4471028: 프로그래밍언어

Lecture 5 — 자기호출 프로그램 유도하기
Deriving Recursive Programs

임현승
2020 봄학기

개요

- 귀납적으로 정의된 집합을 표현하기 위한 OCaml 데이터 타입 정의
- 귀납 정의로부터 자기호출 프로그램 유도하기

귀납적으로 정의된 집합

- 귀납적으로 정의된 집합은 프로그래밍 언어에서의 타입
  - 예: nat, bool, list, tree
- 귀납적으로 정의된 집합 nat:

$$\text{nat} \quad n ::= \mathsf{o} \mid \mathsf{s}\, n$$

해석:

$$\begin{aligned} \mathsf{o} &\equiv 0 \\ \mathsf{s}\,\mathsf{o} &\equiv 1 \\ \mathsf{s}\,\mathsf{s}\,\mathsf{o} &\equiv 2 \\ \mathsf{s}\,\mathsf{s}\,\mathsf{s}\,\mathsf{o} &\equiv 3 \\ &\vdots \end{aligned}$$

- nat은 자연수를 값(원소)로 갖는 타입(집합)

OCaml 데이터 타입 정의

귀납적으로 정의된 집합 nat:

$$\text{nat} \quad n ::= \mathtt{O} \mid \mathtt{S}\ n$$

OCaml 데이터 타입 정의:

```
type nat = Zero | Succ of nat ;;
```

문법:

```
type <name> = <constructor>_1 | ... | <constructor>_n ;;
```

- `<name>` 정의하려는 타입(집합)의 이름
- `<constructor>`$_i$ 집합의 원소를 만들기 위한 생성자(constructor)
- 생성자는 인자를 가질 수 있음, 예: `Succ of nat`

```
<constructor> ::= <constructor_name>
                | <constructor_name> of <type of arguments>
```

nat 값(원소) 만들기

```
# Zero;;
- : nat = Zero
# Succ Zero;;
- : nat = Succ Zero
# Succ (Succ Zero);;
- : nat = Succ (Succ Zero)
```

데이터 타입의 원소를 검사하는 방법

- 데이터 타입 T가 n개의 생성자를 가질 때,
- T 타입의 표현식 e를 계산한 결과를 분석하기 위해 패턴 매칭(pattern matching)을 이용할 수 있다:

```
match e with
| <constructor>_1 -> <expression>_1
...
| <constructor>_n -> <expression>_n
```

- e의 계산 결과가 <constructor>$_i$와 같다면, 패턴 매칭은 <expression>$_i$를 계산
- 패턴 매칭은 e의 계산 결과를 패턴 <constructor>$_1$부터 <constructor>$_n$까지 매칭이 성공할 때까지 차례대로 비교 (first match policy).

예제 1

인자로 주어진 nat 값이 Zero인지 검사하는 함수:

```
(* val is_zero : nat -> bool *)
let is_zero n =
  match n with
  |         ->
  |         ->
```

예제 1

인자로 주어진 nat 값이 Zero인지 검사하는 함수:

```
(* val is_zero : nat -> bool *)
let is_zero n =
  match n with
  | Zero   ->
  | Succ _ ->
```

- 본 예제에서는 n의 값이 Succ p의 형태일 때, p의 실제 값은 중요하지 않음
- 따라서 패턴 Succ _에서 와일드카드 패턴(wildcard pattern) '_'을 사용

예제 1

인자로 주어진 nat 값이 Zero인지 검사하는 함수:

```
(* val is_zero : nat -> bool *)
let is_zero n =
  match n with
  | Zero   -> true
  | Succ _ -> false
```

- 본 예제에서는 n의 값이 Succ p의 형태일 때, p의 실제 값은 중요하지 않음
- 따라서 패턴 Succ _에서 와일드카드 패턴(wildcard pattern) '_'을 사용

예제 2

인자로 주어진 nat 값을 정수(integer) 값으로 변환하는 함수(자기호출 함수를 정의해야 함):

```
(* val trans : nat -> int *)
let rec trans n =
   match n with
   | Zero   ->
   | Succ m ->
```

예제 2

인자로 주어진 nat 값을 정수(integer) 값으로 변환하는 함수(자기호출 함수를 정의해야 함):

```
(* val trans : nat -> int *)
let rec trans n =
   match n with
   | Zero   -> 0
   | Succ m -> 1 + trans m
```

자기호출 프로그램 유도하기

The Smaller-Subproblem Principles

If we can reduce a problem to a smaller subproblem, we can call the procedure that solves the problem to solve the subproblem.

귀납적으로 정의된 집합에 대해(예: OCaml 데이터 타입) 자기호출 함수를 정의할 경우,

1. (Base case) 문제를 바로 풀면 되고,
2. (Inductive case) 귀납 가정에 해당되는 하위 문제(subproblem)를 자기 호출(recursive call)로 해결하고, 그 결과를 이용하여 원래 문제(original problem)를 풀면 된다.

