Преобразование функций высшего порядка в реляционную форму

Лозов Петр

СПбГУ
JetBrains Research

Языки программирования и компиляторы
3 апреля, 2017
Ростов-на-Дону, Россия

Реляционное программирование

- Представление программ как отношений
- Нет различий между аргументами и результатом
  - Гибкость при использовании

Язык miniKanren

- *The Reasoned Schemer* (Daniel P. Friedman, William Byrd, Oleg Kiselyov, 2005)
- Минималистичный DSL для Scheme/Racket
- Семейство языков (μKanren, α-Kanren, cKanren и т.д.)
- Встроен в большое количество языков (например, в OCaml, Haskell, Scala и т.д.)

Язык miniKanren

- *The Reasoned Schemer* (Daniel P. Friedman, William Byrd, Oleg Kiselyov, 2005)
- Минималистичный DSL для Scheme/Racket
- Семейство языков (μKanren, α-Kanren, cKanren и т.д.)
- Встроен в большое количество языков (например, в OCaml, Haskell, Scala и т.д.)

- Реляционное программирование на miniKanren (William E. Byrd, 2009)
  - Неформально описан метод преобразования функций первого порядка

Функция и отношение

```
type number = Zero | S of number
```

Функция и отношение

```
type number = Zero | S of number
```

```
let rec add x y =
  match x with
  | Zero → y
  | S x’ → S (add x’ y)
```

Функция и отношение

```
type number = Zero | S of number
```

```
let rec add x y z
```

```
let rec add x y =
  match x with
  | Zero → y
  | S x' → S (add x' y)
```

Функция и отношение

```
type number = Zero | S of number
```

```
let rec add x y =
  match x with
  | Zero → y
  | S x' → S (add x' y)
```

```
let rec add x y z =
  ((x ≡ Zero) ∧ (y ≡ z))
```

Функция и отношение

```
type number = Zero | S of number
```

```
let rec add x y =
  match x with
  | Zero → y
  | S x' → S (add x' y)
```

```
let rec add x y z =
  ((x ≡ Zero) ∧ (y ≡ z)) ∨
  (fresh (x' z')
```

Функция и отношение

```
type number = Zero | S of number
```

```
let rec add x y =
  match x with
  | Zero → y
  | S x' → S (add x' y)
```

```
let rec add x y z =
  ((x ≡ Zero) ∧ (y ≡ z)) ∨
  (fresh (x' z') (
     (x ≡ S x')
```

Функция и отношение

```
type number = Zero | S of number
```

```
let rec add x y =
  match x with
  | Zero → y
  | S x' → S (add x' y)
```

```
let rec add x y z =
  ((x ≡ Zero) ∧ (y ≡ z)) ∨
  (fresh (x' z') (
     (x ≡ S x') ∧
     (add x' y z')
```

Функция и отношение

$$\textbf{type}\ \text{number} = \text{Zero} \mid \text{S}\ \textbf{of}\ \text{number}$$

```
let rec add x y =
  match x with
  | Zero → y
  | S x’ → S (add x’ y)
```

```
let rec add x y z =
  ((x ≡ Zero) ∧ (y ≡ z)) ∨
  (fresh (x’ z’) (
     (x ≡ S x’) ∧
     (add x’ y z’) ∧
     (z ≡ S z’))
```

Реляционная форма

- Большая гибкость в сравнении с функциями
- Написание отношений более утомительно
- Часто отношение можно получить из некоторой функции регулярным образом

Функциональный язык

- Лямбда-исчисление
- Конструкторы
- Сопоставление с образцом
- Конструкции let и let rec

Необходимость типизации

```
let unbox cf =
  match cf with
  | C f → f
```

Необходимость типизации

```
let unbox cf =
  match cf with
  | C f → f
```

```
let unbox cf res =
  fresh(f) (
    (cf ≡ C f) ∧
    (f ≡ res))
```

Необходимость типизации

- Сужение множества входных функций
- Преобразование конструкторов
- Преобразование сопоставления с образцом

Типизация

- Система Хиндли-Милнера
- Ограничение типов контсрукторов
- Ограничение полиморфизма

