ТЕОРІЯ МОВ ПРОГРАМУВАННЯ

Лекція 3

Мова PCF

Три способи завдання семантики мови

Семантика мови програмування це бінарне відношення на множині термів мови. Оскільки ми вже визначили поняття мови і ввели інструменти для визначення відносин, все готово для опису трьох основних підходів до визначення семантики. Семантика мови зазвичай задається функцією, індуктивним визначенням або рефлексивно-транзитивним замиканням явно заданого відносини. Ці три підходи називаються відповідно *денотаційною семантикою, операційною семантикою з великим кроком і операційною семантикою з малим кроком*.

Денотаційна семантика

Денотаційна семантика корисна для детермінованих мов. В цьому випадку для кожної програми **p** відношення вхідних і вихідних даних, яке визначається програмою, є функцією, що позначається **||p||**. Відношення ‹→ визначено наступною умовою:

p, e ‹→ s тоді і тільки тоді, коли **||p||e= s**.

Звичайно, це просто відкладає рішення задачі до моменту визначення функції **||p||**. Для цього будуть використані два засоби: явні визначення функцій і теорема про нерухому точку.

Операційна семантика з великим кроком

Операційна семантика з великим кроком також називається структурною операційною семантикою (С.О.С.) або природною семантикою. Вона дає індуктивне визначення відношення $\langle , \sigma \rangle \to$.

Операційна семантика з малим кроком

Операційна семантика з малим кроком також називається редукційною семантикою. Вона визначає відношення $\cdot\rightarrow$ в термінах іншого відношення B, яке описує елементарні кроки для послідовного перетворення вхідного терма `t` до результуючого терма `s`.

Наприклад, якщо запустити програму `fun x → (x * x) + x` з входом `4`, буде отриманий результат `20`. Але терм `(fun x → (x * x) + x) 4` не перетворюється в `20` за один крок, спочатку він перетворюється до `(4 * 4) + 4`, а вже потім до `16 + 4` і тільки потім до `20`.

Найбільш важливо не те відношення, яке пов'язує `(fun x → (x * x) + x) 4` і `20`, а `B`, яке пов'язує `(fun x → (x * x) + x) 4` і `(4 * 4) + 4` , а потім - терм `(4 * 4) + 4` з `16 + 4` і, нарешті, терм `16 + 4` з `20`.

Як тільки визначено ставлення може бути отримано с допомогою рефлексивно-транзитивного замикання `B*` відношення `B: t ‹→ s` тоді і тільки тоді, коли `tB*s` і `s` надалі не редукуємо.

Той факт, що `s` нередуціруємо, означає, що всередині `s` більше нічого обчислювати. Наприклад, терм `20` нередуціруємий, а терм `16 + 4` - ні. Терм s нередуціруем, якщо не існує терма `s'`, такого що `sBs'`.

Незавершені обчислення

Виконання програми може видавати результат, приводити до помилки або взагалі не завершаться. Помилки можна розглядати як особливий вид результатів. Для тих програм, що не завершуються, існує кілька способів визначення їх семантики. Перша можливість полягає в тому, щоб вважати, що не існує пари `(t, s)` щодо $\not\to$, якщо `t` не закінчується. Інша альтернатива передбачає додавання спеціального елемента $\bot$ (читається: «дно») в множину можливих вихідних значень і пар `(t, ⊥)` в відношення $\not\to$ для всіх термів `t`, які не закінчуються.

Різниця між даними підходами може здаватися незначним: не так вже й складно видалити всі пари виду `(t,⊥)` або додати такі, якщо стосовно немає жодної пари виду `(t,s)`. Однак читачі, знайомі з проблемами теорії алгоритмів, помітять, що при додаванні пар `(t,⊥)` ставлення $\leftarrow$ перестає бути рекурсивно перелічуваних.

МОВА PCF

У цій лекції ми продемонструємо кілька стилів визначення семантики мови програмування, користуючись одним прикладом: мовою PCF - мовою програмування обчислюваних функцій (Programming language for computable functions), званим також Mini-ML.

