Содержание

Проект. Этап 3

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 типовые переменные в двух контекстах (независимо) — реконструкция типов и параметрический полиморфизм. А именно, ваша реализация должна поддерживать следующее:

- все обязательные части Этапа 1,
- реконструкцию типов (`#type-reconstruction`) — без вариантов и кортежей, но с парами и типами-суммами,
- универсальные типы (`#universal-types`).

ВАЖНО: допускается (но нежелательна) реализация реконструкции типов и универсальных типов в двух независимых исполнениях. Однако, в случае реализации в одной программе, наличие или отсутствие расширения `#type-reconstruction` должно учитываться и приводить к различным результатам для некоторых программ.

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] https://fizruk.github.io/stella/
[2] https://bnfc.digitalgrammars.com
[3] https://www.antlr.org

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 #type-reconstruction, #tuples;

fn main(f : Nat) -> { Nat, auto } {
  return { f, f(0) }
}
```

```
// сообщение об ошибке
ERROR_UNEXPECTED_TYPE_FOR_EXPRESSION:
при попытке унифицировать ожидаемый тип
  fn (?T2) -> ?T3
с полученным типом
  Nat
при проверке выражения
  f
```

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.

[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 `#type-reconstruction`: `TypeAuto`, `TypeVar`
3. для расширения `#universal-types`: `TypeVar`, `DeclFunGeneric`, `TypeForAll`, `TypeAbstraction`, `TypeApplication`

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_OCCURS_CHECK_INFINITE_TYPE` — во время унификации возникает ограничение, порождающее (запрещённый) бесконечный тип;
3. `ERROR_NOT_A_GENERIC_FUNCTION` — при попытке применить (`TypeApplication`) универсальное выражение к типовому аргументу, выражение оказывается не универсальной функцией; ошибка должна возникать до проверки аргумента;
4. `ERROR_INCORRECT_NUMBER_OF_TYPE_ARGUMENTS` — вызов универсальной функции (`TypeApplication`) происходит с некорректным количеством типов-аргументов;
5. `ERROR_UNDEFINED_TYPE_VARIABLE` — в типе содержится необъявленная типовая переменная (только для `#universal-types`).

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. Реконструкция типов (`#type-reconstruction`)

Реконструкция типов в Stella использует типизацию на основе ограничений и расширяет синтаксис специальным типом `auto`. Проверка с реконструкцией типов осуществляется следующим образом:

1. Перед началом проверки, каждое вхождение типа `auto` в исходной программе заменяется на свежую типовую переменную.
2. Затем осуществляется процедура генерации ограничений, следуя правилам типизации на основе ограничений [1, §22.3]. В дополнение к ограничениям для унификации, Stella также собирает множество специальных ограничений для проверки полного покрытия сопоставления с образцом, которое происходит после реконструкции типов. *В вашей реализации не требуется полная поддержка проверки покрытия образцами с реконструкцией типов и достаточно собирать только множество ограничений для унификации типов.*
3. Наконец, осуществляется унификация ограничений, собранный для всей программы, в результате которой либо выдаётся сообщение об ошибке, либо проверка типов успешно завершается. Неоднозначные типы (типовые переменные, без известного конкретного типа) не считаются некорректными в этом режиме.

```
// исходная программа на Stella
language core;

extend with #type-reconstruction;

// addition of natural numbers
fn Nat::add(n : auto) -> fn(auto) -> auto {
  return fn(m : auto) {
    return Nat::rec(n, m, fn(i : auto) {
      return fn(r : auto) {
        return if r then r else r; // r := r + 1
      };
    });
  };
}

// square, computed as a sum of odd numbers
fn square(n : auto) -> auto {
  return Nat::rec(n, 0, fn(i : auto) {
      return fn(r : auto) {
        // r := r + (2*i + 1)
        return Nat::add(i)( Nat::add(i)( succ( r )));
      };
  });
}

fn main(n : auto) -> auto {
  return square(n);
}
```

```
// программа, где все 12 типов-auto заменены на свежие типовые переменные
language core;

extend with #type-reconstruction;

// addition of natural numbers
fn Nat::add(n : ?T1) -> fn(?T2) -> ?T3 {
  return fn(m : ?T4) {
    return Nat::rec(n, m, fn(i : ?T5) {
      return fn(r : ?T6) {
        return if r then r else r; // r := r + 1
      };
    });
  };
}

// square, computed as a sum of odd numbers
fn square(n : ?T7) -> ?T8 {
  return Nat::rec(n, 0, fn(i : ?T9) {
      return fn(r : ?T10) {
        // r := r + (2*i + 1)
        return Nat::add(i)( Nat::add(i)( succ( r )));
      };
  });
}

fn main(n : ?T11) -> ?T12 {
  return square(n);
}
```

1.3.2. Универсальные типы (**#universal-types**)

Универсальные типы в Stella поддержаны синтаксическими конструкциями для объявления именованных и анонимных универсальных функций, а также применения (специализации) универсальных функций.

Универсальные функции объявляются при помощи ключевого слова `generic` перед `fn` и указания списка формальных типов-параметров в квадратных скобках:

```
language core;

extend with #universal-types;

generic fn identity[T](x : T) -> T {
  return x
}

fn main(x : Nat) -> Nat {
  return identity[Nat](x)
}
```

Универсальные типы записываются как `forall X. T` и могут быть как аргументами, так и возвращаемыми значениями других функций:

```
language core;

extend with #universal-types;

generic fn const[X](x : X) -> forall Y. fn(Y) -> X {
  return generic [Y] fn(y : Y) { return x }
}

fn main(x : Nat) -> Nat {
  return const[Nat](x)[Bool](false)
}
```

Универсальные функции могут принимать несколько типов-аргументов:

```
language core;

extend with #universal-types;

generic fn const[X, Y](x : X) -> fn(Y) -> X {
  return fn(y : Y) { return x }
}

fn main(x : Nat) -> Nat {
  return const[Nat, Bool](x)(false)
}
```

Как и в Системе F, универсальные типы в Stella импредикативны и позволяют, в частности, типизировать самоприменимые термы:

```
generic fn self_app[X](f : forall X . fn(X) -> X) -> forall X . fn(X) -> X {
  return f[forall X . fn(X) -> X](f)
}
```

Семантика и правила типизации для универсальных типов следуют традиционному определению Системы F [1, §23].

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.