프로그래밍 언어의 원리

(Introduction to Programming Language Principles)

오학주

September 7, 2022

This draft book may contain errors. Please let me know (hakjoo_oh@korea.ac.kr) if you find any errors or have suggestions.

목차

서문

고려대학교에서 학부 2학년 학생들을 대상으로 강의하는 “프로그래밍 언어(COSE 212)” 과목의 내용을 정리한 책이다. 프로그래밍 언어 분야를 처음 접하는 학생들에게 다소 생소한 내용임에도 불구하고 마땅한 교재가 없어서 학생들이 어려움을 겪어왔다. 이 책이 수강생들과 프로그래밍 언어를 공부하고자 하는 학생들에게 도움이 되기를 바란다.

내용 프로그래밍 언어를 직접 디자인하고 구현해가면서 프로그래밍 언어들이 가지는 주요 원리들을 이해하는 것이 목표이다. 1장과 2장은 이를 위한 기본기에 해당한다. 3장부터는 작은 프로그래밍 언어에서 시작하여 점진적으로 언어를 확장해나간다. 4장에서 함수를 정의하고 사용할 수 있도록 언어를 확장하고 함수와 관련된 개념들을 소개한다. 5장에서는 리스트 등을 추가하여 좀 더 그럴듯한 함수형 프로그래밍 언어를 학생들이 직접 설계하고 구현해 볼 수 있도록 하였다. 6장과 7장에서는 명령형 언어의 특징을 가지도록 언어를 확장하는 방안을 알아본다. 8장에서는 프로그래밍 언어에 정적 타입 시스템을 장착하여 프로그램 실행전에 특정 종류의 오류들을 검출해내는 기술을 체험할 수 있도록 하였다. 9장은 다소 독립적인 내용으로 프로그래밍 언어의 기원에 해당하는 람다 계산식을

소개하고 있다.

이 책을 강의에서 사용한다면 언어 설계에 대한 내용을 수업에서 전달하고, 수업에서 설계한 언어의 구현을 학생들에게 과제로 부여하는 방식이 적당할 것이다. 이를 위해 대부분의 장 마지막에 프로그래밍 과제로 적당한 구현 문제들을 수록하였다. 이 문제들을 모두 직접 풀어본다면 프로그램 실행기(interpreter)와 분석기(analyzer)를 설계하고 구현하는 데 필요한 기본 능력을 갖출 수 있게 된다. 이 책에서 다루는 내용은 강의를 실습을 병행하여 진행할 경우 한 학기 분량에 적합하고, 강의만 진행하는 경우에는 2/3학기 정도의 분량에 해당할 것이다.

참고 문헌 이 책을 쓰면서 아래 서적들을 참고하였다.

- Daniel P. Friedman and Mitchell Wand. Essentials of Programming Languages (3rd edition). MIT Press. 2008.
- 이광근. 프로그래밍 언어 이야기. 2006
- Benjamin C. Pierce. Types and Programming Languages. MIT Press. 2002.

특히 처음 두 책은 지금까지 COSE 212 프로그래밍 언어 과목에서 교재로 사용하던 책들이다. 이 책의 전개 방식과 내용에 있어서 큰 영향을 받았고 각색한 것이 많음을 밝혀둔다. 프로그래밍 언어와 타입 시스템에 대한 더 깊은 내용을 원한다면 세 번째 책을 읽으면 좋을 것이다.

1

귀납법

모든 프로그래밍 언어는 귀납적으로 정의된다. 그래서 프로그래밍 언어 자체는 유한하지만 그 언어로 작성할 수 있는 프로그램의 개수에는 제한이 없다. 이 장에서는 프로그래밍 언어 공부의 가장 기본이 되는 귀납법을 익힌다.

1.1 집합의 귀납적 정의

귀납법은 자기 자신을 이용하여 집합을 정의하는 방법을 말한다. 예를 들어, 집합 S를 다음의 두 조건을 만족하는 가장 작은 자연수 집합으로 정의해 보자.

1. S는 0을 포함한다 ($0 \in S$).
2. S가 자연수 n을 포함하면, S는 $n+3$을 포함한다 ($n \in S \implies n+3 \in S$).

이 정의를 귀납적이라 부르는 이유는 S에 속하는 원소들을 정의하는 데 있어서 S에 이미 속해있는 원소들을 이용하기 때문이다.

이렇게 정의된 집합 S는 무엇일까? 먼저 첫 번째 조건에 의해 집합 S는 0을 포함해야 한다. S가 0을 포함하면 두 번째 조건에 의해 3을 포함해야 한다. 마찬가지로 S가 3을 포함하면 6을 포함

해야 한다. 이 과정을 반복하면 집합 S는 3의 배수들을 모두 포함해야 함을 알 수 있다. 하지만 두 조건을 만족하는 집합이 항상 3의 배수만 포함하는 것은 아니다. 예를 들어, 자연수 전체의 집합 $\{0, 1, 2, 3, \ldots\}$도 위의 두 조건을 만족하고, 집합 $\{0, 3, 6, 9, \ldots\} \cup \{1, 4, 7, 10, \ldots\}$도 두 조건을 만족한다. 이런식으로 두 조건을 만족시키는 집합은 무한히 많이 존재한다. 따라서 위의 정의에서는 두 조건을 만족하는 "가장 작은 집합"으로 S를 정의하고 있고 그 집합이 3의 배수의 집합($\{0, 3, 6, 9, \ldots\}$)인 것이다. 3의 배수의 집합은 두 조건을 만족하고, 두 조건을 만족하는 임의의 집합에 포함된다.

위의 예에서처럼 집합의 귀납적 정의는 두 가지 종류의 규칙으로 구성된다. 첫 번째 종류는 아무 조건 없이 집합에 속하는 기초 원소들을 규정하는 규칙들이다("S는 0을 포함한다"). 두 번째 종류는 집합에 이미 속해있는 원소들을 이용하여 만들어지는 원소들을 정의하는 규칙들이다("S가 어떤 자연수 n을 포함하면, S는 $n + 3$을 포함한다"). 특히 기초 원소들은 항상 잘 정의되어 있어야 한다. 기초 원소가 없는 귀납적 정의는 항상 공집합($\emptyset$)을 의미하기 때문이다.

추론 규칙

앞으로 집합을 귀납적으로 정의할때 추론 규칙(inference rule)을 자주 사용할 것이다. 추론 규칙은 다음과 같이 생겼다.

$$\frac{A_1 \quad A_2 \quad \cdots \quad A_n}{B}$$

하나의 추론 규칙은 가정들(A_1, A_2, ..., A_n)과 결론(B)으로 구성되는데 A_1, A_2, ..., A_n들이 모두 사실이면 B도 사실임을 뜻한다. 가정을 포함하지 않는 추론 규칙을 공리(axiom)라고 부르며, 아무 조건 없이 결론이 사실임을 의미한다.

예를 들어 위에서 정의한 3의 배수들의 집합 S를 추론 규칙으로 정의하면 다음과 같다.

$$\frac{}{0 \in S} \qquad \frac{n \in S}{(n+3) \in S}$$

왼쪽 규칙은 조건 없이 명제 $0 \in S$가 사실임을 뜻한다. 즉, 집합 S는 자연수 0을 포함함을 뜻한다. 오른쪽 규칙은 집합 S가 어떤 자연수 n을 포함하면, S는 자연수 $n+3$도 포함해야 함을 뜻한다. 추론 규칙이 정의하는 집합은 두 조건을 만족하는 가장 작은 집합이지만 "가장 작은 집합"이라는 조건을 명시적으로 붙이지는 않는다.

추론 규칙을 증명 규칙(proof rule)으로 생각하여 다음과 같이 해석하면 의미가 좀 더 명확해진다. 집합 S를 위의 추론 규칙을 이용하여 명제 $n \in S$를 증명할 수 있는 자연수 n들의 집합으로 정의하는 것이다. 예를 들어, S는 자연수 6을 포함하는데 그 이유는 다음과 같이 명제 $6 \in S$를 증명할 수 있기 때문이다.

$$\dfrac{\dfrac{\dfrac{}{0 \in S}}{3 \in S}}{6 \in S}$$

증명은 공리에 해당하는 규칙에서 시작하여 증명 대상이 추론될 때

까지 추론 규칙들을 반복 적용하는 과정이다. 이러한 증명과정을 증명 나무(proof tree)라고도 부른다. 위의 예에서는 공리를 이용하여 명제 $0 \in S$를 유도한 후 오른쪽 규칙을 두 번 적용하여 $6 \in S$를 유도하였다. 이와 같이 $0 \in S, 3 \in S, 6 \in S$, $9 \in S, \ldots$들은 모두 증명 가능하므로 위의 추론 규칙이 정의하는 집합은 3의 배수들을 모두 포함한다. 반면에 $1 \in S$ 또는 $4 \in S$등은 증명할 수 없으므로 집합 S에 속하지 않는다. 따라서 집합 S는 정확히 3의 배수만을 포함하게 된다. 이렇게 추론 규칙을 증명 규칙으로 해석하여 정의할 때에는 "가장 작은 집합"이라는 조건을 따로 생각하지 않아도 규칙에 의해서 닫혀 있는 집합들 가운데 가장 작은 집합을 의미하게 된다.

표현의 간결함을 위해서 추론 규칙으로 집합을 정의할 때 현재 정의하고 있는 집합을 보통 생략한다. 예를 들어, 위의 추론 규칙은 아래와 같이 쓸 수 있다.

$$\overline{0} \qquad \frac{n}{n+3}$$

또한 추론 규칙을 다음과 같이 문맥 자유 문법(context-free grammar)을 이용하여 표현하기도 한다.

$$n \to 0 \mid n+3$$

정의하고자 하는 집합이 기초 원소로 0을 포함하고 있고, 그 집합이 n을 포함하면 $n+3$도 포함해야 한다는 뜻이다. 역시 가장 작은

집합이라는 조건은 생략한다.

귀납적 정의의 예

귀납적으로 정의되는 집합의 몇 가지 예를 살펴보자.

예제 1 아래의 추론 규칙도 3의 배수의 집합 $\{0, 3, 6, 9, \ldots\}$을 정의하고 있다.

$$\frac{}{0} \qquad \frac{}{3} \qquad \frac{x \quad y}{x+y}$$

첫 번째, 두 번째 규칙은 집합이 기초 원소로 0과 3을 포함함을 뜻한다. 세 번째 규칙은 x, y가 집합의 원소일 때 $x+y$도 원소임을 뜻한다.

예제 2 괄호 짝이 맞는 문자열(balanced parentheses)들의 집합

$$S = \{(), ()(), (()), ()()(), (()()), ()(()), (())(), ((())), \ldots\}$$

를 다음과 같은 추론 규칙으로 정의할 수 있다.

$$\frac{}{()} \qquad \frac{s}{(s)} \qquad \frac{s_1 \quad s_2}{s_1 s_2}$$

첫 번째 규칙에 의해 문자열 ()가 집합 S에 속한다. 두 번째 규칙에 의해 어떤 문자열 s가 집합 S에 속하면 (s)도 그 집합에 속해야 한다. 세 번째 규칙에 의해 문자열 s_1, s_2가 집합 S에 속하면, $s_1 s_2$도 그 집합에 속해야 한다. 예를 들어, 문자열 (()())는 다음과 같

이 만들어진다.

$$\dfrac{\dfrac{\dfrac{}{()}\ (\text{첫 번째 규칙})}{(())}\ (\text{세 번째 규칙})}{((())())}\ (\text{두 번째 규칙})$$

각 증명의 단계마다 어떤 추론 규칙을 적용하였는지를 나타내었다.

예제 3 다음과 같이 집합 S를 정의하자.

$$S = \{\mathtt{a}^n\mathtt{b}^{n+1} \mid n \in \mathbb{N}\}$$

여기서 $\mathbb{N}$은 자연수 전체의 집합, x^n은 다음과 같이 정의되는 문자열을 뜻한다(ϵ는 길이가 0인 문자열이다).

$$\begin{aligned} x^0 &= \epsilon \\ x^{i+1} &= xx^i \ (i \geq 0) \end{aligned}$$

따라서 위 집합은 $S = \{\mathtt{b}, \mathtt{abb}, \mathtt{aabbb}, \mathtt{aaabbbb}, \dots\}$를 뜻하고, 이를 추론 규칙으로 정의하면 다음과 같다.

$$\dfrac{}{\mathtt{b}} \qquad \dfrac{s}{\mathtt{a}s\mathtt{b}}$$

첫 번째 규칙은 문자열 b가 집합 S에 속함을 뜻한다. 두 번째 규칙은 집합 S가 문자열 s를 포함하면 문자열 asb도 그 집합에 속함을 뜻한다.

예제 4 영문자의 집합 $\Sigma = \{\mathsf{a}, \ldots, \mathsf{z}\}$를 이용하여 만들 수 있는 모든 문자열들의 집합을 다음과 같은 추론 규칙으로 정의할 수 있다.

$$\frac{}{\epsilon} \qquad \frac{s}{\mathsf{a}s} \qquad \frac{s}{\mathsf{b}s} \qquad \cdots \qquad \frac{s}{\mathsf{z}s}$$

추론 규칙의 오른쪽에 조건을 붙여서 첫 번째 규칙을 제외한 동일한 형태의 여러 규칙들을 다음과 같이 간략하게 표현할 수 있다.

$$\frac{}{\epsilon} \qquad \frac{s}{xs}\ x \in \{\mathsf{a}, \ldots, \mathsf{z}\}$$

문자열 s가 집합에 속하고 x가 임의의 영문자일 때, 문자열 xs도 그 집합에 속함을 의미한다. 동일한 집합을 문법으로 정의하면 다음과 같다.

$$\begin{array}{rcl} s & \rightarrow & \epsilon \\ & | & xs \quad (x \in \{\mathsf{a}, \ldots, \mathsf{z}\}) \end{array}$$

예제 5 정수를 원소로 가지는 리스트들의 집합은 다음과 같이 정의할 수 있다.

$$\frac{}{\mathsf{nil}} \qquad \frac{l}{n \cdot l}\ n \in \mathbb{Z}$$

첫 번째 규칙은 빈 리스트(nil)가 정수 리스트임을 뜻한다. 두 번째 규칙은 l이 정수 리스트일 때 그 앞에 정수 하나를 추가한 리스트 $(n \cdot l)$도 정수 리스트임을 뜻한다. 예를 들어, $-7 \cdot 3 \cdot 14 \cdot \mathsf{nil}$은 위의 규칙으로 만들어지는 리스트이다. 다음과 같이 공리에서 시작하여

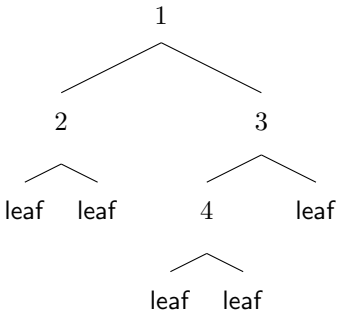
추론 규칙을 반복 적용하여 생성할 수 있기 때문이다.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{nil}}}{14 \cdot \mathsf{nil}}\ 14 \in \mathbb{Z}}{3 \cdot 14 \cdot \mathsf{nil}}\ 3 \in \mathbb{Z}}{-7 \cdot 3 \cdot 14 \cdot \mathsf{nil}}\ -7 \in \mathbb{Z}$$

위의 리스트를 문법으로 정의하면 다음과 같다.

$$\begin{array}{rcl} l & \rightarrow & \mathsf{nil} \\ & | & n \cdot l \quad (n \in \mathbb{Z}) \end{array}$$

예제 6 다음과 같이 생긴 이진 나무(binary tree)를 생각하자.



나무 구조를 매번 그림으로 표현하기는 번거로우므로 위의 이진 나무를 아래와 같이 표현하기로 하자.

$$(1, (2, \mathsf{leaf}, \mathsf{leaf}), (3, (4, \mathsf{leaf}, \mathsf{leaf}), \mathsf{leaf}))$$

앞으로 이와 같이 이차원 구조를 가지는 데이터를 괄호를 이용하여 일차원으로 표현하는 방법을 자주 쓸 것이다. 위와 같이 생긴 이진 나무들의 집합은 아래의 귀납 규칙으로 정의할 수 있다.

$$\frac{}{\mathsf{leaf}} \qquad \frac{t_1 \quad t_2}{(n, t_1, t_2)}\ n \in \mathbb{Z}$$

이진 나무를 만드는 두 가지 방법을 정의하고 있다.

1. 첫 번째 규칙은 하나의 말단 노드(leaf)만 포함하고 있는 빈 이진 나무를 뜻한다.
2. 두 번째 규칙은 이진 나무 t_1과 t_2가 있을 때, 정수 n을 루트 노드로 하고 t_1, t_2를 각각 왼쪽, 오른쪽 자식으로 하여 새로운 이진 나무 (n, t_1, t_2)를 만들 수 있음을 뜻한다.

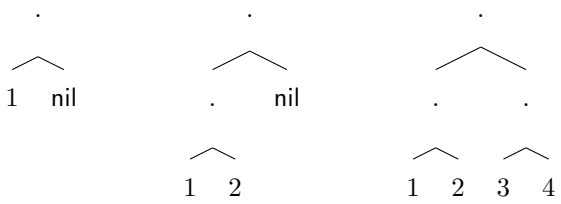
예를 들어, 위 규칙에 의하면

$$(1, (2, \mathsf{leaf}, \mathsf{leaf}), (3, (4, \mathsf{leaf}, \mathsf{leaf}), \mathsf{leaf}))$$

는 이진 나무이다. 다음과 같이 추론 규칙으로 증명할 수 있다.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{leaf}} \quad \dfrac{}{\mathsf{leaf}}}{(2, \mathsf{leaf}, \mathsf{leaf})}\ 2 \in \mathbb{Z} \quad \dfrac{\dfrac{\dfrac{}{\mathsf{leaf}} \quad \dfrac{}{\mathsf{leaf}}}{(4, \mathsf{leaf}, \mathsf{leaf})}\ 4 \in \mathbb{Z} \quad \dfrac{}{\mathsf{leaf}}}{(3, (4, \mathsf{leaf}, \mathsf{leaf}), \mathsf{leaf})}\ 3 \in \mathbb{Z}}{(1, (2, \mathsf{leaf}, \mathsf{leaf}), (3, (4, \mathsf{leaf}, \mathsf{leaf}), \mathsf{leaf}))}\ 1 \in \mathbb{Z}$$

예제 7 다른 스타일로 이진 나무를 정의해보자. 다음과 같이 자료를 내부 노드 대신 말단 노드에 저장하는 이진 나무를 생각해보자.



여기서 nil은 해당 가지에 자식 노드가 없음을 뜻한다. 위의 나무들을 1, $(1, \mathsf{nil})$, $((1,2), \mathsf{nil})$, $((1,2),(3,4))$와 같이 괄호를 이용하여 일차원으로 표현하기로 하면 이진 나무들의 집합을 다음과 같이 귀납적으로 정의할 수 있다.

$$\frac{}{\overline{n}\ n \in \mathbb{Z}} \qquad \frac{t}{(t, \mathsf{nil})} \qquad \frac{t}{(\mathsf{nil}, t)} \qquad \frac{t_1 \quad t_2}{(t_1, t_2)}$$

예를 들어 $((1,2),(3,\mathsf{nil}))$는 다음과 같이 만들어진다.

$$\frac{\dfrac{\overline{1}\ 1 \in \mathbb{Z} \quad \overline{2}\ 2 \in \mathbb{Z}}{(1,2)} \qquad \dfrac{\overline{3}\ 3 \in \mathbb{Z}}{(3, \mathsf{nil})}}{((1,2),(3,\mathsf{nil}))}$$

1.2 프로그래밍 언어의 귀납적 정의

프로그래밍 언어는 귀납적으로 만들어지는 프로그램들의 집합이다. 그 언어에 속한 프로그램의 생김새를 정의하는 규칙, 즉 프로

그램을 작성하는 규칙을 프로그래밍 언어의 문법구조(syntax)라고 하고 그 언어에 속한 프로그램의 의미를 정의하는 규칙을 의미구조(semantics)라고 한다. 두 가지 모두 귀납적으로 정의된다.

문법구조

정수와 사칙 연산자로 만들 수 있는 정수식들의 집합인 정수식 언어를 생각해보자. 예를 들어, 다음의 정수식들이 이 언어에 속하는 프로그램들이다.

```
1,   1+2,   1+(2*(3-4)),   (1+2)/(3*(4-5)),   ...
```

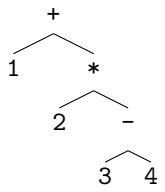
위와 같이 프로그램을 보통 일차원의 문자열로 작성하지만 사실 컴퓨터 프로그램은 나무 형태의 이차원 구조를 가진다. 예를 들어, 문자열

```
1+(2*(3-4))
```

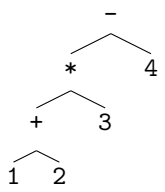
는 그림 1.1a의 프로그램을 의미하고,

```
((1+2)*3)-4
```

는 그림 1.1b의 프로그램을 뜻한다. 참고로 문자열로 작성된 일차원 프로그램을 이차원 나무 구조로 변환해주는 기술을 문법구조 분석(syntax analysis) 또는 파싱(parsing)이라고 한다. 이 책에서는 문법구조를 분석하는 방법은 다루지 않고, 프로그램의 구조가 나무



(a) 1+(2*(3-4))



(b) ((1+2)*3)-4

그림 1.1: 나무 구조로 표현한 프로그램

구조 형태로 명확하게 주어진다고 가정할 것이다.

정수식 언어의 문법구조는 다음과 같은 추론 규칙을 통해 귀납적으로 정의할 수 있다.

$$\overline{n}\ n \in \mathbb{Z} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ - } E_2} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ + } E_2} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ * } E_2} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ / } E_2}$$

프로그래밍 언어의 문법구조는 추론 규칙 대신 문법으로 정의하는 방식이 더 많이 쓰인다. 정수식 언어의 문법구조를 문법으로 표현하면 다음과 같다.

$$\begin{aligned} E \quad &\rightarrow \quad n \quad (n \in \mathbb{Z}) \\ &\mid \quad E_1 \texttt{ + } E_2 \\ &\mid \quad E_1 \texttt{ - } E_2 \\ &\mid \quad E_1 \texttt{ * } E_2 \\ &\mid \quad E_1 \texttt{ / } E_2 \end{aligned}$$

위의 규칙은 정수식을 만드는 방법에 다섯 가지가 있음을 뜻한다. 첫 번째 규칙에 의하여 임의의 정수(n)는 정수식이 될 수 있다. 나머지 규칙들은 두 정수식 E_1, E_2가 있을 때 이들을 사칙 연산자로 연결하여 새로운 정수식을 만들 수 있음을 뜻한다. 귀납 규칙의 특성상, 다섯 가지의 유한한 규칙으로 무한히 많은 임의의 정수식들을 만들어낼 수 있다.

예를 들어, 정수식 `1+(2*(3-4))`는 다음과 같이 만들어진다.

$$\dfrac{\dfrac{\dfrac{\overline{3}\quad\overline{4}}{2\quad 3\ -\ 4}}{1\quad 2\ *\ (3\ -\ 4)}}{1\ +\ (2\ *\ (3\ -\ 4))}$$

위 증명 나무의 위 아래를 뒤집고 나무 구조를 강조해서 표현하면 그림 1.1a가 된다. 이와 같이 귀납적으로 정의된 프로그래밍 언어의 문법구조는 이차원 구조를 가지는 프로그램을 생성하는 규칙이다. 이차원 구조를 일차원으로 혼동없이 쓰기 위해서 괄호를 적절히 써서 `1+(2*(3-4))`와 같이 표현하는 것 뿐이다.

의미구조

문법구조가 프로그래밍 언어에 속한 프로그램들의 생김새를 정의하는 규칙이라면, 의미구조는 프로그램의 실행 의미를 결정하는 규칙이다.

프로그램의 의미는 정의하기 나름이지만, 정수식의 경우 그 식을 계산했을 때 얻게되는 정수값으로 의미를 정의하는 것이 자연

스러울 것이다. "정수식 E를 계산하면 정수 n을 얻는다" 또는 "정수식 E의 의미는 n이다"를 다음과 같이 표현하기로 하자.

$$E \Rightarrow n$$

예를 들어, `1+(2*(3-4))` $\Rightarrow -1$은 프로그램 `1+(2*(3-4))`의 의미가 -1임을 뜻한다. 여기서 $\Rightarrow$는 프로그램과 정수에 관한 이항 관계(binary relation)를 나타낸다. 즉, 프로그램의 집합을 $\mathbb{P}$, 정수의 집합을 $\mathbb{Z}$라고 할 때 $\Rightarrow$는 곱집합 $\mathbb{P} \times \mathbb{Z}$의 부분집합이고 (i.e. $\Rightarrow \subseteq \mathbb{P} \times \mathbb{Z}$) 정수식의 의미구조를 정의한다는 것은 집합 $\Rightarrow$를 정의하는 것이다. 정수식의 경우 다음의 추론 규칙으로 집합 $\Rightarrow$를 정의할 수 있다.

$$\frac{}{n \Rightarrow n} \qquad \text{E-Num}$$

$$\frac{E_1 \Rightarrow n_1 \quad E_2 \Rightarrow n_2}{E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2} \qquad \text{E-Plus}$$

$$\frac{E_1 \Rightarrow n_1 \quad E_2 \Rightarrow n_2}{E_1 \texttt{ - } E_2 \Rightarrow n_1 - n_2} \qquad \text{E-Minus}$$

$$\frac{E_1 \Rightarrow n_1 \quad E_2 \Rightarrow n_2}{E_1 \texttt{ * } E_2 \Rightarrow n_1 * n_2} \qquad \text{E-Mult}$$

$$\frac{E_1 \Rightarrow n_1 \quad E_2 \Rightarrow n_2}{E_1 \texttt{ / } E_2 \Rightarrow n_1/n_2} \; n_2 \neq 0 \qquad \text{E-Div}$$

첫 번째 규칙(E-Num)은 임의의 정수 n에 대하여 (n, n)이 집합 $\Rightarrow$의 원소라는 뜻이다. 정수식 n의 의미를 정수값 n으로 정의한 것이다. 두 번째 규칙(E-Plus)은 (E_1, n_1)과 (E_2, n_2)가 $\Rightarrow$의 원소이면

$(E_1$ + $E_2, n_1+n_2)$도 $\Rightarrow$의 원소임을 뜻한다.[1] 나머지 규칙들의 의미도 비슷하게 정의된다. 단, 나눗셈의 경우 n_2가 0이 아닌 경우로 한정시켜서 의미를 정의하고 있다.

예를 들어, 위의 규칙으로 정의된 집합 $\Rightarrow$는 $(\texttt{(1+2)*(3/3)}, 3)$을 원소로 포함하고 있다. 정수식 `(1+2)*(3/3)`의 의미를 3으로 정의한 것이다. 다음과 같이 $(\texttt{(1+2)*(3/3)}, 3)$가 집합 $\Rightarrow$의 원소임을 증명할 수 있다.

$$\dfrac{\dfrac{\dfrac{}{\texttt{1} \Rightarrow 1}\ \text{E-Num} \quad \dfrac{}{\texttt{2} \Rightarrow 2}\ \text{E-Num}}{\texttt{1+2} \Rightarrow 3}\ \text{E-Plus} \quad \dfrac{\dfrac{}{\texttt{3} \Rightarrow 3}\ \text{E-Num} \quad \dfrac{}{\texttt{3} \Rightarrow 3}\ \text{E-Num}}{\texttt{3/3} \Rightarrow 1}\ \text{E-Div}}{\texttt{(1+2)*(3/3)} \Rightarrow 3}\ \text{E-Mult}$$

반면에, 어떤 정수식들은 올바르게 생겼어도 의미가 정의되지 않을 수 있다. 예를 들어, 정수식 `(3*4)/((1*2)-(1+1))`은 어떤 정수 n에 대해서도 `(3*4)/((1*2)-(1+1))` $\Rightarrow$ n을 위의 추론 규칙으로 증명할 수 없다. 전체식이 나눗셈인 경우이므로 만약 증명이 된다면 규칙 E-Div를 적용하는 방법밖에 없는데, 이 경우 분모를 계산하면 0이므로 E-Div 적용이 불가능하다. 이와 같이 프로그래밍 언어의 문법구조로 정의되는 모든 프로그램이 실행 의미를 가지는 것은 아니다. 컴파일러의 문법구조 분석 단계를 통과하더라도 제대로 실행되지 않는 프로그램이 존재할 수 있는 것이다.