Преобразование типов

$$\begin{array}{rcl} [g] & = & g \to \mathfrak{G} \\ [\forall\alpha.\,t] & = & \forall\alpha.\,[t] \\ [t_1 \to t_2] & = & [t_1] \to [t_2] \end{array}$$

Пример преобразования типов

$$\begin{array}{rcl} [int] & = & int \to \mathfrak{G} \\ [string \to int] & = & (string \to \mathfrak{G}) \to (int \to \mathfrak{G}) \end{array}$$

Преобразование функции

```
map :: (α → β) → α list → β list

let rec map = λ f. λ l.
  match l with
  | Nil       → Nil
  | Cons x xs → Cons (f x) (map f xs)
```

Преобразование конструктора

$\texttt{Cnst}_{Nil}$	$\texttt{Cnst}_{Cons}$
`Nil`	`Cons (f x) (map f xs)`

λ `res. res` $\equiv$ `Nil`	λ `res.`
	  `fresh``(`$\texttt{arg}_1$ $\texttt{arg}_2$`) (`
	    `(f x `$\texttt{arg}_1$`)` $\wedge$
	    `(map f xs `$\texttt{arg}_2$`)` $\wedge$
	    `(res` $\equiv$ `Cons `$\texttt{arg}_1$ $\texttt{arg}_2$`))`

Преобразование сопоставления с образцом

```
match l with
| Nil       → Nil
| Cons x xs → Cons (f x) (map f xs)
```

```
λ res.
  fresh (res_l) (
    (l res_l) ∧ (
      (Case Nil) ∨
      (Case Cons)))
```

Преобразование сопоставления с образцом

$$
\begin{array}{l}
\mathbf{match}\ \mathtt{l}\ \mathbf{with} \\
\mid \mathtt{Nil} \qquad\quad \rightarrow \mathtt{Nil} \\
\mid \mathtt{Cons\ x\ xs} \rightarrow \mathtt{Cons\ (f\ x)\ (map\ f\ xs)}
\end{array}
$$

$$
\begin{array}{l}
\lambda\ \mathtt{res}. \\
\quad \mathbf{fresh}\ (\mathtt{res}_l)\ (\\
\quad\quad (\mathtt{l}\ \mathtt{res}_l) \wedge (\\
\quad\quad\quad ((\mathtt{res}_l \equiv \mathtt{Nil}) \wedge (\mathtt{Cnst}_{Nil}\ \mathtt{res})) \vee \\
\quad\quad\quad (\mathbf{fresh}\ (\mathtt{res}_x\ \mathtt{res}_{xs})\ (\\
\quad\quad\quad\quad (\mathtt{res}_l \equiv \mathtt{Cons}\ \mathtt{res}_x\ \mathtt{res}_{xs}) \wedge \\
\quad\quad\quad\quad ((\lambda \mathtt{x}.\ \lambda \mathtt{xs}.\ \mathtt{Cnst}_{Cons})\ ((\equiv)\ \mathtt{res}_x)\ ((\equiv)\ \mathtt{res}_{xs})\ \mathtt{res})))))
\end{array}
$$

Результат преобразования

$$\text{map} :: ((\alpha \to \mathfrak{G}) \to \beta \to \mathfrak{G}) \to (\alpha\ \text{list} \to \mathfrak{G}) \to \beta\ \text{list} \to \mathfrak{G}$$

```
let rec map = λf. λl. λres.
  fresh (res_l) (
    (l res_l) ∧ (
      ((res_l ≡ Nil) ∧ (res ≡ Nil)) ∨
      (fresh (res_x res_xs) (
        (res_l ≡ Cons res_x res_xs) ∧
          (fresh (arg_1 arg_2) (
            (f ((≡) res_x) arg_1) ∧
            (map f ((≡) res_xs) arg_2) ∧
            (res ≡ Cons arg_1 arg_2)))))))
```

Статическая корректность

Пусть

- P – типизированная функциональная программа,
- P_{rel} – типизированная реляционная программа, являющаяся результатом преобразования P.

Тогда, если P имеет тип t, то P_{rel} имеет тип $[t]$ с точностью до переименования связанных переменных.

Реализация

- Входной язык – OCaml
- Выходной язык – OCanren
- Язык реализации – OCaml

Результаты

- Предложен и формально описан метод преобразования функций высшего порядка в реляционную форму
- Доказана его статическая корректность