значением исключения,
// выброшенного при вычислении первого выражения
try { <выражение> } catch { <образец> => <выражение> }
```

При использовании расширения `#exception-type-declaration`, тип значений, разрешённых для использования в исключениях, определяется декларацией:

```
exception type = <тип>
```

При использовании расширения `#open-variant-exceptions`, тип значений, разрешённых для использования в исключениях, является типом-вариантом, метки для которого определяются декларациями:

```
exception variant <метка> : <тип>
```

Семантика и правила типизации для исключений следуют традиционному определению [1, §14].

Пример хорошо типизированной программы:

```
language core;
extend with #exceptions, #exception-type-declaration;

exception type = Nat

fn fail(n : Nat) -> Bool {
  return throw(succ(0))
```

```
}

fn main(n : Nat) -> Bool {
  return try { fail(n) } catch { a => true }
}
```

1.3.5. Структурные подтипы (`#structural-subtyping`)

Структурные подтипы в Stella расширяют проверку типов правилом включения [1, §15.1] и отношением подтипов, соответствующему традиционному определению [1, §15].

1.3.6. Приведение типов (`#type-cast`)

Приведение типов в Stella представлено выражением:

```
<выражение> cast as <тип>
```

Семантика и правила типизации для приведения типа следует традиционному определению [1, §15.5.1].

Пример хорошо типизированной программы:

```
language core;
extend with #type-cast, #pairs, #top-type, #structural-subtyping;

fn dup(x : Top) -> { Top, Top } {
  return { x, x }
}

fn main(n : Nat) -> Nat {
  return (dup(n) cast as {Nat, Nat}).1
}
```

1.3.7. Верхний и нижний типы (`#top-type`, `#bottom-type`)

Верхний и нижний типы в Stella представлены типами:

```
Top // верхний тип
Bot // нижний тип
```

Семантика и правила типизации для верхнего и нижнего типов следуют традиционному определению [1, §15.4].

1.3.8. Типизация неоднозначных типов (`#ambiguous-type-as-bottom`))

Для выражений, тип которых неоднозначен, при наличии расширения `#ambiguous-type-as-bottom`) используется нижний тип (`Bot`), вместо ошибки типизации.

Пример хорошо типизированной программы:

```
language core;
extend with #ambiguous-type-as-bottom, #structural-subtyping, #sum-types;

fn main(n : Nat) -> Bool + Nat {
  return (fn (x : Nat) {
    return inr(x) // в этом месте выводится тип-сумма Bot + Nat
  })(n)
}
```

1.3.9. Динамическая проверка типа (`#try-cast-as`, `#type-cast-patterns`)

Динамическая проверка типа представлена в Stella выражением

```
try { <выражение> } cast as <тип>
  { <образец> => <выражение> }  // ветка успешного приведения типа
with
  { <выражение> }               // ветка неуспешного приведения
```

Семантика и правила типизации соответствуют традиционному определению [1, §15.5.1].

Динамическая проверка типа также возможна при сопоставлении с образцом:

```
<образец> cast as <тип>
```

Такой образец будет успешно сопоставлен, если приведение типа успешно. Образец при этом должен соответствовать указанному типу.

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.