---

1) 여기서 +는 프로그램에서 덧셈을 의미하기 위해 사용하는 기호이고 +는 수학에서의 덧셈을 의미하는 연산자이다. 따라서 n_1+n_2는 n_1과 n_2의 합에 해당하는 정수를 의미한다. 즉, +는 현재 정의하고 있는 대상 언어(object language)에 속하는 기호이고 +는 그 언어를 정의하는 데 사용하고 있는 메타 언어(metalanguage)에 속하는 기호이다. 혼동의 여지가 없는 경우, 앞으로 이 두 가지를 구분하지 않을 것이다.

지금까지 집합을 정의하는 추론 규칙의 관점에서 의미구조를 정의하였지만, 위의 규칙들을 프로그램 실행 규칙의 관점에서 직관적으로 해석해도 된다. 예를 들어 규칙 E-Plus는 식 E_1 + E_2를 계산하는 과정을 표현하고 있다. E_1 + E_2를 계산하는 과정을 먼저 식 E_1, E_2의 값을 재귀적으로 계산하여 n_1, n_2를 얻은 다음 이 두 정수를 더하는 과정으로 정의하였다. 이런식으로 의미구조를 해석하면 증명 나무를 만드는 과정이 프로그램을 실행하는 과정이 되고, 의미구조 정의를 재귀 함수로 구현하면 곧바로 프로그래밍 언어의 실행기(interpreter)를 얻을 수 있다.

1.3 귀납적 증명

귀납법은 귀납적으로 정의된 집합의 원소들이 가지는 성질을 증명하는 데 사용하는 증명 방법이기도 하다. 귀납법으로 정의된 집합 S가 있다고 하자. 집합 S에 속한 모든 원소 x에 대하여 어떤 성질 $P(x)$가 성립함을 보이기 위해서는 아래 두 가지를 보이면 된다.

1. x가 S의 기초 원소이면 $P(x)$를 직접 보인다.

2. x가 S에 속하는 다른 원소들 $y_1, y_2, \ldots, y_n$을 이용하여 귀납적으로 정의된다면

$$P(y_1), P(y_2), \ldots, P(y_n)$$

들이 모두 사실이라고 가정한 후, 이를 이용하여 $P(x)$가 사

실임을 보인다. 이 때 $P(y_1)$, $P(y_2)$, ..., $P(y_n)$을 귀납 가정(induction hypothesis)이라고 한다.

이러한 증명 방법을 구조적 귀납법(structural induction)이라고 부른다. 수학적 귀납법(mathematical induction)은 집합 S가 자연수 집합인 경우에 해당하는, 구조적 귀납법의 특수한 경우이다.

예제 1 집합 S를 아래 추론 규칙으로 정의되는 집합이라고 하자.

$$\frac{}{3} \qquad \frac{x \quad y}{x+y}$$

이렇게 정의된 집합 S에 속하는 모든 원소들이 3으로 나누어 떨어짐을 구조적 귀납법으로 증명해보자.

- 먼저 기초 원소들에 대해서 보인다. x가 3인 경우 명백히 3으로 나누어 떨어진다.
- 그 다음에는 귀납적으로 생성되는 원소들에 대해서 보인다. 원소들이 다음 규칙으로 생성되는 경우이다.

  $$\frac{x \quad y}{x+y}$$

  귀납 가정(induction hypothesis, I.H.)은 다음과 같다:

  x, y가 3으로 나누어 떨어진다.

귀납 가정에 의해 $x = 3k_1$, $y = 3k_2$라고 하자. 보일것은 다음과 같다:

$$x + y\text{가 } 3\text{으로 나누어 떨어진다.}$$

증명은 다음과 같다.

$$\begin{aligned} x + y &= 3k_1 + 3k_2 \quad \cdots \text{귀납가정} \\ &= 3(k_1 + k_2) \end{aligned}$$

예제 2 집합 S를 아래 추론 규칙으로 정의되는 집합이라고 하자.

$$\frac{}{()} \qquad \frac{x}{(x)} \qquad \frac{x \quad y}{xy}$$

그리고 x가 집합 S의 원소일때 $l(x)$와 $r(x)$가 각각 x 내에 포함된 왼쪽, 오른쪽 괄호의 개수를 의미한다고 하자. 다음과 같이 귀납적으로 정의된다.

$$\begin{aligned} l(()) &= 1 & \qquad r(()) &= 1 \\ l((x)) &= 1 + l(x) & \qquad r((x)) &= 1 + r(x) \\ l(xy) &= l(x) + l(y) & \qquad r(xy) &= r(x) + r(y) \end{aligned}$$

이제 S의 모든 원소 x에 대해서 왼쪽 괄호와 오른쪽 괄호의 개수가 같음을 증명해보자. 증명할 명제는 다음과 같이 쓸 수 있다.

$$\text{모든 } x \in S\text{에 대하여 } l(x) = r(x) \text{ 이다.}$$

다음과 같이 구조적 귀납법으로 증명할 수 있다.

- 기초 원소는 $x = ()$이다. 함수 l과 r의 정의에 의하여 $l(x) = 1 = r(x)$가 성립한다.
- 귀납적으로 만들어지는 경우는 아래 두 가지 경우가 있다.

$$\frac{x}{(x)} \qquad \frac{x \quad y}{xy}$$

  첫 번째 규칙의 경우 귀납 가정은 아래와 같고

$$l(x) = r(x)$$

  증명할 것은 $l((x)) = r((x))$이다. 증명은 다음과 같다.

$$\begin{aligned} l((x)) &= l(x) + 1 && \cdots l((x))\text{의 정의} \\ &= r(x) + 1 && \cdots \text{귀납 가정} \\ &= r((x)) && \cdots r((x))\text{의 정의} \end{aligned}$$

두 번째 규칙의 경우 귀납 가정은

$$l(x) = r(x), \qquad l(y) = r(y)$$

와 같고 증명할 것은 아래와 같다.

$$l(xy) = r(xy).$$

증명은 다음과 같다.

$$\begin{aligned} l(xy) &= l(x) + l(y) && \cdots l(xy)\text{의 정의} \\ &= r(x) + r(y) && \cdots \text{귀납 가정} \\ &= r(xy) && \cdots r(xy)\text{의 정의} \end{aligned}$$

2

함수형 프로그래밍

이 장에서는 OCaml[1]을 이용한 함수형 프로그래밍(functional programming)을 소개한다. 이 책에서 함수형 프로그래밍을 다루는 이유는 크게 세 가지이다. 먼저 앞으로 프로그래밍 언어를 디자인하고 구현하게 될텐데 구현 언어로 함수형 프로그래밍 언어인 OCaml을 사용할 것이다. 프로그래밍 언어의 실행기(interpreter)나 타입 시스템(type system)과 같이 프로그램을 데이터로 다루는 프로그램을 구현하는 데 있어서 있어서 함수형 언어들이 매우 편리하기 때문이다. 두 번째 이유는 OCaml을 비롯한 함수형 언어들은 프로그래밍 언어론 관점에서 잘 설계된 언어들이라는 점이다. 이러한 언어의 특징을 살펴보는 것만으로도 프로그래밍 언어에 대한 많은 공부가 된다. 특히 함수형 프로그래밍의 개념은 최근 들어 대부분의 프로그래밍 언어들이 차용하고 있으므로 다른 언어들을 깊게 이해하는 데에도 필수적이다. 마지막으로 앞으로 우리가 만들어 갈 언어가 OCaml과 유사하다. 다음 장부터 설계할 프로그래밍 언어의 특징에 대해 미리 감을 잡을 수 있다.

1) https://ocaml.org 에서 설치할 수 있다.

2.1 OCaml 기초

OCaml로 프로그램을 작성하는 데 있어서 필요한 가장 기본적인 개념들을 익혀보자.[2)]

실행하기

OCaml 프로그램을 실행시키는 방법에는 크게 세 가지가 있다. 텍스트 편집기를 이용하여 다음과 같이 프로그램을 작성하고 `hello.ml` 이름의 파일로 저장하자.

```
let _ = print_endline "Hello World"
```

표준 출력 함수 `print_endline`을 이용하여 문자열 `"Hello World"`를 출력하는 프로그램이다.

위 프로그램을 실행하기 위해서 먼저 REPL(Read-Eval-Print Loop, 대화형 프로그램 실행 도구)을 이용할 수 있다. 명령창에서 `ocaml`을 입력하면 다음과 같이 REPL이 실행된다(`$`과 `#`을 각각 쉘과 REPL의 프롬프트를 나타내는 문자라고 하자).

```
$ ocaml
        OCaml version 4.09.0

#
```

다음과 같이 입력하고 엔터를 누르면 문자열을 출력해준다.

2) 이 절의 일부 예제는 다음 책에서 가져왔다: Yaron Minsky, Anil Madhavapeddy, Jason Hickey. Real-World OCaml. O'Reilly

```
# print_endline "Hello World";;
Hello World
- : unit = ()
```

여기에서 ;;는 REPL에게 명령을 실행하라는 의미로 사용하는 문자로, 프로그램을 텍스트 파일에 작성할 때는 적을 필요가 없다. REPL에서 직접 프로그램을 입력하지 않고 프로그램이 저장되어 있는 파일을 통째로 읽어올 수도 있다.

```
# #use "hello.ml";;
Hello World
- : unit = ()
```

REPL대신 OCaml 실행기를 직접 이용해도 된다. 다음과 같이 명령어 ocaml뒤에 실행할 파일 이름을 주면 된다.

```
$ ocaml hello.ml
Hello World
```

리눅스 환경에서 컴파일러(ocamlc)를 이용하여 실행 파일을 만들고 실행하는 방법은 다음과 같다:

```
$ ocamlc helloworld.ml
$ ls
a.out   hello.cmi  hello.cmo   hello.ml
$ ./a.out
Hello World
```

컴파일을 하면 실행파일인 a.out이 생성되며, 이를 실행하면 문자열 "Hello World"가 출력된다.

프로그램 구성의 기본단위

C, Java, Python과 같은 명령형 언어(imperative language)와 비교하여 OCaml과 같은 함수형 프로그래밍 언어가 가지는 특징 중 하나는 프로그램을 구성하는 기본 단위가 명령문(statement)이 아닌 식(expression)이라는 점이다. 일반적으로 프로그래밍 언어에서 명령문은 프로그램의 상태를 변화시키는 구문을 의미한다. 예를 들어, C나 Java의 대입문 `x = x + 1`은 메모리에 저장된 변수 `x`의 값을 1 증가시키는 명령문이다. 반면에 식은 프로그램 상태 변경없이 계산의 결과로 어떤 값을 만들어내는 구문을 뜻한다. 예를 들어, 식 `x + y`는 `x`와 `y`의 값을 참조하여 새로운 값을 계산할 뿐 메모리 상태를 변경하지는 않는다. OCaml, Haskell, Lisp과 같은 함수형 언어에서는 메모리 상태 변경 없이 식을 중심으로 값을 계산하는 형태로 프로그램을 작성하는 것이 일반적이다.

명령문 대신 식을 중심으로 프로그램을 작성하다 보니 함수형 프로그래밍에서는 문제를 푸는 절차 대신 풀고자 하는 문제를 기술하는 데 중점을 두어서 프로그램을 작성하는 것이 자연스럽다. 이처럼 프로그래밍 언어의 선택은 문제에 접근하는 사고 방식에 큰 영향을 주는데, 뒤에서 예제와 함께 좀 더 자세히 알아볼 것이다.

기본값

OCaml이 제공하는 가장 기본적인 식에는 정수, 실수, 참/거짓, 문자, 문자열 등의 기본값(primitive value)을 만들어내는 식들이 있다. 예를 들어, REPL에서 산술식(arithmetic expression) `1 + 2 *`

3을 입력해보자.

```
# 1 + 2 * 3;;
- : int = 7
```

OCaml 실행기는 입력으로 주어진 식을 계산하여 결과값이 7이라고 알려줌과 동시에 결과값의 타입(type)도 함께 알려준다. 이 경우 `int`는 결과값이 정수 타입임을 뜻한다. 실수식은 다음과 같이 계산할 수 있다.

```
# 1.1 +. 2.2 *. 3.3;;
- : float = 8.36
```

위 식의 결과값은 8.36이고 실수(`float`) 타입임을 뜻한다. OCaml은 값들을 타입에 따라 명확하게 분류하는 언어이다. 심지어 실수에 대한 덧셈과 정수에 대한 덧셈 연산자가 다르다. 위의 예에서처럼 실수에 대한 연산을 할 때에는 연산자에 `.`을 붙여야 한다. 값의 타입을 명확히 분류하지 않으면 다음과 같이 타입 오류(type error)가 발생한다.

```
# 3 + 2.0;;
Error: This expression has type float but an
expression was expected of type int
```

연산자 `+`는 피연산자로 정수값을 기대하는 데 오른쪽 피연산자의 값이 실수이므로 계산할 수 없다는 뜻이다. 정수와 실수를 더하려면 다음과 같이 명시적으로 타입을 변환해주어야 한다.

```
# 3 + (int_of_float 2.0);;
- : int = 5
```

```
# (float_of_int 3) +. 2.0;;
- : float = 5.
```

int_of_float와 float_of_int는 각각 실수를 정수로, 그리고 정수를 실수로 변환해주는 함수이다.

부울식(boolean expression)은 결과값이 참 또는 거짓인 식이다. 참과 거짓은 각각 true, false로 나타내고 이들 값의 타입은 bool이다.

```
# true;;
- : bool = true
# false;;
- : bool = false
```

산술식에 대한 비교 연산도 부울값을 만들어낸다.

```
# 1 = 2;;  (* equal to *)
- : bool = false
# 1 <> 2;;  (* not equal to *)
- : bool = true
# 2 <= (1+1);;  (* less than or equal to *)
- : bool = true
```

논리 연산자를 이용하여 기존 부울식을 엮어서 새로운 부울식을 만들 수 있다.

```
# (2 > 1) && (3 > 2 || 5 < 2);;
- : bool = true
# not (2 > 1);;
- : bool = false
```

이 외에도 OCaml에서는 문자(character), 문자열(string), 유닛(unit)값을 기본값으로 제공한다.

```
# 'c';;
- : char = 'c'
# "Objective " ^ "Caml";;
- : string = "Objective Caml"
# ();;
- : unit = ()
```

unit은 값을 하나만 가지는 타입이며 그 값은 ()로 나타낸다. C 언어에서 void 타입과 비슷한 역할을 한다고 생각하면 된다. 위의 예에서 ^는 두 문자열을 이어붙이는 연산자이다.

조건식

프로그래밍 언어가 제공하는 가장 기본적인 구문 중 하나는 조건문이다. C나 Java와 달리 OCaml에서는 조건문도 식이며, 따라서 그 결과로 값을 계산한다. 조건식(conditional expression)은 다음과 같이 생겼다.

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$$

여기서 e_1, e_2, e_3는 임의의 식을 뜻한다. 조건식의 의미는 e_1의 값이 참이면 e_2를 계산하고 e_2의 값이 거짓이면 e_3를 계산하라는 뜻이다. 예를 들어, 아래 프로그램은 e_1, e_2, e_3가 각각 2 > 1, 0, 1인 경우이다.

```
# if 2 > 1 then 0 else 1;;
- : int = 0
```

식 `2 > 1`의 값이 `true`이므로 e_2에 해당하는 식 `0`을 계산하고 그 값이 전체 조건식의 값이 된다. 조건을 다음과 같이 바꾸면 전체 조건식의 값은 e_3의 값이 된다.

```
# if 2 < 1 then 0 else 1;;
- : int = 1
```

OCaml에서 조건식을 작성할 때에는 몇가지 규칙을 지켜야 한다. 먼저 e_1은 반드시 부울식이어야 한다. 예를 들어,

```
if 1 then 2 else 3
```

는 컴파일 단계에서 타입 오류가 발생한다.

```
# if 1 then 2 else 3;;
Error: This expression has type int but an expression
was expected of type bool
```

또한 e_2와 e_3는 같은 타입의 값을 계산하는 식들이어야 한다. 예를 들어, 아래 식도 타입 오류가 발생하고 실행되지 않는다.

```
# if true then 1 else true;;
Error: This expression has type bool but an expression
was expected of type int
```

두 번째 규칙은 사실 OCaml의 정적 타입 시스템(static type system)이 요구하는 것이다. e_2와 e_3의 타입이 달라도 프로그램을 실행시키는 데는 문제가 없지만 타입을 실행 전에 결정하려다 보니

필요한 제약 사항이다. 이처럼 정적 타입 시스템을 갖춘 언어는 프로그램을 작성하는 데 있어서 요구하는 규칙이 동적 언어들에 비해 일반적으로 더 많다. 8장에서 타입 시스템을 공부하면 이유를 이해할 수 있게 될 것이다.

변수

OCaml에서 변수는 값에 붙인 이름이다. 이름을 지으려면 키워드 `let`을 이용한다. 예를 들어,

```
# let x = 3 + 4;;
val x : int = 7
# let y = x + x;;
val y : int = 14
```

`3 + 4`의 값을 `x`로, `x + x`의 값을 `y`로 이름지었다. 이렇게 정의한 이름들은 앞으로 어디에서나 접근 가능한 일종의 전역 변수가 된다. 제한된 범위에서만 이름이 유효한 지역 변수를 정의하려면 `let...in` 식을 이용한다.

$$\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

변수 x가 e_1의 값을 가지도록 정의한 후에 식 e_2를 계산하라는 뜻이다. 이 때, 변수 x의 유효범위(scope)는 e_2이다. 즉, 식 e_2안에서만 방금 정의한 변수 x가 의미를 가진다. 그리고 `let...in`도 식이므로 값을 계산해낸다. 그 값은 식 e_2의 값이다. 예를 들어, 아래 프로그램을 보자.

```
# let a = 1 in a * 2;;
- : int = 2
```

변수 a를 1로 정의한 후에 a * 2를 계산하였고 그 값이 전체식의 값이 되었다. 이 때 변수 a는 e_2에 해당하는 a * 2에서만 유효하다. 따라서 위의 let식을 계산한 후 a에 접근하면 오류가 발생한다.

```
# a;;
Error: Unbound value a
```

정의되어 있지 않은 변수 a에 접근했다는 뜻이다.

let x = e_1 in e_2의 e_1과 e_2에는 임의의 식이 올 수 있다. 다음과 같이 let을 중첩하여 사용해도 된다.

```
#  let d =
    let a = 1 in
    let b = a + a in
    let c = b + b in
      c + c;;
val d : int = 8
# d;;
- : int = 8
# a;;
Error: Unbound value a
# b;;
Error: Unbound value b
# c;;
Error: Unbound value c
```

전역 변수 d의 값을 중첩된 let을 이용하여 정의하였다. 위에서 변수 a, b, c는 변수 d의 값을 정의하기 위해서 임시로 사용한 지역 변수이므로 d가 정의된 후에는 접근할 수 없게 된다.

아래 예에서는 let의 e_1 자리에 let을 중첩하여 사용하였다.

```
# let x = (let y = 2 in y * 2) in x;;
- : int = 4
# y;;
Error: Unbound value y
```

이 경우에도 변수 y는 x를 정의하기 위하여 쓰인 지역 변수이므로 전체 식을 계산한 후에는 접근할 수 없다.

함수

함수를 정의할 때에도 let을 사용한다. 다음은 함수 square를 정의한 것이다.

```
# let square x = x * x;;
 val square : int -> int = <fun>
```

함수의 이름 다음에 함수의 인자(argument)의 이름을 적어주고, = 다음에 함수의 몸통(body)을 정의한다. 위의 함수 square는 인자 x를 받아서 결과값으로 x * x를 반환하는 함수를 정의한 것이다. 함수를 정의하면 OCaml 실행기는 그 타입을 알려주는데, 이 경우 int -> int는 정수를 인자로 받아서 정수를 되돌리는 함수라는 뜻이다. 위에서 <fun>은 정의한 값이 함수라는 뜻이다. 함수의 경우 OCaml 실행기는 구체적인 값을 보여주지는 않는다.

함수를 정의한 후에는 다음과 같이 호출하여 사용할 수 있다.

```
# square 2;;
- : int = 4
# square (2 + 5);;
```

```
- : int = 49
# square (square 2);;
- : int = 16
```

함수 square를 아래와 같이 정의할 수도 있다.

```
# let square = fun x -> x * x;;
val square : int -> int = <fun>
```

여기서 fun x -> x * x는 인자 x를 받아서 x * x를 반환하는 함수를 나타내고, 이름 square가 그 함수값을 지칭하도록 하였다. 이렇게 함수를 정의하는 방식은 앞에서 정의한 방식과 정확히 동일하다. 즉, let square x = x * x와 같은 식을 OCaml 컴파일러가 내부적으로는 let square = fun x -> x * x와 같이 처리한다고 생각하면 된다. fun x -> x * x와 같이 정의한 함수를 이름없는 함수(anonymous function)라고 부른다.

다른 예로, 인자 x가 음수인지 여부를 확인하는 함수 neg를 정의해보자. 함수의 몸통이 조건식인 경우이다.

```
# let neg x = if x < 0 then true else false;;
val neg : int -> bool = <fun>
# neg 1;;
- : bool = false
# neg (-1);;
- : bool = true
```

OCaml에서 함수 호출식은 일반적으로 e_1 e_2와 같이 생겼고, e_1과 e_2는 임의의 식이 될 수 있다. 예를 들어, 위에서 sqaure를 호출하는 세 가지 예에서 e_1은 모두 square이고 e_2는 각각 2, 2 + 5,

square 2인 경우이다. 다음과 같이 e_1에 이름없는 함수가 직접 와도 된다.

```
# (fun x -> x * x) 3;;
- : int = 9
# (fun x -> if x > 0 then x + 1 else x * x) 1;;
- : int = 2
```

함수는 여러 인자를 가질 수 있다. 예를 들어, 두 정수 x와 y를 인자로 받아서 각각의 제곱의 합을 반환하는 함수를 작성해보자.

```
# let sum_of_squares x y = (square x) + (square y);;
val sum_of_squares : int -> int -> int = <fun>
# sum_of_squares 3 4;;
- : int = 25
```

OCaml은 위 함수의 타입이 int -> int -> int라고 알려주는데 정수 두 개를 받아서 정수를 반환하는 함수 타입을 뜻한다. 또는 위 타입을 int -> (int -> int)로 해석하여 정수 인자 하나를 받아서 정수에서 정수로 가는 함수를 반환하는 함수라고 해석해도 된다. 실제로 위에서 정의한 함수를 이와 같이 해석하여 사용할 수 있다.

```
# let f = sum_of_squares 3;;
val f : int -> int = <fun>
# f 4;;
- : int = 25
```

변수 f를 sum_of_squares 3의 값으로 정의하였는데, 그 값은 정수 y를 받아서 (square 3) + (square y)의 값을 반환하는 함수가 된다. 따라서 f를 4로 호출할 수 있고 그 값은 25가 된다.

재귀 함수를 정의하는 방법도 동일하지만 `let` 다음에 키워드 `rec`을 붙여주어야 한다. 예를 들어, 함수 `factorial`을 다음과 같이 정의할 수 있다.

```
# let rec factorial a =
    if a = 1 then 1 else a * factorial (a - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

OCaml은 함수를 일급(first-class)으로 취급하는 언어이다. 함수를 정의하고 사용하는 방식이 매우 자유롭다는 뜻이다. 프로그래밍 언어에서 어떤 값이 일급이 되려면 그 값을 변수에 저장할 수 있고, 함수의 인자로 넘길 수 있고, 함수의 결과값으로 반환할 수 있어야 한다. 예를 들어, C 언어에서는 정수, 문자 등의 값들이 일급으로 취급된다. OCaml에서는 함수도 일급으로 취급되어 다음과 같이 함수가 다른 함수를 인자로 받을 수 있다.

```
# let sum f a b =
    (if f a then a else 0) + (if f b then b else 0)
val sum : (int -> bool) -> int -> int -> int = <fun>
# let even x = x mod 2 = 0;;
val even : int -> bool = <fun>
# sum even 3 4;;
- : int = 4
# sum even 2 4;;
- : int = 6
```

위에서 `even`은 인자로 받은 정수 `x`가 짝수이면 참을 반환하는 함수이다. 즉, `even`의 몸통식 `x mod 2 = 0`은 `x`를 2로 나눈 나머지가

0이면 참을 계산하는 부울식이다. 함수 sum은 인자 f, a, b를 받는데, f가 정수에서 참/거짓으로 가는 함수이다. 이 함수를 a, b에 적용하여 참이 계산될 때에만 더해서 반환하는 함수이다. 예를 들어, sum even 3 4는 함수 sum의 첫 번째 인자로 함수 even을 넘겨주었으므로 그 값은 4가 된다.

함수가 다른 함수를 반환할 수도 있다.

```
# let plus a = fun b -> a + b;;
val plus : int -> int -> int = <fun>
# let plus1 = plus 1;;
val plus1 : int -> int = <fun>
# plus1 1;;
- : int = 2
# plus1 2;;
- : int = 3
# let plus2 = plus 2;;
val plus2 : int -> int = <fun>
# plus2 1;;
- : int = 3
```

함수 plus는 인자 a를 받아서 함수 fun b -> a + b를 반환하는 함수이다. 타입 int -> int -> int은 int -> (int -> int)와 같이 해석할 수 있고, 정수 하나를 받아서 정수에서 정수로 가는 함수를 반환하는 함수 타입임을 뜻한다. 변수 plus1을 plus 1로 정의하였으므로 plus1은 인자 b를 받아서 1 + b를 반환하는 함수를 의미한다. 비슷하게 plus2는 정수 인자를 받아서 2를 더하는 함수가 된다.

sum과 plus와 같이 다른 함수를 인자로 받거나 반환하는 함수를 고차 함수(higher-order function)라고 부른다. 프로그래밍 언어

가 고차 함수를 지원하면 더 상위에서 생각하고 프로그래밍 할 수 있게 된다. 즉, 프로그래밍 패턴을 추상화하여 이름 짓는 것이 가능해져서 간결하고 읽기 쉬운 프로그램을 작성하는 데 유리하다. 고차 함수에 대해서는 2.3절에서 자세히 알아보기로 하자.

정적 타입 시스템

OCaml은 정적 타입 시스템을 갖추고 있는 언어이다. 일반적으로 프로그래밍 언어는 정적 타입 시스템을 갖춘 언어(statically typed language)와 동적 타입 시스템을 갖춘 언어(dynamically typed langauge)로 분류할 수 있는데 OCaml은 C, C++, Java, Scala 등과 함께 전자에 속한다. 이들 언어들은 타입 체킹을 정적으로, 컴파일 단계에서 수행하기 때문에 다음과 같이 타입 오류가 있는 프로그램은 컴파일러를 통과하지 못한다.

