# Статический анализатор типов «Penelope»

Итоговая работа по прикладной математике

Centrix14

4 января 2024 г.

## Содержание



## 1 Вычисление типов значений

### 1.1 Простое получение типов

```
(let ((x 14))
  (list (type-of 1) (type-of 14) (type-of x)))
```

Получается, что числа 0 и 1 считаются не числами, а битами. При том

```
(typep 1 'integer)
```

Занятно!

### 1.2 Безопасное вычисление типов

Чтобы вычислять типы безопасно и абстрагироваться от всяких там `BIT` и `SIMPLE-ARRAY` напишем свою функцию.

```
(defun ty (x)
  (typecase x
    (integer 'integer)
    (string 'string)
    (boolean 'boolean)
    (t (type-of x))))
```

`ty` — это распространённое в профессиональной среде сокращение для слова *type*. Так что тут всё с именованием нормально.

## 2 Вычисление типов выражений

### 2.1 Простой обход дерева с определением типов

Простая функция, которая обходит дерево и возвращает типы его частей.

```
(defun get-typed-tree (expr)
  (if (listp expr)
      (map 'list #'get-typed-tree expr)
      (ty expr)))
```

Применим функцию.

```
(map 'list #'get-typed-tree
    '((1 t *)
      (* 2 2)
      (if (> x 2) x 2)
      (lambda (x) (* x 2))
      (max x 2)))
```

### 2.2 Работа в контексте

Определение типов происходит в некотором контексте. Будем описывать его хэш-таблицей.

```
(defvar *env* (make-hash-table))
```

Заполним контекст записями, где ключом будет название привязки, а значением — тип.

```
(map 'list
     (lambda (pair)
       (let ((variable (first pair))
             (type (second pair)))
         (setf (gethash variable *env*) type)
         pair))
     '((age integer)
       (name string)
       (+ math-op)
       (- math-op)
       (* math-op)
       (/ math-op)
       (> math-op)
       (< math-op)
       (= eq)
       (if if)
       (lambda lambda)))
```

Как можно видеть, мы делаем необычную вещь — присваиваем типы ещё и различным операторам (функциям) языка.

### 2.3 Обход дерева с учётом контекста

Теперь перепишем `get-typed-tree` с учётом контекста.

```
(defun get-typed-tree (expr env)
  (if (listp expr)
      (map 'list (lambda (x)
                   (get-typed-tree x env))
           expr)
      (let ((type (gethash expr env)))
        (if type
            type
            (ty expr)))))
```

Проверим, как работает наша функция

```
(get-typed-tree '(> age 2) *env*)
```

### 2.4 Соотнесение типов

Мы можем теперь определять типы выражений. Напишем же функцию, которая будет соотносить член выражения и его тип.

```
(defun assoc-type (expr env)
  (map 'list #'list expr (get-typed-tree expr env)))
```

Посмотрим, что получилось.

```
(assoc-type '(if (> age 18) "Welcome!" "You too young...") *env*)
```

# 3 Полноценная проверка типов

## 3.1 Правила типизации

Правила типизации так же будем хранить в хэш-таблице.

```
(defvar *rules* (make-hash-table :test #'equalp))
```

Ключом будет предпосылка правила, а значением — вывод. Заполним таблицу.

```
(map 'list
     (lambda (rule)
       (let ((thesis (first rule))
             (conclusion (second rule)))
         (setf (gethash thesis *rules*) conclusion)))
     '(((math-op integer integer) integer)
       ((eq integer integer) boolean)
       ((if boolean integer integer) integer)
       ((if boolean boolean boolean) boolean)))
```

## 3.2 Получение типов по правилам

Теперь напишем функцию, которая будет получать тип выражения на основе правил.

```
(defun get-type-from-rule (expr rules)
  (let ((type (gethash expr rules)))
    (if type
        type
        (error "Incorrect types in ~a~%" expr))))
```

Проверим, как работают наши правила.

```
(get-type-from-rule '(math-op integer integer) *rules*)
```

Всё работает как должно. Если ввести неверное правило, то мы получим ошибку.

Так мы получаем 2 части проверки типов: функцию расстановки типов в произвольном дереве и функцию вывода типа по правилам. Посмотрим, как они работают в тандеме.

```
(get-type-from-rule (get-typed-tree '(+ age 2) *env*) *rules*)
```

Работает как и задумано. Теперь остаётся объединить наши функции, чтобы получить полноценную проверку типов.

### 3.3 Проверка типов

По структуре, проверка типов будет такой же, как и `get-typed-tree`, мы лишь добавим вывод типов по правилам.

```
(defun type-check (rules env expr)
  (if (listp expr)
      (get-type-from-rule
       (map 'list (lambda (x)
                    (type-check rules env x))
            expr)
       rules)
      (let ((type (gethash expr env)))
        (if type
            type
            (ty expr)))))
```

Посмотрим, как наша нехитрая конструкция работает.

```
(type-check *rules* *env* '(if (= age 18) t t))
```

Тип выведен верно. Никаких ошибок нет. Есть только одна проблема: правило типизации для `if` мы записали следующим образом: `((if boolean int int) int)`. Далее мы добавляем: `((if boolean boolean boolean) boolean)`. Почему? Всё потому, что вообще правило для `if` необходимо дать в общем виде:

$$\frac{\Gamma \vdash v1 : \mathrm{t} \quad \Gamma \vdash v2 : \mathrm{t} \quad \Gamma \vdash c : \mathrm{bool}}{\Gamma \vdash \mathrm{v1\ if\ c\ else\ v2} : \mathrm{t}}.$$

Но наша система сделать этого не позволяет. Оно, на самом деле, и хорошо. Так мы можем точно и конкретно задать, какие типы могут участвовать в записи условных конструкций.