Функціональна мова PCF

Програми як функції

Раніше ми спостерігали за тим, як детермінована програма обчислює значення функції, і вивели з цього спостереження принципи денотаційної семантики. Той же процес обчислення є основою цілого класу мов програмування - функціональних мов, таких як Caml, Haskell або Lisp, які традиційно використовуються на початку вивчення теорії мов програмування.

Мета цих мов полягає в зближенні понять програми і математичної функції. Іншими словами, ідея в тому, щоб наблизити програми до їх денотаційної семантики.

Основними конструкціями мови PCF будуть явне побудова функції, що записується як **`fun x → t`**, і застосування функції до аргументу, що позначається **`t u`**.
Крім цього, PCF містить константи для кожного натурального числа (включно з **`0`**), операції **`+, -, *, /`**, а також перевірку на нуль **`ifz t then u else v`**. Додавання і множення визначені для всіх натуральних чисел, щоб зробити таким віднімання, будемо використовувати угоду: якщо **`n <m`**, то **`n-m=0`**. Ділення є звичайним цілочисельним діленням, а ділення на **`0`** призводить до помилки.

Функції як об’єкти першого класу

У багатьох мовах програмування допускається визначення функції, яка або приймає іншу функцію як аргумент, або повертає її в якості результату, проте часто це вимагає деякого спеціального синтаксису, що відрізняється від синтаксису для передачі звичайних аргументів, таких як цілі числа або рядки. У функціональних мовах функції визначаються однаково, незалежно від того, чи є їхні аргументи числами або ж функціями.

Наприклад, композиція функції з самою собою визначається як `fun f → fun x → f (f x)`.

Щоб позначити той факт, що функції не розглядаються спеціальним чином, а значить, що їх можна використовувати як аргументи або результати інших функцій, ми будемо говорити, що функції є *об'єктами першого класу*.

Функції з декількома аргументами

У мові PCF немає символу для побудови функцій з кількома аргументами. Ці функції визначаються як функції з одним аргументом, використовуючи ізоморфізм **(A × B) → C = A → (B → C)**. Наприклад, функція, яка має у відповідність числах x і y вираз **x*x+y*y**, визначається як функція, яка має у відповідність числу **x** функцію, яка в свою чергу ставить у відповідність числу **y** вираз **x*x+y*y**. Остаточно отримуємо: **fun x → fun y → x * x + y * y**.

Тепер, щоб застосувати функцію **`f`** до параметрів **`3`** і **`4`** необхідно спочатку застосувати її до **`3`**, отримавши терм **`f 3`**, який представляє функцію, яка ставить у відповідність числу у вираз **`3 * 3 + y * y`**, а потім до **`4`**, отримавши терм **`(f 3)4`**. Так як на нашу угодою застосування асоціюється зліва направо, останній терм можна записати просто як **`f 3 4`**.

Відсутність присвоювань

На відміну від таких мов, як Caml або Java, важливою рисою PCF є повна відсутність присвоювання. У неї немає конструкцій вигляду `x: = t` або `x = t` для позначення привласнення «змінній» нового значення.

Рекурсивні визначення

Далеко не всі математичні функції можуть бути явно визначені. У шкільній програмі, наприклад, функцію зведення в ступінь зазвичай визначають як

$$\mathtt{x}, \mathtt{n} \mapsto \underbrace{\mathtt{x} \times \cdots \times \mathtt{x}}_{\mathtt{n}\ \text{раз}}$$

або за допомогою визначення по індукції.

У мовах програмування використовуються схожі конструкції: цикли та рекурсивні визначення. PCF включає спеціальну конструкцію для визначення рекурсивних функцій.

Часто кажуть, що функція є рекурсивної, якщо вона використовується у власному визначенні. Це, однак, нісенітниця: в мовах програмування, як і взагалі всюди, при спробі використання подібних визначень ми потрапляємо в порочне коло. Не можна «визначити» функцію **`fact`** як **`fun n → ifz n then 1 else n * (fact (n - 1))`**. І взагалі, ми не можемо визначити функцію **`f`** термом **`G`**, що містить входження **`f`**. Однак ми можемо визначити функцію **`f`** як нерухому точку функції **`fun f → G`**. Наприклад, функцію **`fact`** можна визначити як нерухому точку функції **`fun f → fun n → ifz n then 1 else n * (f (n − 1))`**.