```
# 1 + true;;
Error: This expression has type bool but an
expression was expected of type int
```

반면에 Python, Lisp과 같이 동적 타입 시스템을 갖춘 언어들은 타입 체킹을 프로그램 실행 중에 수행한다. 프로그램에 타입 오류가 있는지를 미리 검사하지 않고, 실행 중에 타입이 맞지 않는 계산이 일어나는지 모니터링하는 것이다.

이들 두 부류는 상반된 장단점을 가지고 있다. 예를 들어 정적 타입 시스템을 갖춘 언어들은 타입 오류를 프로그램 개발 중에 검출할 수 있으므로 높은 안정성을 필요로 하는 프로그램 개발에 적합하다. 반면에 동적 타입 시스템을 갖춘 언어들은 정적 언어들보

다 표현의 자유도가 높고 유연하여 프로그램 개발이 빠르다는 장점이 있다.

정적 타입 시스템을 갖춘 언어들은 다시 두 가지로 구분할 수 있다. 첫 번째는 안전한(sound) 타입 시스템을 장착한 언어들이다. 이들 언어들은 프로그램 실행 중에 발생가능한 모든 타입 오류를 컴파일 단계에서 빠짐없이 찾아준다. OCaml, Haskell, Scala 등의 함수형 언어들이 주로 그렇다. 두 번째는 정적 타입 시스템을 갖추었지만 실행 중에 여전히 타입 오류가 발생할 수 있는 안전하지 않은 언어이다. C, C++ 가 대표적이다.

OCaml은 프로그램의 타입을 정적으로 체크함과 동시에 타입을 자동으로 유추해준다. 예를 들어, C나 Java에서는 다음과 같이 변수와 함수의 타입을 항상 적어주어야 했다.

```
public static int f(int n) {
  int a = 2;
  return a * n;
}
```

OCaml에서는 위 함수를 다음과 같이 타입 정보없이 정의할 수 있고, 컴파일러가 대신 타입을 자동으로 유추해준다.

```
# let f n =
    let a = 2 in
      a * n;;
val f : int -> int = <fun>
```

OCaml은 프로그램이 아무리 복잡해도 자동으로 타입을 추론할 수 있다. OCaml 실행기가 타입을 어떻게 자동으로 추론하는지 살펴보기 위해서 앞에서 정의한 함수 `sum`을 예로 들어보자.

```
# let sum f a b =
    (if f a then a else 0) + (if f b then b else 0)
val sum : (int -> bool) -> int -> int -> int = <fun>
```

OCaml은 다음의 과정을 통해 함수의 몸통식을 분석하여 타입 정보를 자동으로 유추해낸다. 먼저 조건식 `if` e_1 `then` e_2 `else` e_3에서 e_2와 e_3의 타입은 같아야 하고, `0`의 타입은 정수이므로 인자 `a`와 `b`의 타입이 `int`임을 유추할 수 있다. 그 다음에 `f`는 함수 호출의 형태(`f a` 또는 `f b`)로 사용되고 있으므로 함수 타입을 가져야 한다는 것과 `f`의 인자로 `a` 또는 `b`가 주어졌다는 점, 그리고 `f a`, `f b`의 결과값이 조건식의 e_1으로 사용되고 있다는 점에서 `f`가 함수 타입 `int -> bool`이어야 함을 추론해낸다. 마지막으로 `sum`은 덧셈의 결과를 반환하므로 그 타입이 `int`임을 추론한다. 이와 같은 과정을 거쳐서 OCaml의 자동 타입 추론(automatic type inference) 알고리즘은 프로그램 코드로부터 그 타입을 실행 전에 자동으로 추론해낸다. 자동 타입 추론의 동작 원리는 8장에서 자세히 살펴볼 것이다.

타입이 자동으로 유추되더라도 타입을 직접 적어주고 싶을 수 있다. 다음과 같이 하면 된다.

```
# let sum (f : int -> bool) (a : int) (b : int) : int
  = (if f a then a else 0) + (if f b then b else 0);;
val sum : (int -> bool) -> int -> int -> int = <fun>
```

이 경우 OCaml은 사용자가 적은 타입이 올바른지를 검증해준다. 예를 들어 다음과 같이 사용자가 적어준 타입이 잘못되었을 경우, 그 오류를 자동으로 찾아준다.

```
# let sum (f : int -> int) (a : int) (b : int) : int =
    (if f a then a else 0) + (if f b then b else 0);;
Error: The expression (f a) has type int but an
expression was expected of type bool
```

사용자가 실수로 f의 타입을 int -> int로 적었는데, f의 결과 타입이 bool이어야 하므로 잘못되었다는 뜻이다. OCaml의 타입 시스템은 안전(sound)하므로 사람이 적은 타입에 오류가 있다면 이를 반드시 찾아준다.

경우에 따라 어떤 식의 타입이 하나로 결정되지 않을 수도 있다. 예를 들어, 아래에 정의한 함수 id는 임의의 타입에 대해서 사용할 수 있는 함수이다.

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# id 1;;
- : int = 1
# id "abc";;
- : string = "abc"
# id true;;
- : bool = true
```

위와 같이 함수가 임의의 타입에 대해서 동작할 경우 OCaml은 'a와 같은 타입 변수를 이용하여 타입을 표현한다. 이러한 타입을 다형 타입(polymorphic type)이라 하고, 다형 타입을 가지는 함수를 다형 함수(polymorphic function)라고 부른다. 임의의 타입에 대해서 사용할 수 있는 함수라는 뜻이다.

OCaml의 다형 타입 시스템은 완전히 자유롭지는 않고 let으로 정의된 다형 함수만 지원한다. 이러한 타입 시스템을 let-다형

타입 시스템(let-polymorphic type system)이라고 부른다. 예를 들어, 아래와 같이 함수 f를 정의하면 다형 함수로 인식되어 문제 없이 실행된다.

```
# let f = fun x -> x in
    let x = f 1 in
      let y = f true in
        3;;
- : int = 3
```

하지만 동일한 의미를 가지는 프로그램을 아래와 같이 let식 없이 작성하면 타입 오류가 발생한다.

```
# (fun f ->
    let x = f 1 in
      let y = f true in
        3) (fun x -> x);;
Error: The expression has type bool but an expression
was expected of type int
```

패턴 매칭

패턴 매칭을 이용하면 중첩된 조건을 간결하게 표현할 수 있다. 예를 들어, 아래 프로그램을 보자.

```
let is123 s = if s = "1" then true
              else if s = "2" then true
              else if s = "3" then true
              else false
```

위의 함수를 패턴 매칭을 이용하여 아래와 같이 작성할 수 있다.

```
let is123 s =
  match s with
  "1" -> true
 |"2" -> true
 |"3" -> true
 | _  -> false
```

또는 다음과 같이 더 간단하게 정의할 수 있다.

```
 let is123 s =
  match s with
  "1" | "2" | "3" -> true
  | _ -> false
```

s의 값이 "a", "b", "c"중 하나이면 true, 그 외의 경우에는 false 라는 뜻이다. 패턴 매칭은 아래에서 튜플/리스트와 같은 자료형이나 사용자 정의 타입의 값을 사용하는 경우 유용하게 사용된다.

패턴 매칭을 중첩하여 사용할 경우에는 다음과 같이 괄호로 중첩된 match...with 문을 감싸주는 것이 좋다

```
match x with
| 1 -> true
| 2 ->
  (match y with
    | "a" -> true
    | "b" -> false)
| 3 -> false
```

또는 괄호 대신 begin ... end 문으로 감싸주어도 된다.

```
match x with
| 1 -> true
| 2 ->
  begin
```

```
    match y with
    | "a" -> true
    | "b" -> false
  end
| 3 -> false
```

위와 같이 하지 않으면 마지막 줄의 패턴이 `match y with`의 경우로 인식되어 타입 오류가 발생하게 된다.

튜플과 리스트

튜플(tuple)과 리스트(list)는 함수형 프로그래밍 언어에서 가장 많이 사용되는 자료 구조이다.

튜플은 값들의 묶음이다. 예를 들어, 튜플 `(1, "one")`은 정수와 문자열의 묶음이고 그 타입은 `int * string`과 같이 표현한다. 튜플 `(2, "two", true)`는 정수, 문자열, 부울값의 묶음이고 타입은 `int * string * bool`이다.

```
# let x = (1, "one");;
val x : int * string = (1, "one")
# let y = (2, "two", true);;
val y : int * string * bool = (2, "two", true)
```

튜플의 각 원소에 접근하려면 패턴 매칭을 이용하면 된다. 예를 들어, 두 원소로 구성된 튜플의 첫 번째와 두 번째 원소를 가져오는 함수 `fst`, `snd`를 다음과 같이 정의할 수 있다.

```
# let fst p = match p with (x,_) -> x;;
val fst : 'a * 'b -> 'a = <fun>
# let snd p = match p with (_,x) -> x;;
val snd : 'a * 'b -> 'b = <fun>
```

또는 다음과 같이 함수의 인자에 튜플 패턴을 직접 사용할 수도 있다.

```
# let fst (x,_) = x;;
val fst : 'a * 'b -> 'a = <fun>
# let snd (_,x) = x;;
val snd : 'a * 'b -> 'b = <fun>
```

타입 `'a * 'b -> 'a`는 함수 `fst`가 임의의 타입 `'a`와 `'b`로 구성된 튜플을 입력으로 받아서 `'a` 타입의 값을 반환하는 다형 함수임을 의미한다. 이와 같이 함수의 타입을 보면 함수가 하는 일을 어느정도 유추할 수 있다.

함수 인자뿐 아니라 `let`에서도 튜플 패턴을 사용할 수 있다. 예를 들어, 아래 코드를 보자.

```
# let p = (1, true);;
val p : int * bool = (1, true)
# let (x,y) = p;;
val x : int = 1
val y : bool = true
```

`p`는 튜플 `(1, true)`를 뜻하고 이를 `x`와 `y`로 분해하였다.

리스트는 같은 타입을 가지는 원소들의 나열이다. 예를 들어, 숫자 1, 2, 3으로 구성된 리스트는 `[1; 2; 3]`와 같이 표현하고 그 타입은 `int list`이다.

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
```

OCaml에서 리스트의 각 원소는 세미콜론(;)으로 구분한다. 각 원소를 콤마(,)로 구분한 리스트 [1, 2, 3]은 튜플 (1,2,3)을 원소로 가지는 리스트인 [(1,2,3)]으로 인식되므로 주의가 필요하다.

```
# [1,2,3];;
- : (int * int * int) list = [(1, 2, 3)]
# [(1,2,3)];;
- : (int * int * int) list = [(1, 2, 3)]
```

빈 리스트는 []로 나타내고 그 타입은 'a list로 다형 타입을 가진다.

```
# [];;
- : 'a list = []
```

OCaml에서 리스트의 모든 원소는 같은 타입이어야 한다. 예를 들어, [1; true]는 OCaml에서 리스트가 아니다.

```
# [1; true];;
Error: This expression has type bool but an expression
was expected of type int
```

이 제약도 정적 타입 시스템에 기인한다. 예를 들어, Python과 같이 동적 타입 시스템을 갖춘 언어에서는 리스트에 서로 다른 타입의 값이 함께 사용될 수 있다.

또한 리스트는 순서가 있는 원소들의 나열이다. 따라서 아래 두 리스트는 서로 다른 리스트이다.

```
# [1;2;3] = [2;3;1];;
- : bool = false
```

리스트의 첫 번째 원소를 머리(head), 첫 번째 원소를 제외한 나머지 리스트를 꼬리(tail)라고 부른다. 예를 들어, 리스트 `[1; 2; 3]`의 머리는 `1`, 꼬리는 `[2; 3]`이다. 일반적으로 어떤 리스트의 타입이 t `list`일 때, 머리의 타입은 t이고 꼬리의 타입은 t `list`이다.

리스트의 원소는 임의의 타입의 값이 될 수 있다. 물론 각 원소의 타입은 같아야 한다.

```
# [1;2;3;4;5];;
- : int list = [1; 2; 3; 4; 5]
# ["OCaml"; "Java"; "C"];;
- : string list = ["OCaml"; "Java"; "C"]
# [(1,"one"); (2,"two"); (3,"three")];;
- : (int * string) list =
    [(1, "one"); (2, "two"); (3, "three")]
# [[1;2;3];[2;3;4];[4;5;6]];;
- : int list list = [[1; 2; 3]; [2; 3; 4]; [4; 5; 6]]
```

마지막 예는 정수 리스트들의 리스트(`int list list`)이다. 이 때, 각 원소에 해당하는 리스트들의 길이는 모두 달라도 된다.

```
# [[1;2;3]; [4]; []];;
- : int list list = [[1; 2; 3]; [4]; []]
```

리스트 `[1;2;3]`와 `[4]`는 길이가 달라도 모두 정수 리스트이고, 빈 리스트 `[]`는 다형 타입이므로 역시 정수 리스트가 될 수 있기 때문이다.

OCaml이 제공하는 기본 리스트 연산자에는 두 가지가 있다. 먼저 `::`(cons라고 읽는다)는 리스트의 맨 앞에 원소를 하나 추가한 리스트를 만들어준다.

```
# 1::[2;3];;
- : int list = [1; 2; 3]
# 1::2::3::[];;
- : int list = [1; 2; 3]
```

두 리스트를 이어붙일 때에는 @(append라고 읽는다)을 사용한다.

```
# [1; 2] @ [3; 4; 5];;
- : int list = [1; 2; 3; 4; 5]
```

리스트를 다루는 함수를 작성할 때 패턴 매칭이 자주 쓰인다. 예를 들어, 리스트의 머리와 꼬리를 구하는 함수 hd와 tl을 정의해보자.

```
# let hd l =
    match l with
    | [] -> raise (Failure "hd is undefined")
    | a::b -> a;;
val hd : 'a list -> 'a = <fun>
# let tl l =
    match l with
    | [] -> raise (Failure "tl is undefined")
    | a::b -> b;;
val tl : 'a list -> 'a list = <fun>
# hd [1;2;3];;
- : int = 1
# tl [1;2;3];;
- : int list = [2; 3]
```

리스트의 머리와 꼬리는 빈 리스트에 대해서는 정의되지 않는다. 리스트 l이 빈 리스트가 아닌 경우이면 머리 a와 꼬리 b로 분해할 수 있고 hd는 a를, tl은 b를 반환한다(리스트 l이 원소가 하나인

리스트인 경우 꼬리 tl은 빈 리스트이다). 위의 정의에서 두 함수의 반환 타입이 다름을 눈여겨 보자. 예외 처리를 생략하면 다음과 같이 간단하게 정의할 수도 있다.

```
# let hd (a::b) = a;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
val hd : 'a list -> 'a = <fun>
# let tl (a::b) = b;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
val tl : 'a list -> 'a list = <fun>
```

이 경우 OCaml은 함수 hd와 tl가 빈 리스트를 고려하지 않고 있음을 알려주고 있다. 이러한 경고 메시지는 컴파일 오류는 아니지만 실행 중에 처리되지 않은 예외를 발생시킬 수 있으므로 미리 모두 제거하는 것이 좋다.

다른 예로, 리스트의 길이를 구하는 함수를 작성해보자.

```
# let rec length l =
    match l with
     [] -> 0
    |h::t -> 1 + length t;;
val length : 'a list -> int = <fun>
# length [1;2;3];;
- : int = 3
```

인자로 주어진 리스트 l이 빈 리스트이면 길이를 0으로 정의하였다. 비어 있지 않다면 l을 머리 h와 꼬리 t로 나눌 수 있고 이 경우

l의 길이는 꼬리 t의 길이에 1을 더한 것과 같아야 한다. 위의 정의에서 h는 사용하고 있지 않으므로 다음과 같이 밑줄(_)로 생략할 수 있다.

```
let rec length l =
  match l with
   [] -> 0
  |_::t -> 1 + length t
```

사용자 정의 타입

새로운 타입은 키워드 type으로 정의한다. 먼저 type을 이용하여 이미 있는 타입에 새로운 이름을 붙일 수 있다.

```
type var = string
type vector = float list
type matrix = float list list
```

또는 새로운 값의 집합을 만들고 타입으로 정의할 수 있다. 예를 들어, '요일'의 집합을 나타내는 타입 days를 다음과 같이 정의할 수 있다.

```
# type days = Mon | Tue | Wed | Thu | Fri | Sat | Sun;;
# Mon;;
- : days = Mon
# Tue;;
- : days = Tue
```

키워드 type 다음에는 정의하고자 하는 타입의 이름(days)을 적어주고, = 다음에는 그 타입에 속하는 값들을 |로 구분하여 나열하였다. Mon, ..., Sun을 생성자(constructor)라고 하고 각각 days 타입의 서로 다른 값을 의미한다. OCaml에서 타입 이름은 소문자로

시작하고 생성자는 대문자로 시작한다. 타입을 새로 정의하였으면 그 타입의 값들에 대해서 동작하는 함수를 정의할 수 있다. 예를 들어, 요일을 입력으로 받아서 다음 요일을 반환하는 함수 nextday를 다음과 같이 정의할 수 있다.

```
# let nextday d =
  match d with
  | Mon -> Tue | Tue -> Wed | Wed -> Thu | Thu -> Fri
  | Fri -> Sat  | Sat -> Sun | Sun -> Mon ;;
val nextday : days -> days = <fun>
# nextday Mon;;
- : days = Tue
```

생성자가 다른 값을 인자로 가지도록 정의할 수도 있다. 예를 들어, 사각형 또는 원을 값으로 가지는 타입 shape을 정의해보자.

```
# type shape = Rect of int * int | Circle of int;;
type shape = Rect of int * int | Circle of int
```

사각형을 의미하는 생성자 Rect는 가로와 세로 길이를 인자로 가지도록 정의하였고, 원을 의미하는 생성자 Circle은 반지름을 인자로 가지도록 하였다. 이렇게 정의한 타입에 속하는 값들의 예는 다음과 같다:

```
# Rect (2,3);;
- : shape = Rect (2, 3)
# Circle 5;;
- : shape = Circle 5
```

이렇게 정의된 도형의 넓이를 구하는 함수는 다음과 같이 작성할 수 있다.

```
# let area s =
    match s with
      Rect (w,h) -> w * h
    | Circle r -> r * r * 3;;
val area : shape -> int = <fun>
# area (Rect (2,3));;
- : int = 6
# area (Circle 5);;
- : int = 75
```

위의 예에서 편의상 원주율의 값을 3으로 근사하였다.

타입을 귀납적으로 정의하는 것도 가능하다. 예를 들어, 1장에서 정의했던 정수 리스트의 집합을 생각해보자.

$$\frac{}{\mathsf{nil}} \qquad \frac{l}{n \cdot l}\ n \in \mathbb{Z}$$

OCaml에서 위의 집합은 아래와 같이 정의할 수 있다.

```
# type intlist = Nil | Cons of int * intlist;;
type intlist = Nil | Cons of int * intlist
```

집합의 이름(타입)을 `intlist`라고 하였고, 그 집합의 원소를 만드는 두 가지 방법을 정의하였다. 먼저 `Nil`은 빈 리스트를 뜻하는 생성자이다. `Cons`는 주어진 리스트 맨 앞에 원소를 하나 추가하여 새로운 리스트를 만드는 생성자이다. 예를 들어, 다음과 같이 리스트를 만들 수 있다.

```
# Nil;;
- : intlist = Nil
# Cons (1, Nil);;
- : intlist = Cons (1, Nil)
```

```
# Cons (1, Cons (2, Nil));;
- : intlist = Cons (1, Cons (2, Nil))
```

예를 들어, `Cons (1, Cons (2, Nil))`는 리스트 `[1;2]`를 뜻한다. 이제 리스트를 다루는 함수를 작성할 수 있다. 리스트의 길이를 구하는 함수는 다음과 같다.

```
# let rec length l =
    match l with
    | Nil -> 0
    | Cons (_, l') -> 1 + length l';;
val length : intlist -> int = <fun>
# length (Cons (1, Cons (2, Nil)));;
- : int = 2
```

다음과 같이 정의했던 정수식을 OCaml 데이터 타입으로 정의해보자.

$$\overline{n}\ n \in \mathbb{Z} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ - } E_2} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ + } E_2} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ * } E_2} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ / } E_2}$$

위 문법구조를 다음과 같이 정의할 수 있다.

```
type exp =
    Int of int
  | Minus of exp * exp
  | Plus of exp * exp
  | Mult of exp * exp
  | Div of exp * exp
```

예를 들어, `(1+2)*(3/3)`은 다음과 같이 표현된다.

```
# Mult(Plus(Int 1, Int 2), Div(Int 3, Int 3));;
- : exp = Mult (Plus (Int 1, Int 2), Div (Int 3, Int 3))
```

정수식의 의미를 나타내는 귀납 규칙은 다음과 같이 재귀 함수로 구현할 수 있다.

```
# let rec eval exp =
    match exp with
    | Int n -> n
    | Plus (e1, e2) -> (eval e1) + (eval e2)
    | Mult (e1, e2) -> (eval e1) * (eval e2)
    | Minus (e1, e2) -> (eval e1) - (eval e2)
    | Div (e1, e2) ->
      let n1 = eval e1 in
      let n2 = eval e2 in
        if n2 <> 0 then n1 / n2
        else raise (Failure "division by 0");;
val eval : exp -> int = <fun>
# eval (Mult (Plus (Int 1, Int 2),
              Div (Int 3, Int 3)));;
- : int = 3
```

의미구조 정의를 그대로 재귀 함수로 옮긴 것이다. 0으로 나누는 경우는 의미가 정의되지 않으므로 예외(exception)를 발생시켰다.

예외 처리

OCaml은 의미가 정의되지 않는 식을 계산하여 런타임 오류(run-time error)가 발생하는 경우 예외(exception)를 발생시킨다. 예를 들어, 어떤 수를 0으로 나누는 경우 다음과 같이 `Division_by_zero` 예외가 발생한다.

```
# let div a b = a / b;;
val div : int -> int -> int = <fun>
# div 10 5;;
```

```
- : int = 2
# div 10 0;;
Exception: Division_by_zero.
```

실행 중 발생하는 예외를 처리하려면 `try ... with`를 이용한다.

```
# let div a b =
    try
      a / b
    with Division_by_zero -> 0;;
val div : int -> int -> int = <fun>
# div 10 5;;
- : int = 2
# div 10 0;;
- : int = 0
```

새로운 예외를 다음과 같이 키워드 `exception`을 이용하여 정의하고 사용할 수 있다.

```
# exception Fail;;
exception Fail
# let div a b =
    if b = 0 then raise Fail
    else a / b;;
val div : int -> int -> int = <fun>
# div 10 5;;
- : int = 2
# div 10 0;;
Exception: Fail.
# try
    div 10 0
  with Fail -> 0;;
- : int = 0
```

모듈

모듈(module)은 타입과 값의 모음이다. 관련된 기능을 한 곳에 모으고 속 내용을 감추는 역할을 한다. 예를 들어, 큐(queue) 자료구조를 다음과 같이 모듈로 정의해보자.

```
module IntQueue = struct
  type t = int list
  exception E
  let empty = []
  let enq q x = q @ [x]
  let is_empty q = q = []
  let deq q =
    match q with
    | [] -> raise E
    | h::t -> (h, t)
  let rec print q =
    match q with
    | [] -> print_string "\n"
    | h::t -> print_int h; print_string " "; print t
end
```

큐를 리스트로 구현한 경우인데, 모듈의 사용자는 구현 디테일을 몰라도 다음과 같이 사용할 수 있다.

```
let q0 = IntQueue.empty
let q1 = IntQueue.enq q0 1
let q2 = IntQueue.enq q1 2
let (_,q3) = IntQueue.deq q2
let _  = IntQueue.print q1
let _  = IntQueue.print q2
let _  = IntQueue.print q3
```

큐를 만들고, 원소들을 추가 및 삭제한 후 상태를 출력하였다. 출력 결과는 아래와 같다.

```
1
1 2
2
```

지금까지 OCaml의 기본 특징들을 설명하였다. 이 정도만 알아두어도 기본적인 프로그램을 작성하는 데 무리가 없을 것이다. 다음 두 절에서는 함수형 프로그래밍에서 많이 쓰이는 두 가지 프로그래밍 스타일인 재귀 함수와 고차 함수에 대해 공부한다.

2.2 재귀 함수

함수형 프로그래밍에서는 반복을 표현할 때 반복문 대신 재귀 함수를 주로 사용한다. 재귀 함수는 반복문의 개념을 포함하는 일반적인 개념일 뿐 아니라 재귀적으로 생각하면 문제를 다른 각도에서 보다 쉽게 해결할 수 있는 경우가 많기 때문이다.

재귀 함수 연습

재귀적으로 문제를 푸는 방법을 연습해보자. 모든 재귀 함수는 다음의 구조를 가진다.

- 풀고자 하는 문제의 크기가 충분히 작은 경우에는 직접 푼다.
- 그렇지 않은 경우,
    1. 원래 문제를 동일한 구조의 부분 문제들로 쪼갠다.

2. 쪼개진 부분 문제들의 해들을 재귀적으로 구한다.

3. 부분 문제들의 해를 모아서 원래 문제의 해를 구성한다.

리스트 길이 구하기 위의 방법을 리스트 길이를 구하는 문제에 적용해보자. 먼저 빈 리스트의 경우, 그 길이는 0으로 바로 구할 수 있다. 리스트가 비어 있지 않으면 머리(head)와 꼬리(tail)로 구분할 수 있고, 꼬리의 길이를 재귀적으로 구한다. 그 길이를 n이라고 할 때, 원래 리스트의 길이는 $n+1$이 된다. 이 과정을 OCaml로 표현하면 다음과 같다.

```
let rec length l =
  match l with
  | [] -> 0
  | hd::tl -> 1 + length tl
```

리스트 이어붙이기 두 리스트를 이어붙이는 함수 `append`를 작성해보자. OCaml에서 기본 연산자 `@`로 제공하고 있지만 재귀 함수 연습을 위해 직접 정의해보자. 예를 들어, 다음과 같이 동작해야 한다.

```
# append [1; 2; 3] [4; 5; 6; 7];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
# append [2; 4; 6] [8; 10];;
- : int list = [2; 4; 6; 8; 10]
```

`append`는 인자로 두 리스트 `l1`과 `l2`를 받아서 `l1@l2`에 해당하는 리스트를 반환해야 한다. 첫 번째 인자 `l1`에 대해서 재귀적으로 생각해보자. `l1`이 빈 리스트인 경우, `l2`가 두 리스트를 이어

붙인 리스트가 된다. l1이 빈 리스트가 아니라면 머리와 꼬리로 나눌 수 있고, 먼저 꼬리와 리스트 l2를 이어붙인 리스트를 재귀적으로 구한다. 그 리스트 앞에 l1의 머리를 추가하면 원래 문제의 답이 된다. OCaml로 표현하면 다음과 같다:

```
let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | hd::tl -> hd::(append tl l2)
```

리스트 뒤집기 리스트를 뒤집는 함수 reverse를 작성해보자.

```
# reverse [1; 2; 3];;
- : int list = [3; 2; 1]
# reverse ["C"; "Java"; "OCaml"];;
- : string list = ["OCaml"; "Java"; "C"]
```

빈 리스트의 경우 뒤집어도 빈 리스트이다. 리스트가 비어 있지 않으면 꼬리를 먼저 뒤집고 머리를 맨 뒤에 이어붙이면 된다. OCaml로 작성하면 다음과 같다.

```
let rec reverse l =
  match l with
  | [] -> []
  | hd::tl -> (reverse tl)@[hd]
```

리스트의 n번째 원소 찾기 리스트의 n번째 원소를 반환하는 함수 nth를 작성해보자(n은 자연수라 가정한다). 주어진 리스트의 범위를 넘어서는 접근에 대해서는 예외를 발생시켜야 한다.

