Символьные вычисления

Денис С. Мигинский

Задача

Реализовать аналитическое вычисление производной выражений, содержащих основные алгебраические операции. Решение должно быть гибким, расширяемым, и т.д., и т.п.

Задача и решение описано в книге
H. Abelson, G.J. Sussman, J. Sussman
“Structure and Interpretation of Computer Programs”

Анализ задачи

Каково представление и основные элементы выражений?

Как абстрагироваться от их системного представления?

Как вычислять производную (алгоритм и т.д.)?

Не ли скрытых проблем, все ли мы учли?

Представление выражений

Основные элементы выражений:

- Числовая константа
- Переменная (в математическом смысле)
- Сумма
- Произведение
- Деление (или возведение в степень -1)
- Экспонента
- …

Для каждого элемента необходимо определить следующие элементы API:

- Конструктор(-ы)
- Проверку типа
- Взятие параметров

API для констант и переменных

```
;порождение константы
(defn constant [num] …)
;проверка типа для константы
(defn constant? [expr] …)
;получение значения константы
(defn constant-value [expr] …)

;порождение переменной
(defn variable [name] …)
;проверка типа для переменной
(defn variable? [expr] …)
;получение значения для переменной
(defn variable-name [expr] …)
;сравнение переменных
(defn same-variables? [v1 v2] …)
```

Тесты для констант и переменных

```
(test/is (variable? (variable :x)))
(test/is (= :x (variable-name (variable :x))))
(test/is (same-variables?
           (variable :x)
           (variable :x)))
(test/is (not (same-variables?
                (variable :x)
                (variable :y))))

(test/is (constant? (constant 1)))
(test/is (= 1 (constant-value (constant 1))))
```

API для алгебраических операций

```
;порождение суммы
(defn sum [expr & rest] …)
;проверка типа для суммы
(defn sum? [expr] …)
;порождение произведения
(defn product [expr & rest] …)
;проверка типа для произведения
(defn product? [expr] …)
;список аргументов выражения
(defn args [expr] …)
;порождение обратного выражения
(defn invert [expr] …)
;проверка типа для обратного выражения
(defn invert? [expr] …)
```

Реализация переменных

```
(defn variable [name]
  {:pre [(keyword? name)]}
  (list ::var name))

(defn variable? [expr]
  (= (first expr) ::var))

(defn variable-name [v]
  (second v))

(defn same-variables? [v1 v2]
  (and
    (variable? v1)
    (variable? v2)
    (= (variable-name v1)
      (variable-name v2))))
```

Реализация некоторых других функций базового API

```
(defn sum [expr & rest]
  (cons ::sum (cons expr rest)))

(defn sum? [expr]
  (= ::sum (first expr)))

(defn args [expr]
  (rest expr))

(defn invert [expr]
  (list ::inv expr))
```

Функция дифференцирования

```
;функция дифференцирования
(defn diff [expr vr] …)
```

Как должна быть устроена функция дифференцирования?

Как сделать функцию расширяемой?

Правила вывода

Существуют правила вывода для:

- константы
- переменной
- суммы
- произведения
- 1/f(x)
- exp(x)

Большинство правил вывода рекурсивны

Реализация списка правил через шаблон Chain of responsibilities

```
(declare diff)
;список правил вывода
(def diff-rules
  (list
    [pred1 transform1]
    [pred2 transform2]
     …))
```

Реализация функции дифференцирования

```
(defn diff [expr vr]
  ((some (fn [rule]
           (if ((first rule) expr vr)
             (second rule)
             false))
         diff-rules)
    expr vr))
```

Правила вывода для констант и переменных

```
;константа
[(fn [expr vr] (constant? expr))
 (fn [expr vr] (constant 0))]
;переменная дифференцирования
[(fn [expr vr]
    (and
      (variable? expr)
      (same-variables? expr vr)))
 (fn [expr vr] (constant 1))]
;другая переменная
[(fn [expr vr] (variable? expr))
 (fn [expr vr] (constant 0))]
```

Правила вывода для суммы

```
;сумма
[(fn [expr vr] (sum? expr))
 (fn [expr vr]
   (apply sum
     (map
       #(diff % vr)
       (args expr))))]
```

Модифицированное правило формирования произведения

```
(defn product [expr & rest]
  (if (empty? rest)
    expr
    (cons ::product (cons expr rest))))
```

Правила вывода для произведения

```
[(fn [expr vr] (product? expr))
 (fn [expr vr]
   (let [first-arg (first (args expr))
         rest-prod
           (apply product
             (rest (args expr)))]
     (sum
       (product
         (diff first-arg vr) rest-prod)
       (product first-arg
         (diff rest-prod vr)))))]
```

Проблема: сумма

```
(diff (sum (constant 2) (variable :x))
  (variable :x))
>>
(::sum
  (::const 0)
  (::const 1))
```

Проблема: произведение

```
(diff (product
        (constant 2)
        (variable :x))
  (variable :x))
>>
(::sum
  (::product
    (::const 0)
    (::var :x))
  (::product
    (::const 2)
    (::const 1)))
```

Еще раз модифицированное правило формирования произведения

```
(defn product [expr & rest]
  (let [normalized-exprs
          (collapse-prod-constants
            (cons expr rest))]
    (if (= 1 (count normalized-exprs))
      (first normalized-exprs)
      (cons ::product
        normalized-exprs))))
```

Свертка констант

```
(defn- collapse-prod-constants [exprs]
  (let [consts (filter constant? exprs)
        other-exprs
          (remove constant? exprs)
        combined-const
          (reduce
            (fn [acc entry]
              (* acc
                (constant-value entry)))
            1 consts)]
```

…

Свертка констант: продолжение

```
(defn- collapse-prod-constants [exprs]
  (let …
    (cond
      (= combined-const 0)
        (list (constant 0))
      (= combined-const 1)
        (if (empty? other-exprs)
          (list (constant 1))
          other-exprs)
      :default
        (cons (constant combined-const)
          other-exprs))))
```

Задача C4

По аналогии с задачей дифференцирования реализовать представление символьных булевых выражений с операциями конъюнкции, дизъюнкции отрицания, импликации. Выражения могут включать как булевы константы, так и переменные. Реализовать подстановку значения переменной в выражение с его приведением к ДНФ.

Код должен быть покрыт тестами, API документирован.