Чи має ця функція нерухому точку? І якщо має, чи є ця нерухома точка єдиною? А якщо вона не єдина, то яку з них ми отримуємо? Залишимо поки ці питання без відповіді, вважаючи просто, що рекурсивна функція визначається як нерухома точка.

У мові PCF символ fix пов'язує змінну в своєму аргументі, а терм **`fix f G`** використовується для позначення нерухомої точки функції **`fun f → G`**. Функція **`fact`** тепер може бути визначена як **`fix f fun n → ifz n then 1 else n * (f (n - 1))`**.

Зауважимо знову, що використання символу **`fix`** дозволяє нам побудувати функцію обчислення факторіала без необхідності явно вказувати її ім'я.

Визначення

Теоретично можна взагалі відмовитися від визначень, замінюючи всюди символи, що визначаються, на їх визначення. Однак з використанням визначень програми стають простіші і зрозуміліші.

Додамо в мову PCF останню конструкцію: **`let x = t in u`**. Входження змінної **`x`** в **`u`** є пов'язаними, а її ж входження в **`t`** немає. Символ **`let`** є бінарним оператором, що зв'язує змінну в другому аргументі.

Опис мови PCF

Мова PCF містить:

- символ **`fun`** з одним аргументом, який пов'язує змінну в своєму аргументі;
- символ **`α`** з двома аргументами, який не пов'язує ніякі змінні (застосування **`α (t, u)`**, спрощено записується як **`t u`**);
- нескінченне число констант, що представляють натуральні числа;
- чотири символи **`+, -, *, /`** с двома аргументами кожен, які не пов'язують ніякі змінні;
- символ **`ifz`** з трьома аргументами, який не пов'язує ніякі змінні;

- символ **`fix`** з одним аргументом, який пов'язує змінну в своєму аргументі;
- символ **`let`** з двома аргументами, який пов'язує змінну в другому аргументі.

Іншими словами, синтаксис PCF визначається індуктивно наступним чином:

```
t = x
  | fun x → t
  | t t
  | n
  | t + t | t - t | t * t | t/t
  | ifz t then t else t
  | fix x t
  | let x = t in t
```

Незважаючи на такий малий розмір, мова PCF є повним за Тьюрінгом, тобто на ньому можна запрограмувати будь-яку обчислюваної функції.

Операційна семантика з малим кроком

Правила

Застосуємо програму `fun x → 2 * x` до константи `3`. Вийде терм `(fun x → 2 * x) 3`. Тепер, згідно з принципами операційної семантики з малим кроком, будемо обчислювати значення цього терма крок за кроком з тим, щоб, якщо пощастить, отримати в результаті `6`. Першим кроком в процесі спрощення терма є передача параметра, тобто заміна формального параметра `x` на фактичний аргумент `3`. Після нього вихідний терм перетворюється в `2 * 3`. На другому кроці обчислень терм `2 * 3` дає результат `6`.

Перший малий крок, передача параметра, може виконуватися щоразу , коли терм має вигляд **`(fun x → t) u`**, тобто функція **`fun x → t`** застосовується до аргументу **`u`**. Як наслідок, ми визначили правило, зване правилом **β**-редукції:

`(Fun x → t) u - → (u / x) t.`

Ставлення **t - → u** читається як «**t** редукується (або зводиться) до **u**».
Другий зі згаданих раніше кроків можна узагальнити так:
p ⊗ q - → n (якщо **p ⊗ q = n**),
де **⊗** є однією з чотирьох певних вище операцій, включених в мову PCF. Додамо аналогічні правила для умов:
ifz 0 then t else u-→t,
ifz n then t else u-→u (якщо **n** - число, відмінне від **0**);
правило для нерухомих точок:
fix x t - → (fix x t / x) t;
і правило для **let**:
let x = t in u - → (t / x) u.