```
# nth [1;2;3] 0;;
- : int = 1
```

```
# nth [1;2;3] 1;;
- : int = 2
# nth [1;2;3] 2;;
- : int = 3
# nth [1;2;3] 3;;
Exception: Failure "list is too short".
```

먼저 빈 리스트에 대해서는 어떤 n에 대해서도 n번째 원소가 정의되지 않으므로 예외를 발생시킨다. 리스트가 비어 있지 않은 경우를 생각해보자. 이 경우 n이 0이면 리스트의 머리가 n번째 원소이다. n이 0이 아니면 꼬리의 $n-1$번째 원소가 원래 리스트의 n번째 원소에 해당한다. OCaml로 표현하면 다음과 같다:

```
let rec nth l n =
  match l with
  | [] -> raise (Failure "list is too short")
  | hd::tl -> if n = 0 then hd else nth tl (n-1)
```

리스트에서 처음 등장하는 특정 원소 지우기 어떤 원소를 리스트에서 제거한 리스트를 반환하는 함수를 작성해보자.

```
# remove_first 2 [1; 2; 3];;
- : int list = [1; 3]
# remove_first 2 [1; 2; 3; 2];;
- : int list = [1; 3; 2]
# remove_first 4 [1;2;3];;
- : int list = [1; 2; 3]
# remove_first [1; 2] [[1; 2; 3]; [1; 2]; [2; 3]];;
- : int list list = [[1; 2; 3]; [2; 3]]
```

인자로 주어진 리스트가 비어 있으면 제거할 원소가 없으므로 빈 리스트를 반환하면 된다. 리스트가 비어 있지 않으면 머리와 꼬리

로 나눌 수 있다. 제거하고자 하는 원소가 머리와 일치하면 꼬리를 그대로 반환하면 된다. 일치하지 않으면 꼬리에서 첫 번째로 등장하는 원소를 제거한 리스트를 구하고 머리를 그 앞에 불이면 된다. OCaml로 표현하면 다음과 같다:

```
let rec remove_first a l =
  match l with
  | [] -> []
  | hd::tl ->
    if a = hd then tl
    else hd::(remove_first a tl)
```

삽입 정렬 삽입 정렬(insertion sort)을 통해 리스트를 정렬하는 함수를 구현해보자. 먼저 정렬된 리스트와 한 원소가 주어졌을 때 그 원소의 적절한 위치를 찾아 리스트에 삽입하는 함수 `insert`를 작성해보자. 예를 들어, 다음과 같이 동작해야 한다:

```
# insert 2 [1;3];;
- : int list = [1; 2; 3]
# insert 1 [2;3];;
- : int list = [1; 2; 3]
# insert 3 [1;2];;
- : int list = [1; 2; 3]
# insert 4 [];;
- : int list = [4]
```

다음과 같이 재귀 함수로 작성할 수 있다.

```
let rec insert a l =
  match l with
  | [] -> [a]
  | hd::tl ->
```

```
    if a <= hd then a::hd::tl
    else hd::(insert a tl)
```

빈 리스트에 원소 a를 삽입하면 [a]가 된다. 비어 있지 않은 리스트 hd::tl에 원소 a를 삽입하는 과정은, 먼저 a가 리스트의 첫 번째 원소인 hd보다 같거나 작으면 hd앞에 a를 놓는 것이고 그렇지 않으면 a를 tl에 재귀적으로 삽입하고 그 앞에 hd를 놓는 것이다. 함수 insert를 정의했으면 정렬 함수 sort는 다음과 같이 간단히 정의된다.

```
let rec sort l =
  match l with
  | [] -> []
  | hd::tl -> insert hd (sort tl)
```

빈 리스트는 정렬해도 빈 리스트이다. 비어 있지 않은 리스트 hd::tl을 정렬하려면 먼저 tl을 재귀적으로 정렬한 후에 insert를 이용하여 hd를 제 위치에 넣으면 된다.

함수형 vs. 명령형 프로그래밍

위에서 정의한 삽입 정렬 함수를 C 언어에서 반복문으로 구현한 아래 코드와 비교해보자.

```
void insert_sort(int arr[], int len) {
  int i, j;
  int tmp;
  for(i = 1; i < len; i++) {
    tmp = arr[i];
    j = i - 1;
    while(j >= 0 && arr[j] > tmp) {
```

```
      arr[j + 1] = arr[j];
      j = j - 1;
    }
    arr[j + 1] = tmp;
  }
}
```

재귀 함수로 작성한 경우가 더 간결하고 가독성이 높다. 재귀 함수를 이용하는 함수형 프로그램이 더 이해하기 쉬운 이유는 문제 푸는 절차를 기술하는 대신 문제 자체를 기술하도록 유도하기 때문이다. 예를 들어, 위에서 정의한 sort의 정의를 다시 보면, 리스트가 비어 있지 않은 경우인 hd::tl을 정렬한 리스트는 tl을 정렬하고 hd를 삽입한 리스트와 일치해야 한다는 조건, 즉 정렬된 리스트가 만족해야 하는 조건을 기술하고 있다. 구현하고자 하는 함수 sort가 다음의 동치 관계를 만족해야 한다고 기술한 것이다.

```
sort (hd::tl) = insert hd (sort tl)
```

OCaml 구현에서는 위와 같이 sort가 만족해야 하는 조건을 재귀 함수로 구현한 것 뿐이다. 다른 예로 팩토리얼을 계산하는 함수 factorial은 인자로 주어진 수가 0보다 큰 경우 다음 조건을 만족해야 한다.

```
factorial n = n * factorial (n-1)
```

인자가 0인 경우와 함께 위 명세를 재귀 함수로 다음과 같이 구현할 수 있다.

```
let rec factorial n =
```

```
  if n = 0 then 1 else n * factorial (n-1)
```

반면에 명령형 프로그래밍은 문제 푸는 절차를 기술하도록 유도하는 패러다임이다. 예를 들어, C 언어로 팩토리얼 함수를 작성한다면 대부분 다음과 같이 구현할 것이다.

```
int factorial (int n) {
  int i; int r = 1;
  for (i = 0; i < n; i++)
    r = r * i;
  return r;
}
```

이 코드는 위의 팩토리얼 함수가 만족해야 하는 조건을 만족하는 여러 구현 중에서 하나를 기술한 것이다. 즉, 명령형 프로그래밍에서는 문제를 기술하는 것 외에 한 단계 더 나아가 구체적인 구현 방안까지 제시해야 한다. 반면에 함수형 프로그래밍에서는 문제를 기술하는 것에 중점을 두며, 이러한 점에서 선언적 프로그래밍(declarative programming)이라 부르기도 한다.

재귀 함수의 비용

참고로 OCaml을 비롯한 함수형 프로그래밍 언어에서는 재귀 함수가 반복문에 비해 더 비싸지 않다. 예를 들어, C 에서 다음과 같은 함수를 호출하면 스택 넘침(stack overflow)이 발생하지만

```
void f() { f(); }       /* stack overflow */
```

OCaml에서는 함수가 호출될 때 메모리를 추가적으로 소모하지 않고 무한히 돈다.

```
let rec f () = f ()   (* infinite loop *)
```

두 경우 모두 함수 f는 무한히 도는 반복을 표현한 것인데, OCaml은 그 의미 그대로 실행 시켜줄 수 있는 것이다. 이 예에서 알 수 있듯이 OCaml에서는 함수가 재귀적으로 정의되었다고 해서 반복문에 비해 특별히 더 비싸지 않다. 함수 호출이 비싼 경우는 그 함수가 기술하고 있는 계산 자체가 비싼 경우이지, 단순히 계산이 재귀적으로 기술되어 있다고 해서 비싸지는 않다.

좀 더 자세히 설명하면, for나 while등의 반복문은 항상 꼬리 재귀 함수(tail-recursive function)의 형태로 변환될 수 있고, 꼬리 재귀 함수들의 호출은 비싸지 않다. 꼬리 재귀 함수란 재귀 함수를 호출한 후 더 이상 할일이 남아 있지 않은 재귀 함수를 가리킨다. 예를 들어, 리스트의 마지막 원소를 반환하는 함수 last를 생각해 보자:

```
let rec last l =
  match l with
  | [a] -> a
  | _::tl -> last tl  (* tail-recursive call *)
```

여기서 함수 호출 last tl은 꼬리 재귀 호출에 해당한다. last tl의 결과값을 계산하고 그 값을 이용해서 추가적으로 해야 할 일이 남아 있지 않다는 뜻이다. 그 결과값을 가지고 추가적으로 할 일이 없으므로 함수가 꼬리 재귀의 형태로 호출되면 호출된 지점으로 반환할 필요가 없고, 따라서 함수 호출 시 추가적인 메모리를 저장할

필요가 없다. 이러한 꼬리 호출 최적화(tail-call optimization)를 많은 함수형 언어 컴파일러들이 지원하고 있다.

반면에 팩토리얼 함수와 같이 꼬리 재귀 함수가 아닌 경우에는 재귀 함수 호출 시 추가적인 메모리가 필요하다.

```
let rec factorial n =
  if n = 1 then 1 else n * factorial (n - 1)
```

재귀 호출 `factorial (a-1)`을 계산한 후 그 결과를 가지고 할 일(a를 곱하는 일)이 남아 있으므로 꼬리 재귀 호출이 아니다. 이 경우 재귀 호출은 메모리를 소모하며 큰 수에 대해서 `factorial`을 호출하면 스택 넘침이 발생할 수 있다.

참고로 꼬리 재귀로 정의되지 않은 함수를 항상 꼬리 재귀의 형태로 변환할 수 있다. 모든 재귀 함수를 반복문으로 표현할 수 있는 것과 같은 원리이다. 예를 들어 `factorial`은 다음과 같이 꼬리 재귀의 형태로 정의할 수 있다:

```
let rec factorial n r =
  if n = 1 then r else factorial (n - 1) (r * n)
```

예를 들어, 5 팩토리얼은 `factorial 5 1`과 같이 구한다.

2.3 고차 함수

함수형 프로그래밍에서는 재귀 함수 외에도 고차 함수(higher-order function)를 많이 사용한다. 고차 함수란 다른 함수를 인자로 받거나 반환하는 함수를 뜻한다. 고차 함수는 프로그래밍 패턴을 추상

화하여 재사용 가능하게 함으로써 프로그램을 더욱 상위에서 작성할 수 있게 해 준다. 자주 사용되는 고차 함수들인 `map`, `filter`, `fold`를 중심으로 살펴보자.

map

아래 정의한 세 함수 `inc_all`, `sqaure_all`, `cube_all`은 각각 주어진 리스트의 각 원소를 증가시키고, 제곱하고, 세제곱하는 함수이다.

```
let rec inc_all l =
  match l with
  | [] -> []
  | hd::tl -> (hd+1)::(inc_all tl)

let rec square_all l =
  match l with
  | [] -> []
  | hd::tl -> (hd*hd)::(square_all tl)

let rec cube_all l =
  match l with
  | [] -> []
  | hd::tl -> (hd*hd*hd)::(cube_all tl)
```

이들 함수들은 모두 공통된 패턴을 따라 정의되어 있다. 오직 차이점은 리스트의 각 원소에 적용하는 연산에 있다. 이 연산을 일반적으로 `f`라고 하면 위의 세 함수가 공유하고 있는 프로그래밍 패턴을 다음과 같은 고차 함수 `map`으로 표현할 수 있다.

```
let rec map f l =
```

```
match l with
| [] -> []
| hd::tl -> (f hd)::(map f tl)
```

map은 리스트 l외에 각 원소에 적용할 연산을 뜻하는 함수 f를 인자로 받아서, 리스트 l의 각 원소 a를 f a의 값으로 치환한 리스트를 생성한다. 즉, l이 리스트 [a1;a2;...;ak]일 때 map f l은 새로운 리스트 [f a1; f a2; ...; f ak]를 만들어낸다. map의 타입을 보면 이러한 의미를 어느정도 파악할 수 있다.

```
map : ('a -> 'b) -> 'a list -> 'b list
```

적용할 함수의 타입이 'a -> 'b이고, 'a 타입의 리스트를 입력으로 받아서 'b 타입의 리스트를 출력으로 만들어냄을 뜻한다. 이와 같이 일반적으로 함수의 타입은 그 함수가 하는일의 요약본(abstraction)이라 할 수 있다.

고차 함수 map을 이용하면 위의 세 함수들을 다음과 같이 간결하게 정의할 수 있다.

```
let inc_all l = map (fun x -> x + 1) l
let square_all l = map (fun x -> x * x) l
let cub_all l = map (fun x -> x * x * x) l
```

또는 공통된 인자 l을 생략하여 다음과 같이 정의해도 된다.

```
let inc_all = map (fun x -> x + 1)
let square_all = map (fun x -> x * x)
let cub_all = map (fun x -> x * x * x)
```

이를 앞의 정의와 비교해보자. map을 이용한 정의에는 각 함수의 의미가 상위 레벨에서 더욱 명확하고 직접적으로 표현되어 있음을 알 수 있다. 예를 들어, sqaure_all의 의미는 리스트 l의 각 원소에 함수 fun x -> x * x를 적용하는 것임이 직접 표현되어 있다. 반면에 map을 이용하지 않고 재귀적으로 작성한 경우에는 이러한 의미가 코드 내에 숨겨져 있고 바로 드러나지는 않는다.

잘 짜여진 프로그램이란 다른 사람이 쉽게 이해할 수 있는 프로그램이다. 구현의 세부 디테일을 모르더라도 상위 레벨에서 이해할 수 있도록 작성되어 있기 때문이다. 이런 측면에서 좋은 프로그래밍 언어란 아이디어를 보다 상위에서 표현할 수 있도록 해 주는 언어이다. 함수형 프로그래밍에서 고차 함수는 프로그램의 추상화 레벨을 높이는 데 있어서 큰 역할을 한다.

filter

아래 두 함수도 하는 일은 다르지만 동일한 패턴을 따라 정의되어 있다.

```
let rec even l =
  match l with
  | [] -> []
  | hd::tl ->
    if hd mod 2 = 0 then hd::(even tl)
    else even tl

let rec greater_than_five l =
  match l with
  | [] -> []
```

```
  | hd::tl ->
    if hd > 5 then hd::(greater_than_five tl)
    else greater_than_five tl
```

even은 리스트 l에서 짝수만 골라내고, greater_than_five는 5보다 큰 수만 골라낸다. 즉, 모두 주어진 리스트로부터 특정한 조건을 만족하는 원소들을 골라내는 함수이고, 고르는 조건만 다를 뿐이다. 이러한 공통 패턴을 다음의 고차 함수로 추상화할 수 있다.

```
let rec filter p l =
  match l with
  | [] -> []
  | hd::tl ->
    if p hd then hd::(filter p tl)
    else filter p tl
```

filter는 함수 p와 리스트 l을 받아서 l의 원소들 중에서 p를 만족하는 원소들만 모으는 일을 한다. 이때 함수 p는 결과값으로 부울값을 반환하는 함수(predicate)이어야 한다. 즉, filter의 타입은 다음과 같다.

```
filter : ('a -> bool) -> 'a list -> 'a list
```

고차 함수 filter를 이용하여 위의 두 함수를 아래와 같이 정의할 수 있다.

```
let even l = filter (fun x -> x mod 2 = 0) l
let greater_than_five l = filter (fun x -> x > 5) l
```

fold

함수형 프로그래밍에서 가장 자주 사용되는 고차 함수 가운데 하나는 fold이다. 아래 두 함수를 비교해보자.

```
let rec sum l =
  match l with
  | [] -> 0
  | hd::tl -> hd + (sum tl)

let rec prod l =
  match l with
  | [] -> 1
  | hd::tl -> hd * (prod tl)
```

두 함수 모두 주어진 리스트의 각 원소를 순회하면서 어떤 연산을 누적 적용하는 패턴을 따르고 있다. 예를 들어, 리스트 [1; 2; 3]에 대해서 위 두 함수를 적용하는 과정을 나타내면 다음과 같다.

```
sum [1; 2; 3] = 1 + (2 + (3 + 0))
prod [1; 2; 3] = 1 * (2 * (3 * 1))
```

두 함수가 하는일의 차이는 누적해서 적용할 연산과 초기값이다. sum의 경우 연산자가 +이고 초기값은 0이며, prod의 연산자는 *이고 초기값은 1이다. 이 두 가지를 추가적인 인자로 가지는 고차 함수 fold_right을 다음과 같이 정의할 수 있다.

```
let rec fold_right f l a =
  match l with
  | [] -> a
  | hd::tl -> f hd (fold_right f tl a)
```

f는 누적시킬 연산, l은 리스트, a는 초기값이다. fold_right을 이용하여 다음과 같이 함수 sum과 prod를 정의할 수 있다.

```
let sum  lst = fold_right (fun x y -> x + y) lst 0
let prod lst = fold_right (fun x y -> x * y) lst 1
```

각 함수의 의미가 더 상위에서 직접적으로 표현되었다. sum은 초기값 0으로 시작하여 리스트의 각 원소에 덧셈을 누적시켜 적용하는 함수이고, prod는 초기값 1로 시작하여 리스트의 각 원소에 곱셈을 누적시켜 적용하는 함수이다.

위의 예와 동일한 의미를 가지지만 sum과 prod가 아래와 같이 꼬리 재귀 함수로 작성되어 있는 경우를 생각해보자.

```
let rec sum a l =
  match l with
  | [] -> a
  | hd::tl -> sum (a + hd) tl

let rec prod a l =
  match l with
  | [] -> a
  | hd::tl -> prod (a * hd) tl
```

위의 두 함수의 공통패턴을 정의한 고차 함수를 fold_left라 부른다.

```
let rec fold_left f a l =
  match l with
  | [] -> a
  | hd::tl -> fold_left f (f a hd) tl
```

fold_left를 이용하여 sum과 prod를 정의하면 다음과 같다.

```
let sum a l = fold_left (fun x y -> x + y) a l
let prod a l = fold_left (fun x y -> x * y) a l
```

fold_right와 fold_left의 차이를 좀 더 자세히 살펴보자. 먼저 다음과 같이 타입이 다르다.

```
fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
fold_left  : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

더욱 중요한 차이는 연산을 누적시키는 방향이 다르다는 점이다. fold_right는 리스트의 가장 마지막 원소부터 시작해서 오른쪽에서 왼쪽으로 누적시킨다.

```
fold_right f [x;y;z] init = f x (f y (f z init))
```

반면에 fold_left는 리스트의 가장 왼쪽 원소부터 시작해서 오른쪽으로 누적시킨다.

```
fold_left f init [x;y;z] = f (f (f init x) y) z
```

따라서 적용하는 연산 f가 결합 법칙(associative law)을 만족하지 않는 경우 두 결과가 다를 수 있다.

```
# fold_right (fun x y -> x - y) [1;2;3] 0;;
- : int 2
# fold_left (fun x y -> x - y) 0 [1;2;3];;
- : int = -6
```

마지막으로 fold_left는 꼬리 재귀 함수이다. 리스트 길이가 긴 경우에는 가능하면 fold_left를 사용하는 것이 좋다.

2.4 연습 문제

문제 1 두 수 n, m $(n \leq m)$을 받아서 n이상 m이하의 수로 구성된 리스트를 반환하는 함수 range를 작성하시오.

```
range : int -> int -> int list
```

예를 들어, range 3 7 는 [3;4;5;6;7]를 만들어낸다.

문제 2 리스트의 리스트를 받아서 모든 원소들을 순서대로 포함하는 하나의 리스트를 반환하는 함수 concat을 작성하시오.

```
concat: 'a list list -> 'a list
```

예를 들어, concat [[1;2];[3;4;5]는 [1;2;3;4;5]를 만들어낸다.

문제 3 두 리스트 a와 b를 순차적으로 결합하는 함수 zipper를 작성하시오.

```
zipper: int list -> int list -> int list
```

순차적인 결합이란 리스트 a의 i번째 원소가 리스트 b의 i번째 원소 앞에 오는 것을 의미한다. 짝이 맞지 않는 원소들은 뒤에 순서대로 붙인다. 예를 들어,

```
# zipper [1;3;5] [2;4;6];;
- : int list = [1; 2; 3; 4; 5; 6]
# zipper [1;3] [2;4;6;8];;
- : int list = [1; 2; 3; 4; 6; 8]
# zipper [1;3;5;7] [2;4];;
- : int list = [1; 2; 3; 4; 5; 7]
```

문제 4 두 원소로 구성된 튜플의 리스트를 두 리스트로 분해하는 함수 unzip을 작성하시오.

```
unzip: ('a * 'b) list -> 'a list * 'b list
```

예를 들어,

```
unzip [(1,"one");(2,"two");(3,"three")]
```

은 ([1;2;3],["one";"two";"three"])을 계산한다.

문제 5 리스트 l과 정수 n을 받아서 l의 첫 n개 원소를 제외한 나머지 리스트를 구하는 함수 drop을 작성하시오.

```
drop : 'a list -> int -> 'a list
```

예를 들어,

```
drop [1;2;3;4;5] 2 = [3; 4; 5]
drop [1;2] 3 = []
drop ["C"; "Java"; "OCaml"] 2 = ["OCaml"]
```

문제 6 고차 함수 sigma를 작성하시오.

```
sigma : (int -> int) -> int -> int -> int
```

sigma f a b는 다음을 계산한다.

$$\sum_{i=a}^{b} f(i).$$

예를 들어,

```
sigma (fun x -> x) 1 10
```

는 55를,

```
sigma (fun x -> x*x) 1 7
```

는 140을 계산한다.

문제 7 고차 함수 iter를 작성하시오.

```
iter : int * (int -> int) -> (int -> int)
```

다음의 조건을 만족해야 한다.

$$\texttt{iter}(n, f) = \underbrace{f \circ \cdots \circ f}_{n}$$

$n = 0$인 경우, `iter`(n, f)는 인자를 그대로 반환하는 함수(identity function)이다. $n > 0$인 경우, `iter`(n, f)는 f를 n번 적용하는 함수이다. 예를 들어,

```
iter(n, fun x -> 2+x) 0
```

는 $2 \times n$를 계산한다.

문제 8 `fold_right`을 이용하여 고차 함수 `all`을 작성하시오.

```
all : ('a -> bool) -> 'a list -> bool
```

`all p l`은 리스트 `l`의 모든 원소들이 함수 `p`의 값을 참으로 만드는지 여부를 나타낸다. 예를 들어,

```
all (fun x -> x > 5) [7;8;9]
```

는 *true*를 계산한다.

문제 9 `fold_left`를 이용하여 정수 리스트를 숫자로 변환하는 함수를 작성하시오.

```
lst2int : int list -> int
```

예를 들어, `lst2int [1;2;3]`는 123을 계산한다. 리스트의 원소들은 0이상 9이하의 수라고 가정한다.

문제 10 아래 함수들을 `fold_right`와 `fold_left`로 다시 정의하시오.

1. ```
   let rec length l =
       match l with
       [] -> 0
       |h::t -> 1 + length t
   ```

2. ```
   let rec reverse l =
       match l with
       | [] -> []
       | hd::tl -> (reverse tl)@[hd]
   ```

3. ```
   let rec is_all_pos l =
     match l with
     | [] -> true
     | hd::tl -> (hd > 0) && (is_all_pos tl)
   ```

4. ```
   let rec map f l =
       match l with
       | [] -> []
       | hd::tl -> (f hd)::(map f tl)
   ```

5. ```
   let rec filter p l =
       match l with
   ```
   ```

```
  | [] -> []
  | hd::tl ->
    if p hd then hd::(filter p tl)
    else filter p tl
```

문제 11 귀납적으로 정의한 자연수의 집합을 생각하자.

$$\frac{}{0} \qquad \frac{n}{n+1}$$

OCaml에서는 다음과 같이 정의할 수 있다.

```
type nat = ZERO | SUCC of nat
```

예를 들어, SUCC ZERO는 1을, SUCC (SUCC ZERO)는 2를 뜻한다. 이렇게 정의된 자연수들 간의 덧셈과 곱셈을 정의하시오.

```
natadd : nat -> nat -> nat
natmul : nat -> nat -> nat
```

예를 들어,

```
# let two = SUCC (SUCC ZERO);;
val two : nat = SUCC (SUCC ZERO)
# let three = SUCC (SUCC (SUCC ZERO));;
val three : nat = SUCC (SUCC (SUCC ZERO))
# natmul two three;;
- : nat = SUCC (SUCC (SUCC (SUCC (SUCC (SUCC ZERO)))))
# natadd two three;;
- : nat = SUCC (SUCC (SUCC (SUCC (SUCC ZERO))))
```

문제 12 산술식 aexp를 다음과 같이 정의하자.

```
type aexp =
 | Const of int
 | Var of string
 | Power of string * int
 | Times of aexp list
 | Sum of aexp list
```

산술식과 문자를 뜻하는 문자열을 입력으로 받아서 그 문자에 대해서 미분한 산술식을 반환하는 함수를 작성하시오.

```
diff : aexp * string -> aexp
```

예를 들어, 산술식 $x^2 + 2x + 1$는 다음과 같이 표현된다.

```
Sum [Power ("x", 2); Times [Const 2; Var "x"]; Const 1]
```

이 식을 e라고 하면 diff(e, "x")는, e를 x에 대해서 미분한 $2x+2$를 반환해야 한다.

```
Sum [Times [Const 2; Var "x"]; Const 2]
```

3

변수와 환경

이제부터 본격적으로 프로그래밍 언어를 디자인하고 구현해보자. 먼저 1장에서 정의한 정수식 언어를 확장하여 변수와 조건식을 사용할 수 있도록 하자.

3.1 문법구조

확장된 언어의 문법구조는 그림 3.1과 같다. 정수식 언어에 네 가지 구문을 추가하였다.

- 프로그램에서 식이 올 수 있는 임의의 위치에 변수 x를 사용할 수 있게 하였다.[1)]
- `let` x `=` E_1 `in` E_2는 변수 x를 선언하는 식이다. E_1의 값을 x라고 한 후 E_2의 값을 계산한다. 이때, 변수 x를 사용할 수 있는 유효범위(scope)는 E_2이다.
- `if` E_1 `then` E_2 `else` E_3는 조건식(conditional expression)으로 다음과 같이 계산된다. 먼저 식 E_1을 계산하면 참($true$) 또는 거짓($false$)값이 나와야 한다. 만약 E_1의 값이 $true$이면 E_2를 계산하고 E_1의 값이 $false$이면 E_3를 계산한다.

1) 여기서 x는 특정 변수가 아닌 임의의 변수를 뜻하는 메타 변수이다.

$$
\begin{array}{rcll}
E & \rightarrow & n & \\
 & | & E_1 \texttt{ + } E_2 & \\
 & | & E_1 \texttt{ - } E_2 & \\
 & | & E_1 \texttt{ * } E_2 & \\
 & | & E_1 \texttt{ / } E_2 & \\
 & | & x & \text{변수} \\
 & | & \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 & \texttt{let}\text{식} \\
 & | & \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 & \text{조건식} \\
 & | & \texttt{iszero } E & \text{부울식}
\end{array}
$$

그림 3.1: 문법구조

- 조건식을 사용하려면 참/거짓값을 만들어내는 식이 필요하므로 구문 `iszero` E를 추가하였다. E의 값을 계산하여 그 값이 0이면 *true* 아니면 *false*를 계산한다.

OCaml의 문법과 유사하므로 어떤 프로그램들을 작성할 수 있을지 쉽게 감이 올 것이다. 몇가지 예를 들어보자.

예제 1 아래는 변수 x의 값을 1로 정의한 후 x+2의 값을 계산하는 프로그램이다. 결과는 3이다.

```
let x = 1 in x + 2
```

예제 2 다음과 같이 let식의 E_2 자리에 또 다른 let식이 올 수 있다.

```
let x = 1
in let y = 2
   in x + y
```

식 x + y에서 x와 y가 각각 1, 2를 뜻하므로 프로그램의 계산결과는 3이다.

