Мини-проект на miniKanren

В этом проекте, реализуйте реляционный (логический) тайпчекер[1]. Упражнения в этом проекте указаны для `faster-minikanren`, но вы можете выбрать другую реализацию miniKanren или даже реализовать и использовать свою версию miniKanren, на любом языке программирования по вашему выбору[2].

В системах типов, утверждение о типизации — это формальное утверждение в виде

$$\Gamma \vdash e : T,$$

которое читается как "в контексте типизации Γ выражение e имеет тип T". Контекст — это просто словарь типов для локальных переменных (переменных в текущей области видимости). Например:

$$x : \mathsf{Int}, f : \mathsf{Int} \to \mathsf{Bool} \vdash f\ (x + x) : \mathsf{Bool}$$

В этом примере x имеет тип Int, а f — функция из Int в Bool. Контекст указывает типы локальных переменных x и f, которые используются в выражении $f\ (x + x)$. Утверждение типизации говорит, что в этом контексте данное выражение имеет тип Bool.

В следующих упражнениях вы постепенно реализуете отношение типизации:

```
(typed-expr° context expr type)
```

Выразительная мощь логического программирования заключается в способности запускать любые отношения в разных "режимах". При хороших условиях, реализовав проверку типов в отношении выше, мы можем надеяться получить автоматически (или с небольшими доработками) другие режимы:

- Если нам известны контекст, выражение и тип — мы производим **проверку типов**. Обычно это достаточно прямолинейная задача, и мы будем фокусироваться на этом режиме при реализации.
- Если контекст и выражение известны, но тип неизвестен — мы производим **вывод типов**. Поскольку многие алгоритмы вывода типов полагаются на унификацию, которая "зашита" в логическое программирование, мы надеемся получить вывод типов практически бесплатно.
- Если контекст и тип известны, но не выражение — мы производим **синтез программ**. Хотя практические алгоритмы синтеза программ требуют эффективного прунинга пространства поиска (даже при синтеза на основе типов), мы надеемся, что мы сможем генерировать хотя бы простые программы.

Несмотря на возможные подводные камни, наивная идея реализовать проверку типов и автоматически получить инструменты для вывода типов и синтеза программ кажется очень привлекательной.

[1] Тайпчекер — это программа, проверяющая корректность программы с точки зрения системы типов.
[2] См. http://minikanren.org/#implementations.

3.1. Контексты

Для начала реализуйте вспомогательные отношения для контекстов. Мы представляем контексты списками пар из идентификатора переменной (например, `x`) и её типа (например, `Int`). Мы также будем разрешать в контексте числовые константы (например, `123`) для удобства.

В следующих упражнениях мы используем только *простые* типы:

- `'Bool` и `'Int` для булевых и целочисленных выражений.
- Если `t1` и `t2` — типы, то `(t1 -> t2)` — тип функций, принимающих значение `t1` на вход и возвращающих значение типа `t2`.
  - корректные типы функций:
    + `'(Int -> Bool)`
    + `'(Int -> (Int -> Int))`
    + `'((a -> b) -> ([a] -> [b]))`
  - некорректные типы функций (не хватает скобок):
    - `'(Int -> Int -> Int)`
    - `'((a -> b) -> [a] -> [b])`
- Другие значения (e.g. `'a`, `'b`, `'[a]`) считаются неинтерпретированными типами. То есть мы разрешаем такие значения в типах, но ничего про них не знаем.

Упражнение 3.1. Реализуйте отношение `context-lookupº` такое, что `(context-lookupº var type context)` выполняется, если `context` содержит информацию о том, что `var` имеет тип `type`.

```
(define sample-context
  '((x . Int)
    (f . (Int -> Bool))
    (y . Int)
    (b . Bool)
    (+ . (Int -> (Int -> Int)))))

(run* (type) (context-lookupº sample-context 'x type))
; '(Int)
(run* (type) (context-lookupº sample-context 'z type))
; '()
(run* (var) (context-lookupº sample-context var 'Int))
; '(x y)
(run* (var type) (context-lookupº sample-context var type))
; '((x Int) (f (Int -> Bool)) (y Int) (b Bool) (+ (Int -> (Int -> Int))))
```

Упражнение 3.2. Реализуйте отношение `context-varsº` такое, что `(context-varsº context vars)` выполняется, если `context` содержит в точности список переменных `vars`.

```
(run* (vars) (context-varsº sample-context vars))
; '((x f y b +))
(run* (context) (context-varsº context '(x y z)))
; '(((x . _.0) (y . _.1) (z . _.2)))
```

3.2. Бестиповые выражения

Прежде, чем перейти к утверждениям типизации, сначала реализуем упрощённую версию для выражений без типов. В отсутствии типов, мы утверждаем только о том, какие переменные (константы) используются в выражении:

$$x, y, f \vdash f\ (x + x)$$

Выражение $f\ (x+x)$ корректно в контексте, содержащем переменные x, y и f. Это так, потому что выражение использует только переменные x и f, которые явно указаны в контексте. Переменная y не используется, но это не считается проблемой.