Редексом назвемо терм, який можна редукувати. Іншими словами, терм `t` є редексом, якщо існує такий терм `u`, що `t` $\rightarrow$ `u`.

Числа

Можна абсолютно справедливо зауважити, що правило
`p ⊗ q - → n` (якщо **`p ⊗ q = n`**),
прикладом застосування якого є перетворення **`2 * 3 - → 6`**, насправді не пояснює семантику арифметичних операцій, воно лише замінює відповідну операцію PCF на математичну. Цей вибір, проте, пояснюється тим фактом, що ми аніскільки не цікавимося семантикою арифметичних операцій, замість цього нашою метою є дослідження семантики інших мовних конструкцій.

Для визначення семантики арифметичних операцій в PCF без звернення до їх математичного змістом було б розглядати версію мови PCF без числових констант, в якій є тільки одна константа для числа `0` і символ `S` - «послідовник» (successor) - з одним аргументом. Число `3`, наприклад, можна подати у такому випадку у вигляді терма `S (S (S (0)))`.

Потім можна додати правила малих кроків:

$$0 + u \longrightarrow u$$

$$S(t) + u \longrightarrow S(t + u)$$

$$0 - u \longrightarrow 0$$

$$t - 0 \longrightarrow t$$

$$S(t) - S(u) \longrightarrow t - u$$

$$0 * u \longrightarrow 0$$

$$S(t) * u \longrightarrow t * u + u$$

$$t/S(u) \longrightarrow \textbf{ifz } t - u \textbf{ then } 0 \textbf{ else } S((t - S(u))/S(u))$$

Зауважимо, що для більшої строгості варто було б додати правило ділення на `0`, яке порушувало б виняток: `error`.

Еквівалентність

Використовуючи правила семантики з малим кроком, отримуємо:

`(Fun x → 2 * x) 3 -→ 2 * 3 -→ 6`.

Таким чином, позначивши через **`-→*`** рефлексивно-транзитивне замикання відношення **`-→`**, можна записати **`(fun x → 2 * x) 3 -→ * 6`**.

Однак це визначення відносини **`-→*`** не дозволяє редукувати терм **`(2 + 3) + 4`** до терму **`9`**. Дійсно, щоб редукувати терм виду **`t + u`** терми **`t`** і **`u`** повинні бути числовими константами, тоді як в нашому прикладі терм **`2 + 3`** є сумою, а не константою.

Першим кроком слід було б обчислити `2 + 3`, що дає число `5`. Тоді другий крок редукує `5 + 4` до `9`. Таким чином, проблема полягає в тому, що, згідно з нашим визначенням, `2 + 3` редукується до `5`, але `(2 + 3) + 4` НЕ редукується до `5 + 4`. Необхідно визначити інше відношення, в якому правила можна застосовувати до будь-якого подтерму редукованих терма. Побудуємо індуктивне визначення відношення : ▷

$$\frac{\mathtt{t} \rhd \mathtt{u}}{\mathtt{t\ v} \rhd \mathtt{u\ v}},$$

$$\frac{\mathtt{t} \rhd \mathtt{u}}{\mathtt{v\ t} \rhd \mathtt{v\ u}},$$

$$\frac{\mathtt{t} \rhd \mathtt{u}}{\mathbf{fun}\ \mathtt{x} \to \mathtt{t} \rhd \mathbf{fun}\ \mathtt{x} \to \mathtt{u}},$$

$$\frac{\mathtt{t} \rhd \mathtt{u}}{\mathtt{t} + \mathtt{v} \rhd \mathtt{u} + \mathtt{v}},$$

...

Можна показати, що терм є редексом згідно відношенню $\triangleright$, якщо і тільки якщо один з його підтермів є редексом згідно відношенню $\multimap$.