예제 3 다음과 같이 let식의 E_1 자리에 또다른 let식이 올 수 있다.

```
let x = let y = 2
        in y + 1
in x + 3
```

식 let y = 2 in y + 1의 값은 3이다. 이를 x라고 한 후 x + 3을 계산한 결과는 6이다. 이 때, 식 x + 3을 계산할 때 변수 y는 사용할 수 없음에 주의하자. 변수 y의 유효범위는 식 y + 1에 국한되고 그 외에서는 사용할 수 없다. 즉, 아래 프로그램은 문법적으로는 문제없지만 제대로 실행되지 않는 프로그램이다.

```
let x = let y = 2
        in y + 1
in x + y
```

예제 4 다음과 같이 동일한 이름을 여러번 정의할 수도 있다.

```
let x = 1
in let y = 2
   in let x = 3
      in x + y
```

마지막 줄에서 x + y를 계산할 때의 x는 가장 마지막에 정의된 3을 의미하고 따라서 x + y의 값은 5가 된다.

예제 5 동일한 이름의 변수를 다시 정의한다고 해서 이전 변수의 값이 바뀌는 것은 아니다. `let`을 이용한 변수의 정의는 기존 변수의 값을 변경하는 것이 아닌 새로운 이름을 만드는 것이며, 그 이름은 자신의 유효범위에서만 의미를 가지기 때문이다. 예를 들어 다음의 프로그램을 보자.

```
let x = 1
in let y = let x = 2
           in x + x
   in x + y
```

위의 프로그램을 계산한 결과는 5이다. 첫번째 줄에서 변수 `x`의 값을 1로 정의하였다. 두 번째 줄에서 정의한 `x`의 값은 2이고 따라서 세 번째 줄의 `x + x`의 값은 4이다. 그 결과 변수 `y`의 값은 4가 된다. 마지막 줄의 식 `x + y`에서 변수 `x`는 첫째 줄에서 정의된 `x`를 가리키며 따라서 `x + y` 값은 5가 된다.

예제 6 조건식은 다음과 같이 사용한다. 정수 1을 계산하는 프로그램이다.

```
let x = 1
in let y = 2
   in if iszero (x - 1) then y - 1 else  y + 1
```

예제 7 이제 프로그램이 정수 뿐 아니라 참/거짓값을 계산할 수 있으므로 타입이 맞지 않아서 실행할 수 없는 프로그램들, 즉 타입 오류(type error)를 가지는 프로그램들이 존재하게 된다. 예를 들어, 아래 프로그램을 보자.

```
let x = 1
in let y = iszero x
   in x + y
```

덧셈은 두 정수에 대해서만 정의되는 연산인데 y가 부울값을 가지기 때문에 식 x + y를 계산할 때 타입 오류가 발생한다. 아직 우리 언어는 정적 타입 시스템을 가지고 있지 않으므로 이러한 타입 오류를 실행 전에 찾을 수 없고, 실행 중에 타입 오류가 발생하면서 프로그램 실행이 비정상 종료하게 된다.

3.2 의미구조

이제 언어의 의미구조를 정의해보자. 그림 3.1의 언어로 작성된 프로그램을 실행시키는 규칙이다.

표기법 두 집합 X와 Y에 대해서 $X+Y$를 합집합(union), $X \times Y$를 곱집합(product)[2], $X \setminus Y$를 차집합(difference)이라 하자. $X \to Y$는 X에서 Y로 가는 유한 함수들의 전체 집합을 뜻한다.[3] 함수 $f : X \to Y$가 유한 함수라는 것은 정의역(domain)이 유한 집합이라는 뜻이다. 유한한 개수의 원소에 대해서 값이 정의된 함수임을 뜻한다. $f : X \to Y$가 유한 함수일 때 $\mathrm{Dom}(f)$는 그 정의역을 나타낸다. 즉, $\mathrm{Dom}(f)$는 집합 X의 유한한 부분집합이다. 유한 함수는 테이블로 나타낼 수 있다. 예를 들어 함수 $f(x) = x+1$의 정의역이

2) $X \times Y = \{(x, y) \mid x \in X \wedge y \in Y \ \}$

3) 일반적으로 $X \to Y$는 X에서 Y로 가는 임의의 함수들의 집합을 뜻하지만 이 책에서는 유한 함수들만 고려하기로 한다.

$\{1, 2, 3\}$일 때 f를 $\{1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 4\}$와 같이 테이블로 표현할 수 있다.

3.2.1 환경

프로그램이 변수를 포함하게 되면 그 의미를 프로그램만 가지고 정의할 수 없고 변수의 현재 값을 알려주는 장치인 환경(environment)이 필요하다. 예를 들어, 식 `x+y`의 의미는 변수 `x`와 `y`가 현재 어떤 값을 가지는지에 따라 달라진다. 만약 `x`의 값이 1이고 `y`의 값이 2인 환경

$$\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}$$

에서 계산하면 `x+y`의 값은 3이지만 다른 환경, 예를 들어

$$\{\mathtt{x} \mapsto 2, \mathtt{y} \mapsto 3\}$$

에서 계산하면 동일한 식 `x+y`가 다른 값 5를 가지게 된다.

환경은 변수에서 값으로 가는 유한 함수이다.

$$Env = Var \to Val$$

가능한 모든 환경들의 집합을 Env로 정의하였다. 여기서 Var는 프로그램에서 사용 가능한 변수들의 집합을 나타낸다.

$$Var = \{\mathtt{x}, \mathtt{y}, \mathtt{z}, \ldots\}$$

Val은 프로그램이 다루는 값들의 집합을 뜻한다. 현재 언어에서 값은 정수($\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$) 또는 참/거짓($\mathbb{B} = \{true, false\}$) 밖에 없으므로 값의 집합 Val을 다음과 같이 정의할 수 있다.

$$Val = \mathbb{Z} + \mathbb{B}$$

이와같이 프로그램의 의미를 정의하기 위해 필요한 집합들의 모임 (Env, Var, Val)을 의미공간(semantic domain)이라고 한다.

앞으로 환경에 대한 다음의 표기법을 사용할 것이다.

- 환경을 나타낼때 기호 ρ를 사용할 것이다 ($\rho \in Env$). 필요에 따라 $\rho', \rho'', \rho_1, \rho_2$등도 사용할 것이다.
- $\{x_1 \mapsto v_1, \ldots, x_k \mapsto v_k\}$는 변수 $x_1, \ldots, x_k$가 각각 값 $v_1, \ldots, v_k$로 매핑되어 있는 환경이다. 예를 들어, $\rho = \{x \mapsto 1, y \mapsto 2\}$는 x의 값이 1, y의 값이 2로 정의되어 있는 환경을 의미한다. 그 밖의 다른 이름들에 대해서는 값이 정의되어 있지 않은 환경이다.
- 환경 ρ에서 이름 x가 의미하는 값은 $\rho(x)$로 표기한다. 환경 ρ가 함수이므로 ρ를 x에 적용한 것이다. 예를 들어, $\rho = \{x \mapsto 1, y \mapsto 2\}$일때, $\rho(x)$는 1이고 $\rho(y)$는 2이다.
- $\emptyset$는 빈 환경을 뜻한다. 아무 변수도 정의되어 있지 않은 환경이다. 즉, $\rho = \emptyset$이면 어떤 변수 $x \in Var$에 대해서도 $\rho(x)$는 정의되지 않는다.

- $\{x \mapsto v\}\rho$는 주어진 환경 ρ에서 변수 x가 값 v를 의미하도록 확장한 환경을 뜻한다. 예를 들어, $\rho = \{x \mapsto 1, y \mapsto 2\}$일 때, $\{x \mapsto 2\}\rho$는 환경 $\{x \mapsto 2, y \mapsto 2\}$를 의미한다. 수학적으로 $\{x \mapsto v\}\rho$는 다음과 같이 정의되는 함수이다.

$$(\{x \mapsto v\}\rho)(y) = \begin{cases} v & \text{if } x = y \\ \rho(y) & \text{if } x \neq y \end{cases}$$

  환경 $\{x \mapsto v\}\rho$에서 변수 y의 값은, $x = y$이면 v이고 $x \neq y$이면 확장하기 이전의 환경 ρ에 정의되어 있는 y 값임을 뜻한다. 예를 들어, $\rho = \{x \mapsto 1, y \mapsto 2\}$일 때 $(\{x \mapsto 2\}\rho)(y) = \rho(y) = 2$이다.

- 환경을 여러 변수에 대해서 확장할 때 $\{x_1 \mapsto v_1, x_2 \mapsto v_2\}\rho$와 같은 표기법을 사용한다. 환경 ρ를 x_1은 v_1, x_2는 v_2를 가지도록 확장한다는 뜻이다. 환경 ρ를 먼저 (x_2, v_2)에 대해서 확장하고, 그 결과를 (x_1, v_1)에 대해서 다시 한번 확장한 것이다. 다음과 같이 정의된다.

$$\{x_1 \mapsto v_1, x_2 \mapsto v_2\}\rho = \{x_1 \mapsto v_1\}(\{x_2 \mapsto v_2\}\rho)$$

  예를 들어, $\rho = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$일 때 $\{x \mapsto 2, y \mapsto 3\}\rho$은 환경 $\{x \mapsto 2, y \mapsto 3, z \mapsto 3\}$을 뜻한다.

3.2.2 추론 규칙

환경 ρ가 주어져 있을 때 식 E의 의미를 정의해보자. "환경 ρ에서 식 E를 계산한 값은 v이다"를 다음과 같이 표현하기로 하자.

$$\rho \vdash E \Rightarrow v$$

정수식의 경우 의미구조를 프로그램과 값 사이의 이항 관계($E \Rightarrow v$)로 정의하였지만, 이제 프로그램의 의미가 환경에 의존하게 되었으므로 의미구조가 환경, 프로그램, 값 사이의 삼항 관계(ternary relation)로 정의된다. 그러한 삼항 관계를 $\rho \vdash E \Rightarrow v$와 같이 표현하는 것이다. 예를 들어,

- $\emptyset \vdash$ `1` $\Rightarrow$ 1: 빈 환경에서 식 `1`을 계산한 결과는 1임을 뜻한다.
- $\{$`x` $\mapsto$ 1$\}$ $\vdash$ `x+1` $\Rightarrow$ 2: 환경 $\{$`x` $\mapsto$ 1$\}$에서 식 `x+1`을 계산한 값은 2임을 뜻한다.
- $\{$`x` $\mapsto$ 1$\}$ $\vdash$ `let y = 2 in x + y` $\Rightarrow$ 3: 환경 $\{$`x` $\mapsto$ 1$\}$에서 `let y = 2 in x + y`를 계산한 결과는 3임을 뜻한다.

의미구조를 정의하는 추론 규칙은 그림 3.2와 같다. 처음 다섯 개 규칙들은 정수식 언어의 의미구조가 환경을 포함하도록 단순히 확장한 것들이다. 변수와 관련하여 새롭게 추가된 규칙들은 E-Var와 E-Let이다. 규칙 E-Var는 주어진 환경 ρ에서 변수 x의 의미는 환경에 정의되어 있는 x의 값 $\rho(x)$임을 뜻한다. 규칙 E-Var는 변수 x가 현재 환경에 정의되어 있는 경우에만 적용 가능하다

$$\frac{}{\rho \vdash n \Rightarrow n} \quad \text{E-Num}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2} \quad \text{E-Plus}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ - } E_2 \Rightarrow n_1 - n_2} \quad \text{E-Minus}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ * } E_2 \Rightarrow n_1 * n_2} \quad \text{E-Mult}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ / } E_2 \Rightarrow n_1/n_2} \; n_2 \neq 0 \quad \text{E-Div}$$

$$\frac{}{\rho \vdash x \Rightarrow \rho(x)} \quad \text{E-Var}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \quad \{x \mapsto v_1\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v} \quad \text{E-Let}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{true} \qquad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v} \quad \text{E-If-T}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{false} \qquad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v} \quad \text{E-If-F}$$

$$\frac{\rho \vdash E \Rightarrow 0}{\rho \vdash \texttt{iszero } E \Rightarrow \mathit{true}} \quad \text{E-Zero-T}$$

$$\frac{\rho \vdash E \Rightarrow n}{\rho \vdash \texttt{iszero } E \Rightarrow \mathit{false}} \; n \neq 0 \quad \text{E-Zero-F}$$

그림 3.2: 의미구조

($x \in \mathrm{Dom}(\rho)$). 규칙 E-Let은 `let`식의 의미를 정의하고 있다. 환경 ρ에서 식 `let` x `=` E_1 `in` E_2는 다음과 같이 계산된다.

1. 먼저 식 E_1의 값을 주어진 환경 ρ에서 계산하여 값 v_1을 얻는다 ($\rho \vdash E_1 \Rightarrow v_1$).

2. 그 다음에 변수 x가 값 v_1을 의미하도록 현재 환경을 확장한다 ($\{x \mapsto v_1\}\rho$).

3. 확장된 환경 $\{x \mapsto v_1\}\rho$에서 식 E_2를 계산하여 값 v를 얻는다.

4. 전체식 `let` x `=` E_1 `in` E_2을 계산한 결과값은 v가 된다.

식 E_2를 계산하는 데 있어서 ρ를 확장한 환경($\{x \mapsto v_1\}\rho$)에서 계산하는 것이 핵심이다. 그 결과 E_2내에서 변수 x를 사용하면 값 v_1을 의미하게 될 뿐 아니라 E_1내에서 변수들이 새로 정의되더라도 E_2를 계산하는 데 있어서 영향을 주지 않게 된다.

조건식 `if` E_1 `then` E_2 `else` E_3의 의미는, E_1을 계산한 결과가 *true*이면 E_2를 계산하고 (E-If-T), *false*이면 E_3를 계산하는 것으로 정의되어 있다(E-If-F). `iszero` E의 값은 E가 0을 계산하면 *true*이고(E-Zero-T), E의 값이 0이 아니면 *false*이다(E-Zero-F).

위의 의미정의에서 n은 정수값을 의미하고 v는 임의의 값을 뜻하는 기호이다. 예를 들어, 사칙연산에 대한 규칙들에서 식 E_1과 E_2의 계산결과를 n_1과 n_2로 표현하고 있는데 이는 E_1과 E_2가 정수값을 계산해야 의미가 정의된다는 것을 뜻한다. 따라서 `1 + true`와 같은 식은 위의 의미구조에 의하면 의미를 가지지 않는 프로그램이 된다. 반면에 `let`식의 의미구조 정의에서는 E_1과 E_2의 계산결과를 각각 v_1과 v로 나타내고 있는데, 임의의 타입의 값이 계산

될 수 있다는 뜻이다. 앞으로도 정수값은 n, 참/거짓값은 b, 임의의 값은 v와 같이 값이 가지는 타입을 뜻하는 기호를 사용하여 의미구조를 정의할 것이다.

지금까지 정의한 의미구조대로 앞의 예제 프로그램들을 실행시켜보자. 경우에 따라 추론 규칙을 적용할 때 공리에 해당하는 규칙들을 생략하거나 간단히 표기하는 등 주요 과정만 나타내기로 하자.

예제 1 빈 환경에서 `let x = 1 in x + 2`은 다음과 같이 실행된다.

$$\dfrac{\dfrac{}{\emptyset \vdash \texttt{1} \Rightarrow 1} \quad \dfrac{\dfrac{}{\{\texttt{x} \mapsto 1\} \vdash \texttt{x} \Rightarrow 1} \quad \dfrac{}{\{\texttt{x} \mapsto 1\} \vdash \texttt{2} \Rightarrow 2}\ \text{E-Plus}}{\{\texttt{x} \mapsto 1\} \vdash \texttt{x + 2} \Rightarrow 3}\ \text{E-Let}}{\emptyset \vdash \texttt{let x = 1 in x + 2} \Rightarrow 3}$$

주어진 프로그램은 `let`식이므로 가장 먼저 실행규칙 E-Let을 적용한다. 규칙 E-Let에 의하여 현재 환경 $\emptyset$에서 식 `1`을 계산하여 값 1을 얻고(E-Num), 현재 환경을 변수 `x`에 대하여 확장($\{\texttt{x} \mapsto 1\}$)한 후 `x + 2`를 계산한다. `x + 2`를 계산하기 위하여 실행규칙 E-Plus를 적용하고 이는 다시 E-Var와 E-Num 규칙을 적용하여 실행된다.

예제 2 빈 환경에서 프로그램 `let x = 1 in let y = 2 in x + y`의 의미는 다음과 같이 계산된다.

$$\dfrac{\emptyset \vdash \texttt{1} \Rightarrow 1 \qquad \dfrac{\{\texttt{x} \mapsto 1\} \vdash \texttt{2} \Rightarrow 2 \quad \dfrac{\{\texttt{y} \mapsto 2, \texttt{x} \mapsto 1\} \vdash \texttt{x} \Rightarrow 1 \quad \{\texttt{y} \mapsto 2, \texttt{x} \mapsto 1\} \vdash \texttt{y} \Rightarrow 2}{\{\texttt{y} \mapsto 2, \texttt{x} \mapsto 1\} \vdash \texttt{x + y} \Rightarrow 3}}{\{\texttt{x} \mapsto 1\} \vdash \texttt{let y = 2 in x + y} \Rightarrow 3}}{\emptyset \vdash \texttt{let x = 1 in let y = 2 in x + y} \Rightarrow 3}$$

예제 3 빈 환경에서 프로그램

```
let x = let y = 2
        in y + 1
in x + 3
```

는 다음과 같이 계산된다.

$$\dfrac{\dfrac{\emptyset \vdash \texttt{2} \Rightarrow 2 \quad \dfrac{\{\texttt{y} \mapsto 2\} \vdash \texttt{y} \Rightarrow 2 \quad \{\texttt{y} \mapsto 2\} \vdash \texttt{1} \Rightarrow 1}{\{\texttt{y} \mapsto 2\} \vdash \texttt{y + 1} \Rightarrow 3}}{\emptyset \vdash \texttt{let y = 2 in y + 1} \Rightarrow 3} \quad \dfrac{\{\texttt{x} \mapsto 3\} \vdash \texttt{x} \Rightarrow 3 \quad \{\texttt{x} \mapsto 3\} \vdash \texttt{3} \Rightarrow 3}{\{\texttt{x} \mapsto 3\} \vdash \texttt{x + 3} \Rightarrow 6}}{\emptyset \vdash \texttt{let x = (let y = 2 in y + 1) in x + 3} \Rightarrow 6}$$

식 `x + 3`을 계산할 때의 환경($\{\texttt{x} \mapsto 3\}$)에는 변수 `y`에 대한 정보가 없음을 눈여겨보자. 따라서, 마지막 줄에서 `x + 3` 대신에 `x + y`를 계산하는 프로그램이었다면 실행의미가 정의되지 않는다. 의미구조 정의에 따라 자연스럽게 변수 `y`는 `x`의 값을 정의하는 `let`식 내에서만 유효하게 된다.

예제 4 빈 환경에서 프로그램

```
let x = 1
in let y = 2
   in let x = 3
      in x + y
```

는 다음과 같이 계산된다.

$$\dfrac{\emptyset \vdash \texttt{1} \Rightarrow 1 \quad \dfrac{\{\texttt{x} \mapsto 1\} \vdash \texttt{2} \Rightarrow 2 \quad A}{\{\texttt{x} \mapsto 1\} \vdash \texttt{let y = 2 in let x = 3 in x + y} \Rightarrow 5}}{\emptyset \vdash \texttt{let x = 1 in let y = 2 in let x = 3 in x + y} \Rightarrow 5}$$

위에서 생략한 부분(A)은 아래와 같다.

$$\dfrac{\{\texttt{y} \mapsto 2, \texttt{x} \mapsto 1\} \vdash \texttt{3} \Rightarrow 3 \quad \dfrac{\{\texttt{y} \mapsto 2, \texttt{x} \mapsto 3\} \vdash \texttt{x} \Rightarrow 3 \qquad \{\texttt{y} \mapsto 2, \texttt{x} \mapsto 3\} \vdash \texttt{y} \Rightarrow 2}{\{\texttt{y} \mapsto 2, \texttt{x} \mapsto 3\} \vdash \texttt{x + y} \Rightarrow 5}}{\{\texttt{y} \mapsto 2, \texttt{x} \mapsto 1\} \vdash \texttt{let x = 3 in x + y} \Rightarrow 5}$$

위 과정에서 `let`을 통해 변수가 정의될 때마다 환경이 변화하는 모습을 파악하는 것이 중요하다.

예제 5 빈 환경에서 프로그램

```
let x = 1
in let y = let x = 2
           in x + x
   in x + y
```

는 다음과 같이 계산된다.

$$\dfrac{\dfrac{\dfrac{\{\mathtt{x}\mapsto 2\}\vdash \mathtt{x}\Rightarrow 2}{\{\mathtt{x}\mapsto 2\}\vdash \mathtt{x}\Rightarrow 2}\quad \dfrac{\{\mathtt{y}\mapsto 4,\mathtt{x}\mapsto 1\}\vdash \mathtt{x}\Rightarrow 1}{\{\mathtt{y}\mapsto 4,\mathtt{x}\mapsto 1\}\vdash \mathtt{y}\Rightarrow 4}}{\{\mathtt{x}\mapsto 2\}\vdash \mathtt{x}+\mathtt{x}\Rightarrow 4}}{\{\mathtt{x}\mapsto 1\}\vdash \mathtt{let\ x=2\ in\ x+x}\Rightarrow 4}\quad \dfrac{}{\{\mathtt{y}\mapsto 4,\mathtt{x}\mapsto 1\}\vdash \mathtt{x}+\mathtt{y}\Rightarrow 5}}{\dfrac{\{\mathtt{x}\mapsto 1\}\vdash \mathtt{let\ y=(let\ x=2\ in\ x+x)\ in\ x+y}\Rightarrow 5}{\emptyset\vdash \mathtt{let\ x=1\ in\ let\ y=(let\ x=2\ in\ x+x)\ in\ x+y}\ \Rightarrow 5}}$$

식 `let x = 2 in x + x`를 계산할 때 생성된 환경 $\{\mathtt{x}\mapsto 2\}$가 `x + x`를 계산할 때에만 사용될 뿐 마지막 식 `x + y`를 계산하는 시점까지 전파되지 않음을 이해하는것이 중요하다. 그 결과 `x + y`에서의 `x`는 첫째줄에서 정의된 `x`를 가리키게 된다.

예제 6 환경 $\rho=\{\mathtt{x}\mapsto 1,\mathtt{y}\mapsto 2\}$에서 프로그램

```
if iszero (x - 1) then y - 1 else y + 1
```

의 실행 과정은 다음과 같다.

$$\dfrac{\dfrac{\dfrac{\rho\vdash \mathtt{x}\Rightarrow 1\quad \rho\vdash 1\Rightarrow 1}{\rho\vdash \mathtt{x}-1\Rightarrow 0}}{\rho\vdash \mathtt{iszero\ (x-1)}\Rightarrow \mathit{true}}\quad \dfrac{\rho\vdash \mathtt{y}\Rightarrow 2\quad \rho\vdash 1\Rightarrow 1}{\rho\vdash \mathtt{y}-1\Rightarrow 1}}{\rho\vdash \mathtt{if\ iszero\ (x-1)\ then\ y-1\ else\ y+1}\Rightarrow 1}$$

3.3 구현

지금까지 설계한 언어의 실행기를 구현해보자. 먼저 언어의 문법 구조를 다음과 같이 OCaml의 데이터 타입으로 정의할 수 있다(편의를 위해 사칙 연산 가운데 덧셈과 뺄셈만 고려하였다).

```
type exp =
  | CONST of int
  | ADD of exp * exp
  | SUB of exp * exp
  | VAR of var
  | LET of var * exp * exp
  | IF of exp * exp * exp
  | ISZERO of exp
and var = string
```

예를 들어, 다음의 프로그램

```
let x = 1
in let y = 2
   in let  y = let x = x + 1
               in x + y
      in x - y
```

은 아래와 같이 표현된다.

```
LET ("x", CONST 1,
   LET ("y", CONST 2,
      LET ("y", LET ("x", ADD(VAR "x", CONST 1),
                  ADD (VAR "x", VAR "y")),
        SUB (VAR "x", VAR "y"))))
```

프로그램의 의미구조를 정의하기 위해서 의미공간 정의가 필요하다. 먼저 값은 정수 또는 부울값이므로 다음과 같이 정의하자.

```
type value = Int of int | Bool of bool
```

예를 들어, 정수값 1은 `Int 1`, 부울값 $true$는 `Bool true`와 같이 표현된다. 환경은 다음과 같이 변수와 값의 리스트로 구현할 수 있다.

```
type env = (var * value) list
let empty_env = []
let extend_env (x,v) e = (x,v)::e
let rec apply_env x e =
  match e with
  | [] -> raise (Failure (x ^ " is not found"))
  | (y,v)::tl -> if x = y then v else apply_env x tl
```

빈 환경을 빈 리스트로 정의하였고 환경을 확장하는 함수 `extend_env`와 환경에서 변수의 값을 가져오는 함수 `apply_env`를 정의하였다.

추론 규칙으로 정의한 의미구조는 아래의 재귀 함수 `eval`로 정의할 수 있다.

```
let rec eval : exp -> env -> value
=fun exp env ->
  match exp with
  | CONST n -> Int n
  | VAR x -> apply_env x env
  | ADD (e1,e2) ->
    let v1 = eval e1 env in
    let v2 = eval e2 env in
      (match v1,v2 with
      | Int n1, Int n2 -> Int (n1 + n2)
      | _ -> raise (Failure "Type Error"))
  | SUB (e1,e2) ->
    let v1 = eval e1 env in
    let v2 = eval e2 env in
      (match v1,v2 with
```

```
      | Int n1, Int n2 -> Int (n1 - n2)
      | _ -> raise (Failure "Type Error"))
  | ISZERO e ->
    (match eval e env with
    | Int n when n = 0 -> Bool true
    | Int n when n <> 0 -> Bool false
    | _ -> raise (Failure "Type Error"))
  | IF (e1,e2,e3) ->
    (match eval e1 env with
    | Bool true -> eval e2 env
    | Bool false -> eval e3 env
    | _ -> raise (Failure "Type Error"))
  | LET (x,e1,e2) ->
    let v1 = eval e1 env in
      eval e2 (extend_env (x,v1) env)
```

추론 규칙을 그대로 재귀함수의 형태로 구현한 것이다. 추론 규칙 정의에서는 암묵적으로 표현했던, 실행중에 타입 오류가 발생할 수 있는 경우들에 대해서 예외처리를 명시적으로 하였다.

프로그램을 실행시키는 함수를 다음과 같이 정의할 수 있다. 주어진 프로그램과 빈 환경을 가지고 eval을 호출한다.

```
let run : exp -> value
=fun pgm -> eval pgm empty_env
```

예를 들어, 다음과 같이 프로그램을 실행시켜 볼 수 있다.

```
# let e1 = LET ("x", CONST 1, ADD (VAR "x", CONST 2));;
val e1 : exp = LET ("x", CONST 1, ADD (VAR "x", CONST 2))
# run e1;;
- : value = Int 3
# let e2 = LET ("x", CONST 1,
   LET ("y", CONST 2,
```

```
      LET ("y", LET ("x", ADD(VAR "x", CONST 1),
                  ADD (VAR "x", VAR "y")),
        SUB (VAR "x", VAR "y"))));;
val e2 : exp =
  LET ("x", CONST 1,
   LET ("y", CONST 2,
    LET ("y", LET ("x", ADD (VAR "x", CONST 1),
                  ADD (VAR "x", VAR "y")),
     SUB (VAR "x", VAR "y"))))
# run e2;;
- : value = Int (-3)
```

4 함수 정의와 호출

이 장에서는 앞장의 언어를 확장하여 함수를 정의하고 사용할 수 있도록 하자.