리스트

$$\begin{array}{lllll} \text{list} & l & ::= & [] & \\ & & | & n :: l & (n \in \mathbb{Z}) \end{array}$$

OCaml 데이터 타입 정의:

```
type list = Nil | Cons of int * list ;;
```

- `Cons`는 인자 두 개를 받음: `int` 타입 인자 한 개, `list` 타입 인자 한개
- `list` 데이터 타입은 이미 OCaml에 정의되어 있음. 다음과 같은 리스트 연산을 자유롭게 사용 가능: `[]`, `::`
- 예: `[]`, `1::2::3::[]`, `[1; 2; 3]`

그런데 `list`는 원소들의 타입이 모두 동일하다면(homogeneous) 그 타입이 어떤 타입이든 상관없이 저장할 수 있는 일반적인 데이터 구조

- 원소의 타입에 따라 새로운 리스트 데이터 구조를 정의해야 할까? 예: `int_list`, `bool_list`, `string_list`, 등.

다형 리스트 데이터 타입(Polymorphic List Datatype)

```
type 'a list = Nil | Cons of 'a * 'a list ;;
```

해석:

- 'a는 *아무 타입*(any type)을 의미
- 아무 타입 'a에 대해
  - Nil은 'a list 타입
  - x가 'a 타입이고 xs가 'a list 타입이면, Cons (x, xs)는 'a list 타입

```
# Nil;;
- : 'a list = Nil
# Cons (1, Nil);;
- : int list = Cons (1, Nil)
# Cons (1.0, Cons (2.0, Nil));;
- : float list = Cons (1., Cons (2., Nil))
# Cons (1, Cons (2.0, Nil));;
Error: This expression has type float but an expression was
         expected of type int
```

자기호출 함수 예

리스트의 길이(length) 계산하기:

```
(* val length : 'a list -> int *)
let rec length l =
  match l with
  |          ->
  |          ->
```

리스트 원소의 합 계산하기:

```
(* val sum : int list -> int *)
let rec sum l =
  match l with
  |          ->
  |          ->
```

자기호출 함수 예

리스트의 길이(length) 계산하기:

```
(* val length : 'a list -> int *)
let rec length l =
  match l with
  | []      ->
  | x :: xs ->
```

리스트 원소의 합 계산하기:

```
(* val sum : int list -> int *)
let rec sum l =
  match l with
  | []      ->
  | x :: xs ->
```

자기호출 함수 예

리스트의 길이(length) 계산하기:

```
(* val length : 'a list -> int *)
let rec length l =
  match l with
  | []      -> 0
  | x :: xs -> 1 + length xs
```

리스트 원소의 합 계산하기:

```
(* val sum : int list -> int *)
let rec sum l =
  match l with
  | []      ->
  | x :: xs ->
```

자기호출 함수 예

리스트의 길이(length) 계산하기:

```
(* val length : 'a list -> int *)
let rec length l =
  match l with
  | []      -> 0
  | x :: xs -> 1 + length xs
```

리스트 원소의 합 계산하기:

```
(* val sum : int list -> int *)
let rec sum l =
  match l with
  | []      -> 0
  | x :: xs -> x + sum xs
```

이진 나무

$$\text{tree} \quad t ::= \texttt{leaf} \mid \texttt{node}(n, t, t) \quad (n \in \mathbb{Z})$$

OCaml 데이터 타입 정의:

```
type tree = Leaf | Node of int * tree * tree
```

Exercise 1: `Leaf`의 개수를 세는 함수를 작성하라.

```
(* val leaves : tree -> int *)
let rec leaves t =
  match t with
   |
   |
```

Exercise 2: `node`의 개수를 세는 함수를 작성하라.

```
(* val nodes : tree -> int *)
let rec nodes t =
  match t with
   |
   |
```

정수식

$$\begin{array}{lrll} \text{exp} & e & ::= & n \quad (n \in \mathbb{Z}) \\ & & \mid & -e \mid e + e \mid e * e \end{array}$$

OCaml 데이터 타입 정의:

```
type exp =
  | INT of int
  | MINUS of exp
  | ADD of exp * exp
  | MUL of exp * exp
```

Exercise: 정수식의 값을 계산하는 해석기(interpreter)를 작성하라.

```
(* val interp : exp -> int *)
let rec interp e =
  match e with
  | Int n -> n
  | ...
```

명제 논리

formula $A, B ::= T \mid F \mid \neg A \mid A \wedge B \mid A \vee B \mid A \Rightarrow B$

OCaml 데이터 타입:

```
type formula = True | False
             | Neg of formula
             | Or of formula * formula
             | And of formula * formula
             | Imply of formula * formula
```

Exercise: 논리식(formula)의 진리값을 계산하는 해석기를 작성하라.

```
(* val eval : formula -> bool *)
let rec eval f =
  match f with
  | True -> true
  | ...
```