Приклад

Як приклад застосування правил семантики з малим кроком для мови PCF обчислимо факторіал `3`.

```
(fix f fun n → ifz n then 1 else n*(f (n - 1))) 3
```

▷ `(fun n → ifz n then 1 else`

```
    n*((fix f fun n → ifz n then 1 else n*(f (n - 1))) (n - 1))) 3
```

▷ `ifz 3 then 1 else`

```
    3*((fix f fun n → ifz n then 1 else n*(f (n - 1))) (3 - 1))
```

▷ `3*((fix f fun n → ifz n then 1 else n*(f (n - 1))) (3 - 1))`

▷ `3*((fix f fun n → ifz n then 1 else n*(f (n - 1))) 2)`

▷ `3*((fun n → ifz n then 1 else`

```
    n*((fix f fun n → ifz n then 1 else n*(f (n - 1))) (n - 1))) 2)
```

$\triangleright$ `3*(2*((fix f fun n` $\rightarrow$ **ifz** `n then 1 else n*(f (n - 1))) (2 - 1)))`

$\triangleright$ `3*(2*((fix f fun n` $\rightarrow$ **ifz** `n then 1 else n*(f (n - 1))) 1))`

$\triangleright$ `3*(2*((fun n` $\rightarrow$ **ifz** `n then 1 else n*`

`((fix f fun n` $\rightarrow$ **ifz** `n then 1 else n*(f (n - 1)))(n - 1))) 1))`

$\triangleright$ `3*(2*(ifz 1 then 1 else`

`1*((fix f fun n` $\rightarrow$ **ifz** `n then 1 else n*(f (n - 1))) (1 - 1))))`

$\triangleright$ `3*(2*(1*`

`((fix f fun n` $\rightarrow$ **ifz** `n then 1 else n*(f (n - 1))) (1 - 1))))`

$\triangleright$ `3*(2*(1*((`**fix** `f` **fun** `n` $\rightarrow$ **ifz** `n` **then** `1` **else** `n*(f (n-1))) 0)))`

$\triangleright$ `3*(2*(1*((`**fun** `n` $\rightarrow$ **ifz** `n` **then** `1` **else**

`n*((`**fix** `f` **fun** `n` $\rightarrow$ **ifz** `n` **then** `1` **else** `n*(f (n-1))) (n-1))`

`) 0)))`

$\triangleright$ `3*(2*(1*((`**ifz** `0` **then** `1` **else**

`0*((`**fix** `f` **fun** `n` $\rightarrow$ **ifz** `n` **then** `1` **else** `n*(f (n-1)))(0-1))))))`

$\triangleright$ `3*(2*(1*1))` $\triangleright$ `3*(2*1)` $\triangleright$ `3*2` $\triangleright$ `6`

Нередуцьовані замкнені терми

Терм **t** називається нередуціруемого, якщо його не можна редукувати за допомогою відносини Δ , тобто, якщо не існує такого терма **u**, що **t** Δ **u**.

Тепер можна визначити відношення «терм **u** є результатом обчислення терма **t**», де **t** - замкнутий термін, в такий спосіб: **t** ‹→ **u**, якщо і тільки якщо **t** Δ * **u**, причому **u** нередуцьований. В цьому випадку терм **u** повинен бути замкнутий. Нарешті, ставлення «програма **p** з вхідними даними $e_1, \ldots, e_n$ повертає **s** »можна записати просто як **p** $e_1 \ldots e_n$ ‹→ **s**.

Числа і нередуцьовані замкнуті терми виду **`fun x → t`** називаються значеннями. Якщо результат обчислення є значенням, ми пов'язуємо значення з вихідним термо і говоримо, що це значення є значенням терма (терм обчислюється до цього значення).

Нажаль, значення не є єдино можливими результатами. Наприклад, терм `(fun x → x) 1 2` можна редукувати до терма `1 2`, який нередуціруем і замкнутий, тобто дійсно є результатом обчислення терма `(fun x → x) 1 2`. Однак цей результат не має сенсу, тому що ми не можемо застосувати одиницю, яка не є функцією, до двійці.