4.1 문법구조

프로그램에서 함수를 사용하려면 함수를 생성하는 방법과 호출하는 방법 두 가지를 지원해야 한다. 이를 위해 언어의 문법구조를 그림 4.1과 같이 확장하자. 기존 언어에 마지막 두 경우(함수 생성식, 함수 호출식)을 추가하였다. `fun` x E는 x를 인자로 받아서 E의 계산결과를 반환하는 함수를 정의하는 구문이다. 이 때, x를 형식 인자(formal parameter), E를 함수의 몸통식(body expression)이라고 한다. 인자 x의 값은 몸통 E에서만 사용할 수 있다. 함수 호출은 일반적으로 E_1 E_2의 형태를 가진다. E_1은 호출할 함수를 계산하는 식이고 E_2는 함수의 인자를 계산하는 식이다. E_2의 값을 실제 인자(actual parameter)라고 부른다.

이 언어로 작성할 수 있는 프로그램의 예는 다음과 같다.

- `fun x (x+1)`

  인자 `x`를 받아서 `x+1`의 값을 반환하는 함수를 뜻한다.

$$
\begin{array}{rcll}
E & \rightarrow & n & \\
 & | & x & \\
 & | & E_1 \texttt{ + } E_2 & \\
 & | & E_1 \texttt{ - } E_2 & \\
 & | & E_1 \texttt{ * } E_2 & \\
 & | & E_1 \texttt{ / } E_2 & \\
 & | & \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 & \\
 & | & \texttt{iszero } E & \\
 & | & \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 & \\
 & | & \texttt{fun } x\ E & \text{함수 생성식} \\
 & | & E_1\ E_2 & \text{함수 호출식}
\end{array}
$$

그림 4.1: 문법구조

- `fun x (let y = x+1 in if iszero y then x else y)`

  함수 몸통에는 임의의 식이 올 수 있다. 이 경우 `let`식을 이용하여 `x+1`을 `y`라고 한 후 조건식의 결과를 반환하는 함수를 뜻한다.

- `fun x (fun y (x+y))`

  함수 몸통에 다른 함수 생성식이 쓰인 경우이다. 인자 `x`를 받아서 함수 `fun y (x+y)`를 반환하는 함수를 뜻한다.

- `fun f (f 1)`

  함수 인자로 다른 함수를 받는 경우이다. 인자 `f`로 함수를 받고 전달된 함수를 인자 1을 가지고 호출한 결과를 반환하는 함수를 뜻한다.

- `let f = fun x (x+1) in (f 2)`

  다른값과 마찬가지로 함수에도 이름을 붙일 수 있다. 함수

`fun x (x+1)`를 `f`라고 이름지은 후 `f`를 실제 인자 2를 가지고 호출하는 프로그램이다. 결과값은 3이다.

- `let f = fun x (x+1) in (f (f 2))`

  함수 호출식(E_1 E_2)에서 E_1과 E_2는 임의의 식이 될 수 있다. 이 예제에서 함수 호출식 `(f (f 2))`는 E_2가 또 다른 함수 호출식 `(f 2)`인 경우이다. `(f 2)`를 먼저 계산하고, 그 결과를 인자로 함수 `f`를 다시 한번 더 호출하고 있으므로 최종 결과값은 4이다.

- `(fun f (f (f 2)) fun x (x+1))`

  이 예제에서는 함수 호출식의 E_1 자리에 이름 대신 함수 생성식 `fun f (f (f 2))`을 바로 사용했다. E_2의 자리에도 함수 생성식 `fun x (x+1)`이 쓰였는데, 이 함수가 인자 `f`로 전달되고 두 번 호출되어 계산결과는 4가 된다.

- `let f = fun x (fun y (x+y)) in ((f 3) 4)`

  이 예제에서 함수 `f`는 인자 `x`, `y`를 받아서 `x+y`를 반환하는 함수라고 해석할 수 있다. `((f 3) 4)`에서 `x`, `y`에 각각 3, 4를 전달하여 `f`를 호출하였고 결과값은 7이 된다.

위의 예제에서 볼 수 있듯이, 그림 4.1의 언어에서는 함수를 매우 자유롭게 사용할 수 있다. 함수를 변수에 저장할 수 있고, 다른 함수의 인자로 넘길 수 있고, 다른 함수의 결과값으로 반환할 수도 있다. OCaml과 같이 고차 함수를 지원하는 언어이다.

참고로 프로그래밍 언어가 함수를 지원하면 let식은 필수가 아니게 된다. 임의의 let식을 동일한 의미를 가지는 함수 호출식으로 변환할 수 있기 때문이다. 임의의 E_1, E_2에 대하여 다음의 동치관계를 이용하면 된다.

$$\texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \;\equiv\; (\texttt{fun } x\ E_2)\ E_1$$

예를 들어 프로그램

```
let f = fun x (x+1)
in (f (f 2))
```

은 함수 호출을 이용한 아래 프로그램과 동일한 의미를 가진다.

```
(fun f (f (f 2)) fun x (x+1))
```

let식과 같이 다른 구문으로 표현 가능하지만 프로그램 작성의 편의를 위해 프로그래밍 언어에서 제공하는 구문을 설탕구조(syntactic sugar)라고 한다.

OCaml이 제공하는 함수 정의 구문도 설탕구조이다. 예를 들어, 아래 OCaml 프로그램을 보자.

```
let square x = x * x in
  let add x y = x + y in
    (add 1 (square 2))
```

다음과 같이 그림 4.1의 문법구조만으로 표현할 수 있다.

```
let square = fun x (x * x) in
  let add = fun x (fun y (x + y)) in
    (add 1 (square 2))
```

4.2 의미구조

이제 함수 생성식과 호출식의 실행 의미를 정의해보자.

자유 변수 함수의 의미를 정의하기 위해서는 먼저 프로그램에 등장하는 변수들을 자유 변수와 묶인 변수로 구분하는 것이 필요하다. 자유 변수(free variable)란 주어진 식에서 그 정의를 찾을 수 없는 변수를 말한다. 반면에 주어진 식에서 정의를 찾을 수 있는 변수를 묶인 변수(bound variable)라고 한다. 식 E에 등장하는 자유 변수들의 집합 $FV(E)$는 다음과 같이 귀납적으로 정의된다(식 E에 등장하는 모든 변수들의 집합을 $VAR(E)$라고 할 때 묶인 변수들의 집합은 $VAR(E) \setminus FV(E)$가 된다).

$$
\begin{aligned}
FV(n) &= \emptyset \\
FV(x) &= \{x\} \\
FV(E_1 \texttt{ + } E_2) &= FV(E_1) \cup FV(E_2) \\
FV(\texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2) &= FV(E_1) \cup (FV(E_2) \setminus \{x\}) \\
FV(\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3) &= FV(E_1) \cup FV(E_2) \cup FV(E_3) \\
FV(\texttt{fun } x \ E) &= FV(E) \setminus \{x\} \\
FV(E_1 \ E_2) &= FV(E_1) \cup FV(E_2)
\end{aligned}
$$

주어진 식이 하나의 변수(x)이면 해당 변수는 식 x내에서 자유 변수이다. 식 E_1 + E_2내의 자유 변수는 식 E_1과 E_2내의 자유 변수들의 모음이다. 식 let x = E_1 in E_2의 자유 변수를 구할 때에도 E_1과 E_2의 자유 변수들을 모은다. 단, 변수 x가 정의되었으므로 E_2

의 자유 변수에서 x는 제외해야 한다. 식 `fun` x E 내에서도 x는 함수의 인자로 정의되었으므로 자유 변수가 아니다. 몇 가지 예를 들어 보자.

- $FV($`fun y (x+y)`$) = \{$`x`$\}$
- $FV($`fun x (let y = 1 in x+y+z)`$) = \{$`z`$\}$
- $FV($`fun x (fun y (x+y))`$) = \emptyset$
- $FV($`let x = y in fun y (x+y+z)`$) = \{$`y`$, $`z`$\}$
- $FV($`let x = x in x`$) = \{x\}$

정적/동적 유효범위 함수의 의미는 함수에 등장하는 자유 변수의 값을 해석하는 방식에 따라 달라진다. 예를 들어, 아래 프로그램을 생각해보자.

```
let x = 1
in let f = fun y (x+y)
   in let x = 2
      in let g = fun y (x+y)
         in (f 1) + (g 1)
```

위 프로그램의 의미는 자유 변수의 값을 결정하는 방식에 따라 두 가지로 해석할 수 있다.

- 함수식 `fun y (x+y)`에 등장하는 자유 변수(`x`)의 값을 그 함수가 정의되는 시점에서의 값으로 해석한다면 위 프로그램의 실행 결과는 5가 된다. 함수 `f`가 정의되는 시점에서의 `x`

의 값은 1이므로 f는 y를 인자로 받아서 1을 증가시키는 함수로 해석하는 것이다. 따라서 함수 호출식 (f 1)의 결과는 2가 된다. 반면에 함수 g가 정의되는 시점에서의 x 값은 2이므로 g는 인자의 값을 2 증가시키는 함수를 뜻한다. 따라서 (g 1)의 호출 결과는 3이다.

- 함수에 등장하는 자유 변수의 값을 그 함수가 호출되는 시점에서의 값으로 결정한다면 위 프로그램의 실행 결과는 6이 된다. 마지막 식 (f 1) + (g 1)이 계산되는 시점에서 x값은 2이기 때문에 함수 f와 g 모두 인자값을 2 증가시키는 함수로 해석하는 것이다. 따라서 두 함수 호출식 (f 1)과 (g 1)의 결과는 모두 3이다.

프로그래밍 언어가 첫 번째 방식으로 함수 호출을 처리한다면 정적 유효범위(static scoping)를 지원하는 것이다. 두 번째 방식은 동적 유효범위(dynamic scoping)라고 부른다.[1)]

대부분의 프로그래밍 언어들은 정적 유효범위를 지원하고 있다. 이 방식에서는 변수의 유효범위가 프로그램 실행 전에 정적으로 결정되어 프로그램을 이해하기가 훨씬 쉬워지기 때문이다. 예를 들어, 위의 예제 프로그램을 실행하는 경우 변수 x의 유효범위는 그림 4.2와 같이 프로그램의 실행과 무관하게 실행 전에 정해진다. 바깥 사각형은 첫째줄에서 정의된 x의 유효범위를 뜻한다. 즉, 이 영역에 등장하는 자유 변수 x는 첫째줄에서 정의된 x, 즉 1을 의

1) 일반적으로 프로그래밍 언어에서 '정적(static)'은 '실행 전에 결정되는'을 의미한다. '동적(dynamic)'은 '실행 후에 결정되는'을 뜻한다.

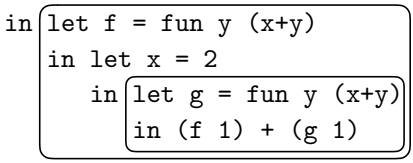


그림 4.2: 변수 x의 정적 유효범위

미한다. 안쪽 사각형은 셋째줄에서 정의된 x의 유효범위이다. 즉, 이 영역에서 쓰이는 x는 첫째줄이 아닌 셋째줄의 x, 즉 2를 가리킨다. 다른 변수들의 유효범위도 이와 같이 프로그램의 생김새에 따라 실행 전에 결정된다. 모든 변수의 유효범위를 프로그램을 실행시키지 않고 알 수 있으므로 프로그램을 해석하기 쉽다.

반면에 동적 유효범위를 사용하여 프로그램을 실행하면 변수의 유효범위가 실행 시점에 정해진다. 예를 들어, 이 방식으로 프로그램을 실행할 때 셋째줄에서 정의된 변수 x는 둘째 줄에 있는 식 x+y에서도 유효하게 되는데 실행 전에 이 사실을 알아내는 것은 일반적으로 불가능(undecidable)하다. 변수의 유효범위를 실행 전에 예측하기 어렵고, 프로그램을 실행시켜 보아야만 알 수 있기 때문에 프로그램을 해석하기 어려워진다.

4.2.1 정적 유효범위

정적 유효범위를 지원하도록 함수의 의미를 정의해보자. 먼저 프로그램에서 함수를 값으로 사용할 수 있게 되었으므로 다음과 같이

의미공간을 확장하자.

$$\begin{aligned} \mathit{Val} &= \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} \\ \mathit{Procedure} &= \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \\ \mathit{Env} &= \mathit{Var} \to \mathit{Val} \end{aligned}$$

이제 프로그램에서 다룰 수 있는 값은 정수와 부울값 외에도 함수값($\mathit{Procedure}$)이 추가되었다. 함수값 $(x, E, \rho) \in \mathit{Procedure}$은 세 원소로 구성된다. x는 함수의 형식 인자, E는 함수의 몸통식, ρ는 함수가 정의되는 시점에서의 환경이다. 마지막 원소인 환경 ρ는 함수의 자유 변수를 함수가 정의되는 시점에서의 값으로 해석하기 위해서 필요하다. 이렇게 정의한 함수값을 클로저(closure)[2]라고도 부른다.

이제 함수 생성과 호출식의 실행 의미를 정의하자. 함수 생성식의 의미는 다음과 같다.

$$\frac{}{\rho \vdash \texttt{fun}\ x\ E \Rightarrow (x, E, \rho)}$$

환경 ρ가 주어졌을때 식 `fun` x E를 계산하면 함수값 (x, E, ρ)가 생성된다는 뜻이다. 정적 유효범위를 위하여 함수가 정의되는 시점에 주어진 환경 ρ를 함수값의 마지막 원소로 저장하고 있다. 예를 들어 빈 환경($\emptyset$)에서 함수 생성식 `fun x x`를 계산하면 함수값 $(\texttt{x}, \texttt{x}, \emptyset)$가 만들어진다.

2) 함수를 호출하기 위해 필요한 모든 정보를 담고 있는 값이라는 뜻이다.

함수 호출식의 의미는 다음과 같다.

$$\frac{\rho \vdash E_1 \Rightarrow (x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad \{x \mapsto v\}\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

함수 호출 $E_1\ E_2$를 실행하기 위해서 먼저 E_1과 E_2를 각각 현재 환경 ρ에서 계산한다. E_1을 계산하면 함수값 (x, E, ρ')이 나와야 한다. 만약 함수값이 아닌 다른 값, 예를 들어 정수값이 E_1의 결과로 나온다면 함수 호출이 불가능하고 프로그램의 의미가 정의되지 않는다(타입 오류가 발생한다). 함수값 (x, E, ρ')에서 ρ'는 해당 함수가 정의될 시점에서 저장해 놓은 환경이다. 함수 호출이 일어나는 시점에서의 환경인 ρ와 다를 수 있기 때문에 ρ'로 표시하였다. 그 다음 E_2를 계산하여 함수의 인자로 넘길 값을 계산한다. 그 값을 v라고 하자(여기서 v는 E_2의 값이 정수, 참/거짓, 함수 중 임의의 값이 될 수 있음을 나타낸다). E_1과 E_2를 계산했다면 마지막으로 해야 할 일은 함수의 몸통인 E를 계산하는 것이다. 이때 함수가 정의되는 시점에서의 환경인 ρ'을 이용하는 것이 정적 유효범위의 핵심이다. ρ'에서 형식 인자 x가 실제 인자 v를 가지도록 확장한 후 몸통식 E를 계산하면 값 v'이 얻어지고 이 값이 함수 호출식 $E_1\ E_2$의 결과값이 된다.

몇 가지 실행 예를 살펴보자.

예제 1 환경 $\rho = \{\texttt{y} \mapsto 2\}$에서 함수 호출식 `(fun x (x+y)) 1`를 계산하는 과정은 다음과 같다.

$$\dfrac{\rho \vdash \texttt{fun x (x+y)} \Rightarrow (\texttt{x}, \texttt{x+y}, \rho) \quad \rho \vdash 1 \Rightarrow 1 \quad \dfrac{\dfrac{}{\{\texttt{x} \mapsto 1\}\rho \vdash \texttt{x} \Rightarrow 1} \quad \dfrac{}{\{\texttt{x} \mapsto 1\}\rho \vdash \texttt{y} \Rightarrow 2}}{\{\texttt{x} \mapsto 1\}\rho \vdash \texttt{x+y} \Rightarrow 3}}{\rho \vdash \texttt{(fun x (x+y)) 1} \Rightarrow 3}$$

예제 2 빈 환경에서 프로그램 아래 프로그램이 실행되는 과정을 살펴보자.

```
let x = 1
in let f = fun y (x+y)
    in let x = 2
        in (f 3)
```

1. `let x = 1 in ...`을 실행하면 다음과 같은 환경이 만들어진다:

$$\rho_1 = \{\texttt{x} \mapsto 1\}$$

2. 환경 ρ_1에서 `let f = fun y (x+y) in ...`을 실행하면 `f`가 함수값 $(\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})$을 가지도록 현재 환경 ρ_1이 확장된다. 확장된 환경을 ρ_2라고 하자.

$$\rho_2 = \{\texttt{x} \mapsto 1, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})\}$$

3. 환경 ρ_2에서 `let x = 2 in ...`을 실행하면 `x`는 이제 정수 2를 뜻하게 된다:

$$\rho_3 = \{\texttt{x} \mapsto 2, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})\}$$

`f`가 의미하는 함수값 안에 존재하는 환경에서 `x`는 여전히 1을 의미함을 눈여겨보자.

4. 이제 환경 ρ_3을 가지고 함수 호출식 `(f 3)`을 계산한다. 의미구조 규칙에 의해 다음 단계를 거쳐서 실행된다.

   a) 먼저 `f`가 현재 환경 ρ_3에서 의미하는 값을 계산한다. 함수값 $(\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})$을 얻는다.

   b) 실제 인자 `3`을 계산하여 정수값 3을 얻는다.

   c) 함수값 $(\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})$으로부터 함수가 정의된 시점에서의 환경 $\{\texttt{x} \mapsto 1\}$을 가져오고 전달된 인자를 포함하도록 확장한다. 이를 ρ_4라고 하자.

$$\rho_4 = \{\texttt{x} \mapsto 1, \texttt{y} \mapsto 3\}$$

   d) 환경 ρ_4에서 함수의 몸통 `x+y`를 계산하여 4를 얻는다.

위의 전체 과정을 추론 규칙을 이용한 증명 나무로 표현하면 다음과 같다.

$$\dfrac{\emptyset \vdash \texttt{1} \Rightarrow 1 \qquad \dfrac{\{\texttt{x} \mapsto 1\} \vdash \texttt{fun y (x+y)} \Rightarrow (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\}) \quad \vdots}{\{\texttt{x} \mapsto 1\} \vdash \begin{array}{l}\texttt{let f = fun y (x+y)} \\ \texttt{in let x = 2} \\ \texttt{in (f 3)}\end{array} \Rightarrow 4}}{\emptyset \vdash \begin{array}{l}\texttt{let x = 1} \\ \texttt{in let f = fun y (x+y)} \\ \texttt{in let x = 2} \\ \texttt{in (f 3)}\end{array} \Rightarrow 4}$$

위 과정에서 생략된 부분($\vdots$)은 다음과 같다.

$$\dfrac{\{\ \texttt{x} \mapsto 1, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})\ \} \vdash \texttt{2} \Rightarrow 2 \quad \vdots}{\{\ \texttt{x} \mapsto 1, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})\ \} \vdash \texttt{let x = 2 in (f 3)} \Rightarrow 4}$$

위 과정에서 생략한 부분($\vdots$)는 다음과 같다.

$$\dfrac{\begin{array}{l}\left\{\begin{array}{l}\texttt{x} \mapsto 2, \\ \texttt{f} \mapsto (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})\end{array}\right\} \vdash \texttt{f} \Rightarrow (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\}) \\ \left\{\begin{array}{l}\texttt{x} \mapsto 2, \\ \texttt{f} \mapsto (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})\end{array}\right\} \vdash \texttt{3} \Rightarrow 3 \\ \left\{\begin{array}{l}\texttt{x} \mapsto 1 \\ \texttt{y} \mapsto 3\end{array}\right\} \vdash \texttt{x+y} \Rightarrow 4\end{array}}{\left\{\begin{array}{l}\texttt{x} \mapsto 2, \\ \texttt{f} \mapsto (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})\end{array}\right\} \vdash \texttt{(f 3)} \Rightarrow 4}$$

4.2.2 동적 유효범위

동적 유효범위를 지원하는 것은 정적 유효범위보다 간단하다. 함수 생성시점의 환경을 쓰지 않고 함수가 호출될 시점에 주어진 환경을 그대로 이용하면 된다. 앞에서 정의한 정적 유효범위를 지원하는 함수 호출식의 의미를 아래와 같이 바꾸면 된다.

$$\frac{\rho \vdash E_1 \Rightarrow (x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad \{x \mapsto v\}\rho \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

정적 유효범위를 지원하는 규칙과 달리 함수 몸통을 계산할 때 ρ' 대신 함수 호출 시 주어진 환경인 ρ를 이용하도록 하였다.

이렇게 함수 호출식의 의미만 바꾸어도 되지만 불필요하게 함수 생성 시점의 환경을 함수값에 저장하게 된다. 다음과 같이 의미 정의를 단순화하자. 먼저 함수값에서 환경을 없애서 다음과 같이 새로 정의하자.

$$Procedure = Var \times Exp$$

동적 유효범위에서는 함수 호출 시점에 주어진 환경을 함수 몸통 계산에 사용할 것이므로 함수가 생성될 때의 환경을 더 이상 가지고 다니지 않아도 된다. 함수 생성과 호출식의 의미구조는 다음과 같이 단순해진다.

$$\frac{}{\rho \vdash \texttt{fun}\ x\ E \Rightarrow (x, E)}$$

$$\frac{\rho \vdash E_1 \Rightarrow (x, E) \quad \rho \vdash E_2 \Rightarrow v \quad \{x \mapsto v\}\rho \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

이와 같이 의미구조 정의 관점에서 동적 유효범위는 정적 유효범위보다 간단하다. 하지만 프로그램의 실행 의미를 파악하는 것은 더욱 복잡해졌다.

예를 들어 앞의 예제 프로그램은 다음과 같이 실행된다.

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \texttt{1} \Rightarrow 1} \quad \dfrac{\dfrac{\{\texttt{x} \mapsto 1\} \vdash \texttt{fun y (x+y)} \Rightarrow (\texttt{y}, \texttt{x+y}) \quad \vdots}{\{\texttt{x} \mapsto 1\} \vdash \begin{array}{l}\texttt{let f = fun y (x+y)} \\ \texttt{in let x = 2} \\ \texttt{in (f 3)}\end{array} \Rightarrow 5}}{\emptyset \vdash \begin{array}{l}\texttt{let x = 1} \\ \texttt{in let f = fun y (x+y)} \\ \texttt{in let x = 2} \\ \texttt{in (f 3)}\end{array} \Rightarrow 5}$$

위 과정에서 생략한 부분(⋮)은 다음과 같다.

$$\dfrac{\{\ \texttt{x} \mapsto 1, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y})\ \} \vdash \texttt{2} \Rightarrow 2 \quad \vdots}{\{\ \texttt{x} \mapsto 1, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y})\ \} \vdash \begin{array}{l}\texttt{let x = 2} \\ \texttt{in (f 3)}\end{array} \Rightarrow 5}$$

위 과정에서 생략한 부분(⋮)은 다음과 같다.

$$\dfrac{\begin{array}{c}\{\ \mathtt{x} \mapsto 2, \mathtt{f} \mapsto (\mathtt{y}, \mathtt{x+y})\ \} \vdash \mathtt{f} \Rightarrow (\mathtt{y}, \mathtt{x+y}) \\ \{\ \mathtt{x} \mapsto 2, \mathtt{f} \mapsto (\mathtt{y}, \mathtt{x+y})\ \} \vdash \mathtt{3} \Rightarrow 3 \\ \{\ \mathtt{x} \mapsto 2, \mathtt{f} \mapsto (\mathtt{y}, \mathtt{x+y}), \mathtt{y} \mapsto 3\ \} \vdash \mathtt{x+y} \Rightarrow 5\end{array}}{\left\{\ \mathtt{x} \mapsto 2, \mathtt{f} \mapsto (\mathtt{y}, \mathtt{x+y})\ \right\} \vdash \mathtt{(f\ 3)} \Rightarrow 5}$$

함수 호출식 `(f 3)`을 계산할 때의 환경이 몸통식 `x+y`를 계산하는 시점까지 전파되어 사용되는 점이 정적 유효범위에 따른 실행 과정과 다르다.

4.2.3 재귀 함수

아래와 같은 프로그램을 생각해보자.

```
let f = fun x (f x) in (f 1)
```

재귀 함수를 의도한 것이지만 앞에서 정의한 정적 유효범위를 지원하는 의미구조에 의하면 제대로 돌지 않는 프로그램이다. `let`식으로 `f`를 정의한 직후의 환경은 다음과 같다:

$$\{\mathtt{f} \mapsto (\mathtt{x}, \mathtt{(f\ x)}, \emptyset)\}$$

이 환경에서 함수 호출 `(f 1)`을 실행하면 `f`가 의미하는 함수의 몸통식 `(f x)`와 저장된 환경 $\emptyset$을 가져온다. 환경을 인자에 대해서 확장하면 $\{\mathtt{x} \mapsto 1\}$이 되고 몸통식 `(f x)`를 계산하게 되는데 현재 환

경에 f에 대한 정보가 없으므로 더 이상 실행이 불가능하다.

$$\dfrac{\{\mathtt{f} \mapsto (\mathtt{x}, \mathtt{f\ x}, \emptyset)\} \vdash \mathtt{f} \Rightarrow (\mathtt{x}, \mathtt{f\ x}, \emptyset) \quad \dfrac{\{\mathtt{x} \mapsto 1\} \vdash \mathtt{f} \Rightarrow ? \quad \{\mathtt{x} \mapsto 1\} \vdash \mathtt{x} \Rightarrow 1}{\{\mathtt{x} \mapsto 1\} \vdash \mathtt{f\ x} \Rightarrow ?}}{\{\mathtt{f} \mapsto (\mathtt{x}, \mathtt{f\ x}, \emptyset)\} \vdash \mathtt{f\ 1} \Rightarrow ?}$$

따라서 현재 언어에서는 재귀 함수를 사용할 수 없다. 재귀 함수를 지원하기 위하여 언어를 확장해보자. 먼저 다음과 같이 문법 구조를 확장하자.

$$\begin{array}{rcl} E & \rightarrow & \cdots \\ & | & \mathtt{letrec}\ f(x) = E_1\ \mathtt{in}\ E_2 \end{array}$$

재귀 함수를 정의하기 위해 식 letrec $f(x)$ = E_1 in E_2를 추가하였다. 여기서 f는 함수의 이름, x는 인자, E_1은 함수의 몸통이다. let과 마찬가지로 정의된 함수 f는 E_2에서만 사용할 수 있다. 예를 들어, 앞의 예제를 다음과 같이 작성할 수 있다.

```
letrec f(x) = (f x) in (f 1)
```

의미구조 정의를 위해서 프로그램이 다루는 값이 재귀 함수를 포함하도록 다음과 같이 확장하자.

$$\begin{array}{rcl} \mathit{Val} & = & \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} + \mathit{RecProcedure} \\ \mathit{Procedure} & = & \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \\ \mathit{RecProcedure} & = & \mathit{Var} \times \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \\ \mathit{Env} & = & \mathit{Var} \rightarrow \mathit{Val} \end{array}$$

재귀 함수의 값이 일반 함수와 다른 점은 함수 이름을 추가적으로 기억한다는 점이다. 재귀 함수도 정적 유효범위를 가지고 호출하기 위해서 정의될 때의 환경을 함수값에 포함시켰다.