Упражнение 3.3. Реализуйте отношение `untyped-expr°` такое, что `(untyped-expr° vars expr)` выполняется, если `expr` — корректное выражение в бестиповом контексте `vars`:

- Если переменная (контанта) используется в `expr`, она должна явно выть указана в `vars`. Но если переменная (константа) указана в `vars`, она не обязана использоваться в `expr`.
- Выражение может быть одним из следующих:
  - переменная: x, y, z, ...
  - литерал целого числа: $0, 1, 2, ...$
  - бинарная операция: $(e_1 + e_2)$ or $(e_1 \times e_2)$
  - применение функции к одному аргументу: $(f\ x)$
  - условное выражение: if e_1 then e_2 else e_3

1. Убедитесь, что `untyped-expr°` способен порождать выражения, которые используют переменные (и константы) из заданного списка:

```
(run 6 (expr) (untyped-expr° '(x) expr))
; '(x (x + x) (x x) (x * x) (if x then x else x) (x + (x + x)))
(run 8 (expr) (untyped-expr° '(x y) expr))
; '(x y (x + x) (x * x) (x + y) (x x) (y x) (if x then x else x))
```

2. Убедитесь, что `untyped-expr°` определять контекст, необходимый для данного выражения:

```
(run 1 (vars) (untyped-expr° vars
  '(if (p x) then (x * (f (x + 1))) else (f (f x)))))
; '((p x f 1 . _.0))
```

Подсказка: используйте ограничение `symbolo` *и* `numbero` *для проверки выражения на то, является ли оно переменной или литералом.*

Подсказка: возможно, вам будет полезно использовать `member°` *и/или* `not-member°`.

Нам может быть полезно ограничить размер выражений, используя реляционную арифметику из `minikanren/numbers`:

Упражнение 3.4. Реализуйте отношение `bounded-untyped-expr°` такое, что `(bounded-untyped-expr° max-depth vars expr)` выполняется, если `expr` — корректное выражение в бестиповом контексте `vars` и глубиной (высотой синтаксического дерева) не более `max-depth`.

```
(run* (expr) (bounded-untyped-expr° (build-num 2) '(x) expr))
; '(x (x x) (x + x) (x * x) (if x then x else x))
(run 1 (vars) (bounded-untyped-expr° (build-num 5) vars
  '(if (p x) then (x * (f (x + 1))) else (f (f x)))))
; '((p x f 1 . _.0))
```

3.3. Проверка типов, вывод типов и синтез программ с простыми типами

Упражнение 3.5. Реализуйте отношение `typed-expr°` такое, что `(typed-expr° context expr type)` выполняется, если `expr` — корректное выражение типа `type` в контексте `context`:

- переменная x имеет тип T тогда и только тогда, когда контекст Γ явно упоминает $x : T$
- выражения $(e_1 + e_2)$ и $(e_1 \times e_2)$ имеют тип Int и требуют, чтобы выражения e_1 и e_2 имели тип Int
- применение функции к аргументу $(f\ x)$ имеет тип T тогда и только тогда, когда f тип функций $X \to T$ и x имеет тип X (для какого-то X)
- выражение if e_1 then e_2 else e_3 имеет тип T, если e_1 имеет тип Bool, а e_2 и e_3 имеют тип T

1. Убедитесь, что `typed-expr°` работает в режиме **проверки типов**: оно может проверить, что данное выражение хорошо типизировано в данном контексте с данным типом:

```
(run* (q) (typed-expr° sample-context '(if (f x) then (x + y) else y) 'Int))
; '(_.0)
(run* (q) (typed-expr° sample-context '(if (f x) then (x + b) else y) 'Int))
; '()
```

2. Убедитесь, что `typed-expr°` работает в режиме **вывода типов**: оно может проверить, что данное выражение хорошо типизированно в данном контексте, и вывести тип:

```
(run* (type) (typed-expr°
  `((1 . Int) . ,sample-context)    ; we add constants explicitly to the context
  '(if (f x) then (x + 1) else y)   ; expression is given
  type))                            ; type is inferred
; '(Int)
```

3. Убедитесь, что `typed-expr°` работает в режиме **вывода контекста**: оно может вывести переменные (и константы), использующиеся в выражении, а также их типы, при которых выражение хорошо типизировано:

```
(run 1 (context) (typed-expr° context '(if (f x) then (x + b) else y) 'Int))
; '(((f Int -> Bool) (x . Int) (b . Int) (y . Int) . _.0))
(run 1 (context) (typed-expr° context '((f (g x)) + (g (f x))) 'Int))
; '(((f Int -> Int) (g Int -> Int) (x . Int) . _.0))
(run 1 (context type) (typed-expr° context '(if (p x) then (f x) else x) type))
; '((((p _.0 -> Bool) (x . _.0) (f _.0 -> _.0) . _.1) _.0))
```

4. Убедитесь, что `typed-expr°` работает в режиме **синтеза программ**: оно может синтезировать программы в данном контексте с данным типом:

```
(run 6 (expr) (typed-expr° sample-context expr 'Int))
; '(x y (x + x) (x * x) (x + y) (if b then x else x))
(run 6 (expr) (typed-expr° sample-context expr 'Bool))
; '(b (f x) (f y) (f (x + x)) (if b then b else b) (f (x * x)))
(run 6 (expr) (typed-expr° sample-context expr '(Int -> Int)))
; '((+ x) (+ y) (+ (x + x)) (+ (x * x)) (+ (x + y)) (+ (if b then x else x)))
```