Нередуцьовані замкнуті терми, які не є значеннями, називають тупиковими (stuck). Вони можуть приймати одну з наступних форм: $V_1\ V_2$, де V_1 і V_2 нередуцьовані замкнуті терми, причому V_1 не є функцією **fun x → t** (наприклад, **1 2**); $V_1 \otimes V_2$, де V_1 і V_2 нередуцьовані, замкнуті і хоча б один з них не є числом (наприклад, **1 + (fun x → x)**); **ifz** V_1 **then** V_2 **else** V_3, де V_1, V_2 і V_3 нередуцьовані, замкнуті і V_1 не є числом (наприклад, **ifz (fun x → x) then 1 else 2**).

Незавершувані обчислення

Легко побачити, що відношення $\not\to$ не є тотальним, тобто існують терми `t`, для яких не існує такого терма `u`, що `t` $\not\to$ `u`. Наприклад, терм `b = fix x x` редукується до себе і тільки до себе. Його неможливо звести до жодного нередуцьованого терма.

Таким чином, можна стверджувати, що символ **`fix`** з PCF є в деякому сенсі надмірною. Однак ця надмірність згодом, коли в PCF будуть додані типи, зникне.

Злиття

Чи можливо в процесі редукції отримати з замкнутого терма кілька різних результатів? І взагалі, чи можна звести терм до декількох різних нередуцьованих термах? Відповідь на ці питання негативна. Фактично кожна PCF-програма є детермінованою, але це не така вже тривіальна властивість. Давайте з'ясуємо, чому.

Терм `(3 + 4) + (5 + 6)` має два підтерма, кожен з яких є редексом. Ми можемо почати, редукувати спочатку `3 + 4` до `7` або `5 + 6` до `11`. Дійсно, терм `(3 + 4) + (5 + 6)` редукується і до `7 + (5 + 6)`, і до `(3 + 4) + 11`. На щастя, обидва ці терма можна редукувати далі, і, продовжуючи обчислення, ми приходимо до одного і того ж результату `18` в обох випадках.

Щоб довести, що будь-який терм можна звести не більше ніж до одного нередуцьованого терму, потрібно показати, що якщо два обчислення, що стартували на одному і тому ж термі, привели до двох різних термів, то вони неминуче рано чи пізно досягнуть одного і того ж нередуцьованого терма.

Цей факт є наслідком того, що відношення Δ має властивість злиття. Кажуть, що відношення **R** має властивість злиття, якщо кожного разу, коли має місце **a R * b_1** і **a R * b_2**, існує деякий **c**, таке що **b_1 R * c** і **b_2 R * c**.

Властивість злиття в дійсності тягне за собою те, що будь-який терм може бути зведений до не більше ніж одному нередуціруемого терму. Якщо терм **t** можна редукувати до двох далі нередуціруемого термам u_1 і u_2, то **t** Δ * u_1 і **t** Δ * u_2. В силу властивості злиття відносини Δ існує такий терм **v**, що u_1 Δ * **v** і u_2 Δ * **v**. Так як u_1 нередуцьований, то єдиний такий терм **v**, що u_1 Δ * **v**, є сам u_1. Отже, $u_1 = v$ і точно так же $u_2 = v$. Звідси отримуємо, що $u_1 = u_2$. Іншими словами, **t** зводиться до не більше ніж одному нередуцьованого терму.

Ми не будемо доводити тут властивість злиття відношення Δ. Ідея полягає в тому, що коли терм **t** містить два редекса $\mathbf{r_1}$ і $\mathbf{r_2}$, терм $\mathbf{t_1}$ отриманий редуцируванням $\mathbf{r_1}$, а терм $\mathbf{t_2}$ отриманий редуцированием $\mathbf{r_2}$, то в термі $\mathbf{t_1}$ можна знайти залишки редекса $\mathbf{r_2}$ і редукувати їх. Схожим чином можна редукувати залишки $\mathbf{r_1}$ в $\mathbf{t_2}$, отримавши той же терм. Наприклад, редуціруя **5 + 6** в **7 + (5 + 6)** та редуцируя **3 + 4** в **(3 + 4) + 11**, ми отримуємо один і той же терм **7 + 11**.

Стратегії редукції розглянемо на наступній лекції.

На наступній лекції ми продовжимо розглядати основи теорії мов програмування.