재귀 함수 생성식의 의미는 다음과 같다.

$$\frac{\{f \mapsto (f, x, E_1, \rho)\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{letrec } f(x) \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v}$$

환경 ρ에서 재귀 함수 $f(x)$ = E_1의 함수값 (f, x, E_1, ρ)을 계산한 후 현재 환경(ρ)을 확장하여 E_2를 계산한다. 재귀 함수를 정의하고 있다는 점을 제외하면 일반적인 let식의 의미와 동일하다.

재귀 함수를 호출하는 규칙은 다음과 같다.

$$\frac{\rho \vdash E_1 \Rightarrow (f, x, E, \rho') \qquad \rho \vdash E_2 \Rightarrow v \qquad \{x \mapsto v, f \mapsto (f, x, E, \rho')\}\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

비재귀 함수 호출과 달리 함수 호출을 할 때 함수가 정의된 시점의 환경 ρ'에 함수 인자와, 호출되는 함수를 함께 확장한다는 점이 다르다.

예를 들어, 앞의 프로그램은 다음과 같이 실행된다. 함수가 재귀적으로 무한히 호출되는 모습을 볼 수 있다(주요 과정만 나타내

$$
\begin{array}{rcl}
E & \to & n \\
& | & x \\
& | & E_1 \texttt{ + } E_2 \\
& | & E_1 \texttt{ - } E_2 \\
& | & E_1 \texttt{ * } E_2 \\
& | & E_1 \texttt{ / } E_2 \\
& | & \texttt{iszero } E \\
& | & \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \\
& | & \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \\
& | & \texttt{letrec } f(x) \texttt{ = } E_1 \texttt{ in } E_2 \\
& | & \texttt{fun } x \ E \\
& | & E_1 \ E_2
\end{array}
$$

그림 4.3: 재귀 함수를 포함한 언어의 문법구조

었다).

$$
\dfrac{\dfrac{\dfrac{\vdots}{\{\texttt{x} \mapsto 1, \texttt{f} \mapsto (\texttt{f}, \texttt{x}, \texttt{f x}, \emptyset)\} \vdash \texttt{f x} \Rightarrow}}{\{\texttt{x} \mapsto 1, \texttt{f} \mapsto (\texttt{f}, \texttt{x}, \texttt{f x}, \emptyset)\} \vdash \texttt{f x} \Rightarrow}}{\{\texttt{f} \mapsto (\texttt{f}, \texttt{x}, \texttt{f x}, \emptyset)\} \vdash \texttt{f 1} \Rightarrow}
\over
\emptyset \vdash \texttt{letrec f(x) = (f x) in (f 1)} \Rightarrow
$$

이제 재귀 함수를 이용하여 영원히 도는 프로그램을 작성할 수 있게 되었다. 끝나지 않는 프로그램의 실행은 위와 같이 무한한 크기의 증명 나무를 만드는 과정에 해당한다.

재귀 함수까지 포함한 언어의 문법구조와 의미구조를 정리하면 그림 4.3과 4.4와 같다. 이제 의미구조 정의에서 함수 호출식 $E_1\ E_2$을 위한 실행 규칙이 두 가지가 되었다. E_1을 계산한 결과에 따라서 적용할 규칙을 선택하게 된다.

참고로 동적 유효범위에서는 재귀 함수가 별다른 확장없이 잘

$$\frac{}{\rho \vdash n \Rightarrow n} \qquad \frac{}{\rho \vdash x \Rightarrow \rho(x)}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2}$$

$$\frac{\rho \vdash E \Rightarrow 0}{\rho \vdash \texttt{iszero } E \Rightarrow \mathit{true}} \qquad \frac{\rho \vdash E \Rightarrow n}{\rho \vdash \texttt{iszero } E \Rightarrow \mathit{false}}\ n \neq 0$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{true} \qquad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{false} \qquad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \{x \mapsto v_1\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v}$$

$$\frac{\{f \mapsto (f, x, E_1, \rho)\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{letrec } f(x) \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v}$$

$$\frac{}{\rho \vdash \texttt{fun } x\ E \Rightarrow (x, E, \rho)}$$

$$\frac{\rho \vdash E_1 \Rightarrow (x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad \{x \mapsto v\}\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

$$\frac{\begin{array}{c}\rho \vdash E_1 \Rightarrow (f, x, E, \rho') \qquad \rho \vdash E_2 \Rightarrow v \\ \{x \mapsto v, f \mapsto (f, x, E, \rho')\}\rho' \vdash E \Rightarrow v'\end{array}}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

그림 4.4: 재귀 함수를 포함한 언어의 의미구조

동작한다. 다시 아래 프로그램을 생각해보자.

```
let f = fun x (f x) in (f 1)
```

동적 유효범위를 이용한 함수호출의 실행 의미

$$\frac{\rho \vdash E_1 \Rightarrow (x, E) \quad \rho \vdash E_2 \Rightarrow v \quad \{x \mapsto v\}\rho \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

에 따라서 프로그램을 실행해보면 다음과 영원히 실행된다.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\{\texttt{f} \mapsto (\texttt{x}, \texttt{f x}), \texttt{x} \mapsto 1\} \vdash \texttt{f x} \Rightarrow}}{\{\texttt{f} \mapsto (\texttt{x}, \texttt{f x}), \texttt{x} \mapsto 1\} \vdash \texttt{f x} \Rightarrow}}{\{\texttt{f} \mapsto (\texttt{x}, \texttt{f x})\} \vdash \texttt{f 1} \Rightarrow}}{\emptyset \vdash \texttt{let f = fun (x) (f x) in (f 1)} \Rightarrow}$$

동적 유효범위에서는 함수의 몸통을 함수 호출이 일어날 때의 환경에서 계산하는데 그 환경에는 이미 함수 f에 대한 정보가 존재하고 있다.

4.3 구현

이 장에서 정의한 언어의 실행기를 구현해보자. OCaml로 정의한 대상 언어는 다음과 같다.

```
type program = exp
and exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
```

```
  | SUB of exp * exp
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
and var = string
```

예를 들어, 아래 프로그램

```
let f = fun x (x+1)
in f (f 1)
```

은 다음과 같이 표현된다.

```
  LET ("f", PROC ("x", ADD (VAR "x", CONST 1)),
    CALL (VAR "f", CALL (VAR "f", CONST 1)))
```

값과 환경은 다음과 같이 정의할 수 있다.

```
type value = Int of int | Bool of bool
           | Procedure of var * exp * env
           | RecProcedure of var * var * exp * env
and env = var -> value

let empty_env =
  fun x -> raise (Failure ("env is empty"))
let extend_env (x,v) e =
  fun y -> if x = y then v else (e y)
let apply_env e x = e x
```

위에서 환경을 수학적 정의 그대로 함수로 정의하였다. 빈 환경은 아무 정보도 가지지 않으므로 임의의 변수에 대해서 예외를 발생시키는 함수로 정의하였다. 환경 e에서 변수 x의 값은 함수 호출

(e x)을 통해 얻을 수 있다. 환경 e에서 변수 x가 값 v를 가지도록 확장한 환경은 아래 함수가 된다.

```
fun y -> if x = y then v else (e y)
```

변수 y를 받아서 y가 x와 일치하면 v를, 아니면 이전 환경 e에서 값을 찾는다는 뜻이다. 이와 같은 구현이 익숙하지 않다면 앞 장의 리스트로 구현한 환경을 이용해도 된다.

이 장에서 정의한 프로그래밍 언어의 실행기

```
eval : exp -> env -> value
```

를 구현해보자. 정적 유효범위와 동적 유효범위를 사용하는 두 경우를 모두 구현하고 테스트해보자.

5

함수형 언어 Fun

지금까지의 언어를 좀 더 그럴듯하게 확장해보자. 앞 장의 언어를 기본으로 하되 유닛(()), 참/거짓(`true`, `false`), 리스트, 상호 재귀 함수(mutually recursive function)를 사용할 수 있도록 한 함수형 언어 Fun을 설계하고 구현해보자.

5.1 문법구조

Fun의 문법구조는 그림 5.1과 같다. 예를 들어, OCaml로 표현한 리스트를 뒤집는 함수

```
let rec reverse l =
  match l with
  | [] -> []
  | hd::tl -> (reverse tl)@[hd]
in reverse [1;2;3]
```

를 Fun에서는 다음과 같이 작성할 수 있다.

```
letrec reverse(l) =
  if (isnil l) then nil
  else (reverse (tail l)) @ ((head l)::nil)
in reverse 1::2::3::nil
```

다음과 같이 상호 재귀 함수를 정의하고 사용할 수 있다.

```
letrec even(x) = if (x = 0) then true  else odd(x-1)
```

$$
\begin{array}{lcl}
P & \rightarrow & E \\
E & \rightarrow & \texttt{()} \\
 & | & \texttt{true} \mid \texttt{false} \\
 & | & n \\
 & | & x \\
 & | & E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E \\
 & | & E \texttt{ = } E \mid E \texttt{ < } E \\
 & | & \texttt{not } E \\
 & | & \texttt{nil} \\
 & | & E \texttt{ :: } E \\
 & | & E \texttt{ @ } E \\
 & | & \texttt{head } E \\
 & | & \texttt{tail } E \\
 & | & \texttt{isnil } E \\
 & | & \texttt{if } E \texttt{ then } E \texttt{ else } E \\
 & | & \texttt{let } x \texttt{ = } E \texttt{ in } E \\
 & | & \texttt{letrec } f(x) \texttt{ = } E \texttt{ in } E \\
 & | & \texttt{letrec } f(x_1) = E_1 \texttt{ and } g(x_2) = E_2 \texttt{ in } E \\
 & | & \texttt{fun } x\ E \\
 & | & E\ E \\
 & | & \texttt{print } E \\
 & | & E; E
\end{array}
$$

그림 5.1: Fun의 문법구조

```
    and odd(x) = if (x = 0) then false else even(x-1)
in (even 9)
```

함수 even은 인자 x가 짝수이면 참을, 홀수이면 거짓을 반환하는 함수이다. 함수 odd는 인자가 홀수인 경우 참을 반환하는 함수이다. 두 함수를 상호 재귀적으로 정의하여 서로가 서로를 호출하도록 하였다. 간편함을 위해 그림 5.1의 문법구조에서 두 개의 함수만 상호 재귀적으로 작성할 수 있도록 제한을 두었다.

$$
\begin{array}{rcl}
v \in Val & = & \{\cdot\} + \mathbb{Z} + Bool + List + \\
 & & Procedure + RecProcedure + MRecProcedure \\
n \in \mathbb{Z} & = & \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
b \in Bool & = & \{true, false\} \\
s \in List & = & Val^* \\
Procedure & = & Var \times Exp \times Env \\
RecProcedure & = & Var \times Var \times Exp \times Env \\
MRecProcedure & = & (Var \times Var \times Exp) \times (Var \times Var \times Exp) \times Env
\end{array}
$$

그림 5.2: Fun의 의미공간

5.2 의미구조

의미공간은 그림 5.2와 같다. 값은 유닛($\cdot$), 정수($\mathbb{Z}$), 부울값($Bool$), 리스트($List$), 함수($Procedure$), 재귀 함수($RecProcedure$), 상호 재귀 함수($MRecProcedure$)들의 집합으로 정의하였다. 의미공간 정의에서 Val^*는 값을 원소로 가지는 리스트들의 집합을 뜻한다. 리스트는 $[a_1, a_2, \ldots, a_n]$과 같이 나타내고 빈 리스트는 []로 나타내기로 하자. 값 v와 리스트 s에 대해서 $v :: s$는 s앞에 v를 추가한 리스트를 뜻한다고 하자. 두 리스트 s_1과 s_2가 있을 때 $s_1 @ s_2$는 s_1과 s_2를 이어붙인 리스트를 뜻한다. 환경(Env)은 이전과 동일하게 변수에서 값으로 가는 함수이다.

$$\rho \in Env = Var \to Val$$

문법구조에 의해 상호 재귀 함수는 항상 두 함수가 함께 정의되므로 그 값을 아래와 같이 정의하였다.

$$(Var \times Var \times Exp) \times (Var \times Var \times Exp) \times Env$$

두 함수의 이름, 인자, 몸통식을 포함하고 추가로 정의될 때의 환경을 기억하도록 하였다(환경은 두 함수가 공유한다). 예를 들어, 위에서 `even`, `odd`를 정의한 경우 아래와 같은 값이 만들어진다.

$$((\texttt{even}, \texttt{x}, \texttt{if (x=0) then true else odd(x-1)}),$$
$$(\texttt{odd}, \texttt{x}, \texttt{if (x=0) then false else even(x-1)}), \emptyset)$$

의미구조 규칙들을 하나씩 정의해보자. 상수를 뜻하는 식들은 다음과 같이 기본값들을 만들어낸다.

$$\frac{}{\rho \vdash \texttt{()} \Rightarrow \cdot}$$

$$\frac{}{\rho \vdash \texttt{true} \Rightarrow \mathit{true}}$$

$$\frac{}{\rho \vdash \texttt{false} \Rightarrow \mathit{false}}$$

$$\frac{}{\rho \vdash n \Rightarrow n}$$

변수가 뜻하는 바는 환경에서 그 변수가 가지는 값이다.

$$\frac{}{\rho \vdash x \Rightarrow \rho(x)}$$

산술 연산자들의 의미는 다음과 같다.

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ - } E_2 \Rightarrow n_1 - n_2}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ * } E_2 \Rightarrow n_1 * n_2}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ / } E_2 \Rightarrow n_1/n_2} \; n_2 \neq 0$$

비교 연산자(=, <)의 의미를 다음과 같이 정의하자.

$$\frac{\rho \vdash E_1 \Rightarrow b_1 \qquad \rho \vdash E_2 \Rightarrow b_2}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{true}} \; b_1 = b_2$$

$$\frac{\rho \vdash E_1 \Rightarrow b_1 \qquad \rho \vdash E_2 \Rightarrow b_2}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{false}} \; b_1 \neq b_2$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{true}} \; n_1 = n_2$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{false}} \; n_1 \neq n_2$$

$$\frac{\rho \vdash E_1 \Rightarrow [a_1, \ldots, a_n] \quad \rho \vdash E_2 \Rightarrow [b_1, \ldots, b_m]}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{true}} \; n = m \wedge \forall i. a_i = b_i$$

$$\frac{\rho \vdash E_1 \Rightarrow [a_1, \ldots, a_n] \quad \rho \vdash E_2 \Rightarrow [b_1, \ldots, b_m]}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{false}} \; n \neq m \vee \exists i. a_i \neq b_i$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ < } E_2 \Rightarrow \mathit{true}} \; n_1 < n_2$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ < } E_2 \Rightarrow \mathit{false}} \; n_1 \geq n_2$$

위의 의미구조 정의에서 두 값이 같음을 비교할 때(E_1 `=` E_2) 값이 정수, 부울값, 리스트인 경우만 고려하였다. 리스트의 경우 길이가 같고 원소들이 모두 동일한 경우 같다고 정의하였다.[1] 함수값을 비교하는 것은 불가능하므로 고려하지 않았다. 함수값은 그 속에 프로그램(함수의 몸통)을 포함하고 있고 두 프로그램의 의미가 같은지를 확인하는 것은 일반적으로 계산 불가능(undecidable)한 문제이기 때문이다. OCaml에서도 다음과 같이 두 함수를 비교하면 예외가 발생한다.

```
# (fun x -> x) = (fun y -> y);;
Exception: Invalid_argument "compare: functional value".
```

논리 연산자 `not`의 의미는 다음과 같다.

$$\frac{\rho \vdash E \Rightarrow \mathit{true}}{\rho \vdash \texttt{not } E \Rightarrow \mathit{false}}$$

$$\frac{\rho \vdash E \Rightarrow \mathit{false}}{\rho \vdash \texttt{not } E \Rightarrow \mathit{true}}$$

리스트는 다음 세 가지 방법으로 만들어진다.

$$\frac{}{\rho \vdash \texttt{nil} \Rightarrow []}$$

1) 사실 리스트의 경우 두 값이 같을 조건이 엄밀하게 정의되지 않았고 해석의 여지가 남아있다. 예를 들어, 리스트의 원소가 또 다른 리스트를 포함하는 경우 위 의미정의에서는 어떻게 해석할지 잘 드러나지 않는다. 구현에서는 연산자 =를 어떻게 정의하는지에 따라 의미가 결정될 것이다.

$$\frac{\rho \vdash E_1 \Rightarrow v \qquad \rho \vdash E_2 \Rightarrow s}{\rho \vdash E_1 \texttt{ :: } E_2 \Rightarrow v :: s}$$

$$\frac{\rho \vdash E_1 \Rightarrow s_1 \qquad \rho \vdash E_2 \Rightarrow s_2}{\rho \vdash E_1 \texttt{ @ } E_2 \Rightarrow s_1 @ s_2}$$

Fun에서 빈 리스트는 `nil`로 표현한다. 연산자 `::`와 `@`는 OCaml의 것과 동일하다. 위 의미정의에서 v와 s는 각각 임의의 값과 리스트 값을 나타낸다. 다른 리스트 연산자들의 의미는 다음과 같다.

$$\frac{\rho \vdash E \Rightarrow v :: s}{\rho \vdash \texttt{head } E \Rightarrow v} \qquad \frac{\rho \vdash E \Rightarrow v :: s}{\rho \vdash \texttt{tail } E \Rightarrow s}$$

$$\frac{\rho \vdash E \Rightarrow []}{\rho \vdash \texttt{isnil } E \Rightarrow \mathit{true}} \qquad \frac{\rho \vdash E \Rightarrow v :: s}{\rho \vdash \texttt{isnil } E \Rightarrow \mathit{false}}$$

그 밖에 식들의 의미는 앞 장에서 정의한 그대로이다. 조건식의 의미는 다음과 같다.

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{true} \qquad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{false} \qquad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v}$$

let식의 의미는 다음과 같다.

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \{x \mapsto v_1\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v}$$

$$\frac{\{f \mapsto (f, x, E_1, \rho)\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{letrec } f(x) = E_1 \texttt{ in } E_2 \Rightarrow v}$$

함수 정의와 호출의 의미는 다음과 같다.

$$\frac{}{\rho \vdash \texttt{fun}\ x\ E \Rightarrow (x, E, \rho)}$$

$$\frac{\rho \vdash E_1 \Rightarrow (x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad \{x \mapsto v\}\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

$$\frac{\begin{array}{c}\rho \vdash E_1 \Rightarrow (f, x, E, \rho') \qquad \rho \vdash E_2 \Rightarrow v \\ \{x \mapsto v, f \mapsto (f, x, E, \rho')\}\rho' \vdash E \Rightarrow v'\end{array}}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

상호 재귀 함수를 정의하고 호출하기 위한 의미구조도 필요하다. 먼저 상호 재귀 함수는 다음과 같이 생성된다.

$$\frac{\{f \mapsto (f, x, E_1, g, y, E_2, \rho), g \mapsto (g, y, E_2, f, x, E_1, \rho)\}\rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{letrec}\ f(x) = E_1\ \texttt{and}\ g(y) = E_2\ \texttt{in}\ E_3 \Rightarrow v}$$

함수값 $(f, x, E_1, g, y, E_2, \rho)$는 함수 f의 값을 뜻하며 함수 g와 상호 재귀적으로 정의되었음을 뜻한다. 함수 g의 값은 $(g, y, E_2, f, x, E_1, \rho)$와 같이 나타낸다. 상호 재귀적으로 정의된 함수를 호출하는 규칙은 다음과 같다.

$$\frac{\begin{array}{c}\rho \vdash E_1 \Rightarrow (f, x, E_f, g, y, E_g, \rho') \quad \rho \vdash E_2 \Rightarrow v \\ \left\{\begin{array}{l} x \mapsto v, \\ f \mapsto (f, x, E_f, g, y, E_g, \rho'), \\ g \mapsto (g, y, E_g, f, x, E_f, \rho') \end{array}\right\} \rho' \vdash E_f \Rightarrow v'\end{array}}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

호출된 함수(f)의 몸통식 E_f를 계산할 때 환경을 상호적으로 정의된 함수 g에 대해서도 확장해 주는 것이 핵심이다. 그래야 E_f내에서 g를 호출할 수 있다.

식 `print` E는 E의 값을 계산하여 화면에 출력하는 역할을 한다.

$$\frac{}{\rho \vdash \texttt{print}\ E \Rightarrow \cdot}$$

보통 의미구조 규칙에 입출력 등의 외부 환경과 상호작용까지 표현하지는 않는다. 따라서 식 `print` E를 계산하면 유닛값이 생성된다는 것만 표현하였다. 마지막으로 식 $E_1; E_2$는 두 식 E_1과 E_2를 차례로 계산하라는 뜻이다.

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \rho \vdash E_2 \Rightarrow v_2}{\rho \vdash E_1; E_2 \Rightarrow v_2}$$

식 E_1을 계산하기는 하지만 그 결과값 v_1은 무시하고 있다.

5.3 구현

지금까지 정의한 문법구조와 의미구조를 OCaml로 구현해보자. 먼저 문법구조는 다음의 데이터 타입으로 정의할 수 있다.

```
type program = exp
and exp =
  | UNIT
  | TRUE
  | FALSE
  | CONST of int
```

```
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
  | DIV of exp * exp
  | EQUAL of exp * exp
  | LESS of exp * exp
  | NOT of exp
  | NIL
  | CONS of exp * exp
  | APPEND of exp * exp
  | HEAD of exp
  | TAIL of exp
  | ISNIL of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | LETMREC of (var * var * exp) *
               (var * var * exp) * exp
  | PROC of var * exp
  | CALL of exp * exp
  | PRINT of exp
  | SEQ of exp * exp
and var = string
```

값과 환경을 다음과 같이 정의하자.

```
type value =
  | Unit
  | Int of int
  | Bool of bool
  | List of value list
  | Procedure of var * exp * env
  | RecProcedure of var * var * exp * env
  | MRecProcedure of var * var * exp *
```

```
                    var * var * exp * env
and env = (var * value) list
```

프로그램을 받아서 값을 계산하는 아래 함수를 구현해보자.

```
run : program -> value
```

몇 가지 실행 예를 보이면 다음과 같다. 앞에서 정의한 상호 재귀 함수

```
letrec even(x) = if (x = 0) then true  else odd(x-1)
    and odd(x) = if (x = 0) then false else even(x-1)
in (even 8)
```

는 OCaml 데이터 타입으로 다음과 같이 표현된다.

```
  LETMREC
   (("even", "x",
     IF (EQUAL (VAR "x", CONST 0), TRUE,
      CALL (VAR "odd", SUB (VAR "x", CONST 1)))),
   ("odd", "x",
    IF (EQUAL (VAR "x", CONST 0), FALSE,
     CALL (VAR "even", SUB (VAR "x", CONST 1)))),
   CALL (VAR "even", CONST 8))
```

위 프로그램을 run으로 실행하면 Bool true가 계산되어야 한다.

아래 프로그램에서는 팩토리얼 함수를 정의하고 1에서 10 팩토리얼을 역순으로 출력하도록 하였다.

```
letrec factorial(x) =
         if (x = 0) then 1
         else factorial(x-1) * x
```

```
in letrec loop n =
     if (n = 0) then ()
     else (print (factorial n); loop (n-1))
   in (loop 10)
```

OCaml로 표현한 프로그램은 다음과 같고

```
  LETREC ("factorial", "x",
   IF (EQUAL (VAR "x", CONST 0), CONST 1,
    MUL (CALL (VAR "factorial",
            SUB (VAR "x", CONST 1)), VAR "x")),
   LETREC ("loop", "n",
    IF (EQUAL (VAR "n", CONST 0), UNIT,
     SEQ (PRINT (CALL (VAR "factorial", VAR "n")),
      CALL (VAR "loop", SUB (VAR "n", CONST 1)))),
    CALL (VAR "loop", CONST 10)))
```

실행하면 아래 내용이 화면에 출력된다. 계산되는 최종 값은 `Unit` 이다.

```
3628800
362880
40320
5040
720
120
24
6
2
1
```

1부터 10까지의 정수로 구성된 리스트를 만드는 프로그램은 다음과 같이 작성할 수 있다.

```
letrec range(n) =
      if (n = 1) then (1::nil)
      else n::(range (n-1))
in (range 10)
```

OCaml로는 다음과 같이 표현할 수 있고

```
  LETREC ("range", "n",
   IF (EQUAL (VAR "n", CONST 1), CONS (CONST 1, NIL),
    CONS (VAR "n",
      CALL (VAR "range", SUB (VAR "n", CONST 1)))),
   CALL (VAR "range", CONST 10))
```

실행하면 리스트 값 `List [Int 10; Int 9; ...; Int 1]` 가 생성된다.

리스트를 뒤집는 함수

```
letrec reverse(l) =
  if (isnil l) then nil
  else (reverse (tail l)) @ ((head l)::nil)
in reverse 1::2::3::nil
```

를 OCaml로 정의하면 다음과 같고

```
  LETREC ("reverse", "l",
   IF (ISNIL (VAR "l"), NIL,
    APPEND (CALL (VAR "reverse", TAIL (VAR "l")),
     CONS (HEAD (VAR "l"), NIL))),
   CALL (VAR "reverse",
    CONS (CONST 1, CONS (CONST 2, CONS (CONST 3, NIL)))))
```

실행하면 리스트 값 `List [Int 3; Int 2; Int 1]`이 생성된다.

6

상태 변화

지금까지 디자인한 언어는 상태 변화를 지원하지 않는 순수한 함수형 언어(purely functional language)였다. 이러한 언어에서 프로그램 실행은 순전히 값을 계산하는 과정이었다. 이 장에서는 명령형 언어(imperative language)의 주요 특징인 값의 변화가 가능하도록 언어를 확장해보자.

지금까지는 변수가 한번 정의되면 그 값을 변경하는 것이 불가능했다. 예를 들어, 아래 프로그램을 실행하면서 함수 f가 호출되는 횟수를 세고 싶다고 해보자.

```
let f = fun x (x)
in (f (f 1))
```

아래와 같이 시도해볼 수 있겠지만 작동하지 않는다.

```
let counter = 0
in let f = fun x (let counter = counter + 1 in x)
   in let a = (f (f 1))
      in counter
```

함수가 호출되는 횟수를 세기 위하여 변수 counter를 도입하고 함수 몸통이 계산될 때마다 counter의 값을 하나 증가시켜서 다시 정의하였다. 그다음 셋째줄에서 함수를 두 번 호출한 후 마지막에 counter의 값을 계산하는 프로그램이다. 최종값으로 2를 의도한

것이지만 실제 계산 결과는 (정적/동적 유효범위에 상관없이) 0이다. 함수 몸통에서 `counter`의 값을 다시 정의한다고 해서 첫째줄에서 정의한 변수 `counter`의 값이 바뀌는 것은 아니기 때문이다.

위와 같은 일을 가능하게 하려면 환경외에 값의 변경이 가능한 장치인 메모리가 필요하다. 메모리를 지원하고 값의 변화가 가능하도록 언어를 확장해보자. 프로그래밍 언어가 값의 변화를 지원하는 방식은 메모리에 접근하고 사용하는 방식에 따라 크게 두 가지로 분류할 수 있다.

6.1 첫 번째 방안

첫 번째는 메모리 할당, 접근, 변경을 명시적인 문법을 통해 지원하는 언어들이다. OCaml, F#등의 언어에서 사용하는 방식이다.[1)]

6.1.1 문법구조

앞장의 언어를 그림 6.1과 같이 확장하자. 네 가지 구문을 새롭게 추가하였다.

- `ref` E는 메모리에 새로운 주소를 할당하는 명령문이다. 새로 할당한 주소를 l이라고 하면, E의 값을 계산하여 l에 저장한 후 주소 l을 반환한다.

1) 앞에서 소개하지는 않았지만 OCaml도 이 절에서 사용하는 것과 동일한 문법으로 메모리 상태 변화를 지원한다.

$$
\begin{array}{rcll}
E & \rightarrow & n & \\
 & | & x & \\
 & | & E_1 \texttt{ + } E_2 & \\
 & | & \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 & \\
 & | & \texttt{iszero } E & \\
 & | & \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 & \\
 & | & \texttt{fun } x\ E & \\
 & | & E_1\ E_2 & \\
 & | & \texttt{ref } E & \text{메모리 할당} \\
 & | & \texttt{!}E & \text{메모리 접근} \\
 & | & E_1 := E_2 & \text{메모리 변경} \\
 & | & E_1; E_2 & \text{순서문}
\end{array}
$$

그림 6.1: 문법구조

- $\texttt{!}E$는 E가 의미하는 메모리 주소에 담겨 있는 값을 의미한다.
- $E_1 := E_2$는 E_1이 가리키는 메모리 주소에 담긴 값을 E_2가 의미하는 값으로 변경한다.
- $E_1; E_2$는 E_1과 E_2를 순서대로 실행하는 명령문이다.

예를 들어 이 언어로 함수의 호출 횟수를 세는 프로그램을 아래와 같이 작성할 수 있다.

```
let cnt = ref 0
in let f = fun x (cnt := !cnt + 1; !cnt)
   in let a = (f 0)
      in let b = (f 0)
         in a + b
```

호출 횟수를 저장하기 위하여 메모리를 할당하고 초기값으로 0을 저장한 후 변수 cnt가 그 메모리 주소를 가리키도록 하였다. 이러

한 변수 cnt를 레퍼런스(reference) 변수라고 부른다. 함수 f가 호출될 때마다 cnt가 가리키는 메모리 주소에 담긴 값을 1씩 증가시킨 후 함수의 결과값으로 현재 값(!cnt)을 반환한다. 따라서 a, b는 각각 1, 2가 되어 최종적으로 a+b의 값으로 3을 계산하게 된다.

6.1.2 의미구조

의미구조를 정의하기 위해 다음과 같이 의미공간을 확장하자.

$$\begin{aligned} \mathit{Val} &= \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} + \mathit{Loc} \\ \mathit{Procedure} &= \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \\ \rho \in \mathit{Env} &= \mathit{Var} \to \mathit{Val} \\ \sigma \in \mathit{Mem} &= \mathit{Loc} \to \mathit{Val} \end{aligned}$$

이제 프로그램 실행 중에 메모리 주소가 만들어질 수 있고, 메모리 주소가 값으로 사용될 수 있다. *Loc*을 메모리 주소들의 전체 집합이라고 하자. 메모리는 주소에서 값으로 가는 유한함수로 정의하였다. $\sigma, \sigma', \sigma_1$등의 문자를 메모리를 지칭하는데 사용할 것이다. 함수와 환경의 정의는 변화없이 이전과 동일하다. 이 언어에서 변수는 여전히 값에 붙인 이름을 뜻한다. 달라진 점은 메모리 주소도 값이 될 수 있고 메모리의 해당 주소에 변경 가능한 값이 저장된다는 것이다.

이제 프로그램 실행은 값을 계산하는 것 외에도 메모리 상태 변화를 수반하게 된다. 프로그램을 현재 환경과 메모리에서 실행시키면 그 결과로 값이 계산될 뿐 아니라 변경된 메모리도 얻게 된다.

이를 다음과 같이 표현하자.

$$\rho, \sigma \vdash E \Rightarrow v, \sigma'$$

환경 ρ와 메모리 σ가 주어져 있을 때 프로그램 E를 실행하면 그 결과로 값 v가 계산되고 입력으로 주어진 메모리 σ가 출력 메모리 σ'으로 상태가 변경된다는 의미이다.

새로 추가된 네 가지 구문을 제외하면 이전 언어의 의미구조를 메모리 변화를 고려하도록 단순히 확장하면 된다(그림 6.2). 각 식마다 계산할 때 생길 수 있는 메모리 변화가 표시되어 있고 이러한 변화를 계속 누적시키고 있다. 예를 들어, 사칙 연산의 경우를 보자.

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2, \sigma_2}$$

식 E_1 + E_2을 계산하기 위해 주어진 메모리 σ_0를 이용하여 먼저 식 E_1을 계산하고 그 동안에 발생한 메모리 변화를 σ_1에 기록하였다. 그다음에 식 E_2를 메모리 σ_1을 가지고 계산하게 되고 식 E_2를 계산하면서 발생한 변화들이 σ_2에 담긴다. 결과적으로 σ_2에 식 E_1 + E_2를 계산하면서 발생한 모든 변화가 담기게 된다.

새로 추가한 구문들의 의미를 정의해보자.

- 먼저 메모리 할당문의 의미는 다음과 같다.

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash \texttt{ref } E \Rightarrow l, \{l \mapsto v\}\sigma_1} \quad l \notin \mathrm{Dom}(\sigma_1)$$

$$\frac{}{\rho, \sigma \vdash n \Rightarrow n, \sigma} \qquad \frac{}{\rho, \sigma \vdash x \Rightarrow \rho(x), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow 0, \sigma_1}{\rho, \sigma_0 \vdash \texttt{iszero } E \Rightarrow \mathit{true}, \sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow n, \sigma_1}{\rho, \sigma_0 \vdash \texttt{iszero } E \Rightarrow \mathit{false}, \sigma_1} \; n \neq 0$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow \mathit{true}, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow \mathit{false}, \sigma_1 \qquad \rho, \sigma_1 \vdash E_3 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad \{x \mapsto v_1\}\rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v, \sigma_2}$$

$$\frac{}{\rho, \sigma \vdash \texttt{fun } x \; E \Rightarrow (x, E, \rho), \sigma}$$

$$\frac{\begin{array}{c}\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \\ \{x \mapsto v\}\rho', \sigma_2 \vdash E \Rightarrow v', \sigma_3\end{array}}{\rho, \sigma_0 \vdash E_1 \; E_2 \Rightarrow v', \sigma_3}$$

그림 6.2: 메모리 변화를 포함하도록 확장한 의미구조

할당문 `ref` E를 실행하기 위하여 먼저 E의 값을 주어진 환경(ρ)과 메모리(σ_0)에서 계산한다. E의 값을 v라고 하고 E를 계산하면서 변경된 메모리를 σ_1이라고 하자. 그다음 메모리 σ_1에서 새 주소를 할당한다. 여기서 조건식 $l \notin \mathrm{Dom}(\sigma_1)$은 주소 l이 메모리 σ_1에 할당되어 있지 않은 새 주소임을 의미한다. 새 주소 l을 메모리 σ_1에 새롭게 할당한다는 뜻이다.

그다음 `ref` E는 할당한 주소값 l을 값으로 반환하고, 메모리 σ_1내의 주소 l의 값을 변경한다($\{l \mapsto v\}\sigma_1$). 처음 주어진 메모리 σ_0가 식 E를 계산하면서 σ_1으로 변경된 후 최종적으로 $\{l \mapsto v\}\sigma_1$로 변경된다.

- 두 번째는 메모리 주소에 들어있는 값에 접근하는 구문이다.

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow l, \sigma_1}{\rho, \sigma_0 \vdash \texttt{!}E \Rightarrow \sigma_1(l), \sigma_1}$$

먼저 식 E의 값을 계산한다. 이때 E의 계산결과는 주소값이어야 한다(l). 식 E를 계산했을 때 다른 타입의 값이 나오는 경우는 의미가 정의되지 않는다. 예를 들어, `!(1+2)` 또는 `!(fun` x $(x+1)$) 등은 문법구조에 의하면 잘 정의된 프로그램이지만 의미를 가지지 않는다. 식 E가 주소값 l을 계산하고 메모리 변화가 σ_1에 반영되어 있다고 하자. 그러면 `!`E는 메모리 σ_1에서 주소 l이 가지는 값인 $\sigma_1(l)$을 결과값으로 내놓고 메모리의 주소값을 읽어오는 연산 자체는 메모리 변화를 일으키지 않으므로 최종 출력 메모리는 σ_1이 된다.

- 메모리 주소의 값을 변경시키는 대입문(assignment)의 의미는 다음과 같다.

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow l, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash E_1 := E_2 \Rightarrow v, \{l \mapsto v\}\sigma_2}$$

먼저 E_1을 계산하여 주소값 l을 얻을 수 있어야 한다. 이때

메모리 변화가 σ_1에 반영되어 있다고 하자. 그다음에 E_2를 현재 환경 ρ와 E_1을 계산한 후의 메모리 σ_1을 가지고 실행시킨다. 그 결과값을 v라고 하고 결과 메모리를 σ_2라고 하자. 대입문은 값 v를 반환하고, 메모리 σ_2의 주소 l이 값 v를 가지도록 한다.

- $E_1; E_2$의 의미는 두 식을 차례로 계산하는 것이다.

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2}{\rho, \sigma_0 \vdash E_1; E_2 \Rightarrow v_2, \sigma_2}$$

식 E_1을 먼저 계산하고 그다음에 E_2를 계산하면서 메모리 변화를 누적시키고 있다. 이때 E_1의 값 v_1은 무시된다. E_1의 역할은 단지 메모리 변화를 일으키는 것이다.

의미구조에 따라 아래 프로그램이 어떻게 실행되는지 살펴보자. 초기 환경과 메모리는 모두 빈 환경($\emptyset$)과 빈 메모리($\emptyset$)이다.

```
let cnt = ref 0
in let f = fun x (cnt := !cnt + 1; !cnt)
   in let a = (f 0)
      in let b = (f 0)
         in a + b
```

1. 먼저 첫 번째 `let`을 처리하기 위하여 식 `ref 0`을 계산한다. 메모리 할당의 의미 정의에 따라 아래와 같이 실행된다.

$$\frac{\emptyset, \emptyset \vdash 0 \Rightarrow 0, \emptyset}{\emptyset, \emptyset \vdash \texttt{ref 0} \Rightarrow l_1, \{l_1 \mapsto 0\}} \; l_1 \notin \mathrm{Dom}(\emptyset)$$

새로운 주소 l_1을 할당하고 그 주소에 0을 저장하였다. 입력 메모리 $\emptyset$가 출력 메모리 $\{l_1 \mapsto 0\}$로 변경되었고 주소 l_1을 값으로 반환하였다. 이 결과를 가지고 `let cnt = ...`을 실행한 후의 환경과 메모리는 다음과 같다.

$$\begin{aligned} \rho_1 &: \{\texttt{cnt} \mapsto l_1\} \\ \sigma_1 &: \{l_1 \mapsto 0\} \end{aligned}$$

첫 번째 `let` 이후 변수 `cnt`는 주소값 l_1을 의미하며, 메모리의 해당 주소에는 값 0이 들어있음을 뜻한다.

2. 이제 환경 ρ_1과 메모리 σ_1을 가지고 두 번째 `let`식을 계산한다. 이를 위해 먼저 함수식

```
fun x (cnt := !cnt + 1; !cnt)
```

을 계산하여 다음과 같은 함수값을 얻는다.

$$(\texttt{x}, \texttt{(cnt:=!cnt+1;!cnt)}, \{\texttt{cnt} \mapsto l_1\})$$

만들어진 함수값에 이름 `f`를 붙여서 환경을 갱신한다. 이 상황에서의 환경과 메모리를 ρ_2, σ_2라고 하자.

$$\begin{aligned} &\rho_2 : \{\texttt{f} \mapsto (\texttt{x}, \texttt{(cnt:=!cnt+1;!cnt)}, \{\texttt{cnt} \mapsto l_1\}), \texttt{cnt} \mapsto l_1\} \\ &\sigma_2 : \{l_1 \mapsto 0\} \end{aligned}$$

3. 환경 ρ_2과 σ_2를 가지고 세 번째 `let`식을 계산한다. 이를 위해 먼저 함수 호출식 `(f 0)`을 계산하면, 함수 몸통

```
(cnt:=!cnt+1;!cnt)
```

을 계산하게 된다. 이때 사용할 환경은 함수값에 저장되어 있는 환경인 $\{\texttt{cnt} \mapsto l_1\}$에서 함수 인자를 확장한 환경인 $\{\texttt{x} \mapsto 0, \texttt{cnt} \mapsto l_1\}$이며 메모리는 호출 당시의 메모리인 $\{l_1 \mapsto 0\}$을 그대로 사용한다. 이들을 각각 ρ_3, σ_3라고 하자.

$$\begin{aligned} \rho_3 &: \{\texttt{x} \mapsto 0, \texttt{cnt} \mapsto l_1\} \\ \sigma_3 &: \{l_1 \mapsto 0\} \end{aligned}$$

이러한 환경과 메모리에서 함수 몸통을 계산하자. 먼저 대입문 `cnt:=!cnt+1`은 다음과 같이 계산된다.

$$\dfrac{\dfrac{}{\rho_3, \sigma_3 \vdash \texttt{cnt} \Rightarrow l_1, \sigma_3} \quad \dfrac{\dfrac{\dfrac{}{\rho_3, \sigma_3 \vdash \texttt{cnt} \Rightarrow l_1, \sigma_3}}{\rho_3, \sigma_3 \vdash \texttt{!cnt} \Rightarrow 0, \sigma_3} \quad \dfrac{}{\rho_3, \sigma_3 \vdash \texttt{1} \Rightarrow 1, \sigma_3}}{\rho_3, \sigma_3 \vdash \texttt{!cnt+1} \Rightarrow 1, \sigma_3}}{\rho_3, \sigma_3 \vdash \texttt{cnt:=!cnt+1} \Rightarrow 1, \{l_1 \mapsto 1\}\sigma_3}$$

먼저 `cnt:=!cnt+1`을 계산하려면 먼저 `cnt`의 값을 환경 ρ_3에서 찾는다. 주소값 l_1이다. 그다음 `!cnt`를 계산한다. 현재 메모리의 주소값 l_1에 담겨 있는 값인 0이다. 그다음 `!cnt+1`을 계산하면 1이 된다. 이 값을 메모리에서 `cnt`가 뜻하는 주

소 l_1에 저장한다. 다음과 같이 메모리 상태가 바뀌게 된다.

$$\sigma_4 \quad : \quad \{l_1 \mapsto 1\}\sigma_3 = \{l_1 \mapsto 1\}$$

그다음에 환경 ρ_3와 메모리 σ_4에서 `!cnt`를 계산한 결과는 1이다. 마지막으로 현재 환경에서 변수 `a`가 1을 의미하도록 환경을 갱신한다.

$$\{\texttt{a} \mapsto 1, \texttt{f} \mapsto (\texttt{x}, (\texttt{cnt:=!cnt+1;!cnt}), \{\texttt{cnt} \mapsto l_1\}), \texttt{cnt} \mapsto l_1\}$$

이 환경을 ρ_4라고 하자.

4. 넷째줄의 함수 호출식을 환경 ρ_4와 메모리 σ_4에서 계산한다. 이전 단계와 비슷한 과정을 거치면 다음과 같이 메모리가 변경된다.

   $$\sigma_5 \quad : \quad \{l_1 \mapsto 2\}$$

   환경은 변수 `b`가 포함되어 $\{\texttt{b} \mapsto 2\}\rho_4$가 된다.

5. 최종 환경과 메모리에서 식 `a+b`를 계산한 값은 3이다.

6.2 두 번째 방안

C, Java, Python 등의 언어에서는 값의 변화를 위한 메모리 할당과 접근이 명시적으로 드러나지 않는다. 이들 언어에서 사용하는 방식으로 상태 변화를 지원해보자.

$$
\begin{array}{rcll}
E & \to & n & \\
& | & x & \\
& | & E_1 \texttt{ + } E_2 & \\
& | & \texttt{iszero } E & \\
& | & \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 & \\
& | & \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 & \\
& | & \texttt{fun } x\ E & \\
& | & E_1\ E_2 & \\
& | & x \texttt{ := } E & \text{대입문} \\
& | & E_1; E_2 & \text{순서식}
\end{array}
$$

그림 6.3: 문법구조

6.2.1 문법구조

문법구조는 그림 6.3과 같다. 기존 언어에 대입문(assignment)과 순서식(sequence expression)만 새롭게 추가하였다. 대입문 x := E는 변수 x의 값을 E의 값으로 변경한다. 순서식 $E_1; E_2$의 의미는 두 식 E_1, E_2를 차례대로 계산하는 것이다.

6.2.2 의미구조

의미공간을 다음과 같이 정의하자.

$$
\begin{aligned}
\mathit{Val} &= \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} \\
\mathit{Procedure} &= \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \\
\rho \in \mathit{Env} &= \mathit{Var} \to \mathit{Loc} \\
\sigma \in \mathit{Mem} &= \mathit{Loc} \to \mathit{Val}
\end{aligned}
$$

$$\frac{}{\rho, \sigma \vdash x \Rightarrow \sigma(\rho(x)), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash x \texttt{ := } E \Rightarrow v, \{\rho(x) \mapsto v\}\sigma_1}$$

$$\frac{\begin{array}{c}\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \\ \{x \mapsto l\}\rho, \{l \mapsto v_1\}\sigma_1 \vdash E_2 \Rightarrow v, \sigma_2\end{array}}{\rho, \sigma_0 \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v, \sigma_2} \; l \notin \mathrm{Dom}(\sigma_1)$$

$$\frac{\begin{array}{c}\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \\ \{x \mapsto l\}\rho', \{l \mapsto v\}\sigma_2 \vdash E \Rightarrow v', \sigma_3\end{array}}{\rho, \sigma_0 \vdash E_1 \; E_2 \Rightarrow v', \sigma_3} \; l \notin \mathrm{Dom}(\sigma_2)$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \quad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2}{\rho, \sigma_0 \vdash E_1; E_2 \Rightarrow v_2, \sigma_2}$$

그림 6.4: 의미 구조

이 경우에도 메모리는 주소에서 값으로 가는 함수이다. 메모리 주소들의 집합을 $Loc = \{l_1, l_2, \ldots\}$이라 하고 가능한 모든 메모리들의 집합을 Mem이라고 하자($Mem = Loc \to Val$). 환경은 이전과 달리 변수에서 메모리 주소로 가는 함수로 정의하였다($Env = Var \to Loc$). 따라서 프로그램에서 변수가 의미하는 바는 더 이상 값이 아니라 메모리 주소가 된다. 변수의 값은 변수가 의미하는 메모리 주소에 저장된 값을 뜻한다. 메모리에서 각 주소에 담긴 값은 변경 가능하므로 결국 모든 변수의 값을 변경 가능하게 되었다. 메모리에 저장할 수 있는 값은 정수, 부울, 함수값으로 하였다. 이 언어에서는 메모리 주소가 값에 포함되지 않도록 하였다. 앞의 언어와 달리 메모리 주소를 직접 생성하고 그 주소에 직접 접근할 수

없기 때문이다.

첫 번째 방식과 동일하게 프로그램 실행은 값을 계산하는 것 외에도 메모리 상태를 변화시키게 된다. 즉, 프로그램을 현재 환경과 메모리에서 실행시키면 그 결과로 값이 계산될 뿐 아니라 변경된 메모리도 얻게 된다.

$$\rho, \sigma \vdash E \Rightarrow v, \sigma'$$

의미정의는 그림 6.4과 같다. 이전의 명시적으로 상태 변화를 표현하는 언어와 의미구조 규칙이 동일한 구문들을 생략하면 차이점은 다음 다섯 가지이다.

- 변수:

$$\frac{}{\rho, \sigma \vdash x \Rightarrow \sigma(\rho(x)), \sigma}$$

  이제 변수의 값을 읽으려면 환경과 메모리를 모두 참조해야 한다. 환경은 변수에서 주소로 가는 함수이므로 $\rho(x)$는 변수 x가 의미하는 메모리 주소를 뜻한다. 메모리는 주소에서 값으로 가는 함수이므로 $\sigma(\rho(x))$는 메모리에서 변수 x가 의미하는 주소에 담긴 값을 뜻한다.

- 대입문:

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash x \texttt{ := } E \Rightarrow v, \{\rho(x) \mapsto v\}\sigma_1}$$

  변수 x가 의미하는 메모리 주소에 담긴 값을 식 E의 값으로 변경한다. 식 E의 값을 현재 환경과 메모리에서 계산한 값을

v, 결과 메모리를 σ_1이라고 하자. 대입문 x `:=` E는 값 v를 계산해내고 메모리 σ_1에서 x가 의미하는 주소($\rho(x)$)의 값을 v로 변경하도록($\{\rho(x) \mapsto v\}\sigma_1$) 정의하였다.

변수가 대입문의 왼쪽에서 쓰일 때에는 변수가 가리키는 주소($\rho(x)$)를 의미함을 눈여겨보자. 이전 규칙에서 변수가 식의 위치에서 쓰일 때는 그 주소에 담긴 값을 의미하였던 것과 다르다. 이처럼 두 번째 방안을 따르는 언어(e.g. C, Java)에서는 변수가 사용되는 위치에 따라 변수의 의미가 달라진다. 대입문의 왼쪽에서 쓰일 때의 값(l-value)과 대입문의 오른쪽에서 쓰일 때의 값(r-value)이 다르다. 예를 들어, `x := x`에서 왼쪽의 `x`는 주소를 오른쪽의 `x`는 그 주소에 담긴 값을 의미한다.

- `let`식:

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \quad \{x \mapsto l\}\rho, \{l \mapsto v_1\}\sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v, \sigma_2} \; l \notin \mathrm{Dom}(\sigma_1)$$

이전 언어와 마찬가지로 `let`식의 역할은 새로운 변수를 정의하여 현재 환경을 확장하는 것이다. 차이점은 이제 변수가 값이 아닌 메모리 주소를 뜻한다는 점이다. 먼저 E_1의 값을 계산하여 값 v_1, 메모리 σ_1을 얻는다. 그다음 변수 x를 위한 새로운 메모리 주소 l을 현재 메모리 σ_1에서 할당한다($l \notin$

$\mathrm{Dom}(\sigma_1)$). 그다음 변수 x가 주소 l을 의미하도록 현재 환경을 확장한다($\{x \mapsto l\}\rho$). 메모리를 확장하여 E_1의 값인 v_1이 변수 x가 의미하는 메모리 주소인 l에 저장되도록 한다($\{l \mapsto v_1\}\sigma_1$). 마지막으로 확장된 환경과 메모리에서 식 E_2를 계산한다.

- 함수 호출:

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \qquad \{x \mapsto l\}\rho', \{l \mapsto v\}\sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}\ l \notin \mathrm{Dom}(\sigma_2)$$

식 E_1를 계산하여 호출할 함수의 클로저 (x, E, ρ')와 메모리 σ_1를 계산한다. 그다음 메모리 σ_1에서 E_2를 계산하여 함수의 인자로 전달할 값 v를 계산하고 메모리를 σ_2로 변화시킨다. 이제 함수 몸통식 E를 계산하는데, 그 전에 형식 인자 x를 위해 새로운 메모리 주소 l을 할당한다. 환경 ρ'에서 변수 x가 주소 l을 가지도록 확장하고 현재 메모리 σ_2 역시 확장하여 주소 l에 인자값 v가 담기도록 한다. 이렇게 확장된 환경과 메모리에서 몸통식 E를 계산한 결과가 함수 호출의 결과가 된다.

아래 프로그램을 의미구조에 따라 실행시켜 보자.

```
let f = let cnt = 0
        in fun (x) (cnt := cnt + 1; cnt)
in let a = (f 0)
```

```
in let b = (f 0)
   in a + b
```

1. 먼저 빈 환경($\emptyset$)과 빈 메모리($\emptyset$)에서 첫 번째 `let f = ...`을 실행한다. 이를 위해 먼저 아래 식의 값을 계산해야 한다.

```
let cnt = 0 in fun (x) (cnt := cnt + 1; cnt)
```

변수 `cnt`의 값을 정의한 후의 환경과 메모리는 다음과 같다.

$$\begin{aligned} \rho_1 &= \{\texttt{cnt} \mapsto l_1\} \\ \sigma_1 &= \{l_1 \mapsto 0\} \end{aligned}$$

변수 `cnt`를 위한 주소 l_1을 메모리에 할당하였고 값 0을 그 주소에 저장하였다. 이 환경과 메모리에서 함수식

```
fun (x) (cnt := cnt + 1; cnt)
```

을 계산하면 함수값

$$(\texttt{x}, \texttt{(cnt:=cnt+1;cnt)}, \{\texttt{cnt} \mapsto l_1\})$$

이 만들어지고 메모리 변화는 없다. 이제 `let f = ...` 처리를 위해서 새로운 메모리 주소 l_2를 할당한 후 환경에서 `f`가 l_2를 가리키도록 하고 메모리에서 l_2가 함수값을 가지도록 하

면 다음과 같이 환경과 메모리가 갱신된다.

$$\begin{aligned}\rho_2 &= \{\texttt{f} \mapsto l_2\} \\ \sigma_2 &= \{l_1 \mapsto 0, l_2 \mapsto (\texttt{x}, \texttt{(cnt:=cnt+1;cnt)}, \{\texttt{cnt} \mapsto l_1\})\}\end{aligned}$$

현재 환경 ρ_2에는 변수 cnt에 대한 정보가 없음을 눈여겨보자. cnt는 함수값에 저장된 환경에만 존재하기 때문에 함수 내에서만 접근 가능한 변수가 되었다.

2. 이제 환경 ρ_2와 메모리 σ_2에서 `let a = (f 0)`을 계산한다. 함수 호출식을 계산하기 위해서 먼저 새 주소 l_3를 메모리 σ_2에서 할당하고 인자값인 0을 이 주소에 저장해둔다. 따라서 함수 몸통식 `(cnt:=cnt+1;cnt)`을 계산할때의 환경과 메모리는 다음과 같다.

$$\begin{aligned}\rho_3 &= \{\texttt{x} \mapsto l_3, \texttt{cnt} \mapsto l_1\} \\ \sigma_3 &= \{l_1 \mapsto 0, l_2 \mapsto (\texttt{x}, \texttt{(cnt:=cnt+1;cnt)}, \rho_1), l_3 \mapsto 0\}\end{aligned}$$

함수 몸통의 첫 번째 식인 대입문(`cnt:=cnt+1`)을 계산하면 메모리가 아래와 같이 변경된다.

$$\{l_1 \mapsto 1, l_2 \mapsto (\texttt{x}, \texttt{(cnt:=cnt+1;cnt)}, \rho_1), l_3 \mapsto 0\}$$

몸통의 두 번째 식은 cnt이므로 환경에서 변수 cnt가 의미하는 주소 l_1을 가져오고 메모리에서 주소 l_1에 저장된 값 1을 가져온다. 함수 호출이 끝난 후의 환경은 다음과 같다(l_4

는 변수 a를 정의하기 위해 할당한 새로운 주소이다).

$$\{\texttt{f} \mapsto l_3, \texttt{a} \mapsto l_4\}$$

메모리는 다음과 같다.

$$\{l_1 \mapsto 1, l_2 \mapsto (\texttt{x}, \texttt{(cnt:=cnt+1;cnt)}, \rho_1), l_3 \mapsto 0, l_4 \mapsto 1\}$$

3. 비슷하게 두 번째 let식도 계산할 수 있고 그 결과 b의 값은 2가 된다. 따라서 a+b의 값은 3이 된다.

6.2.3 함수 호출 방식

프로그래밍 언어에서 메모리 상태 변화가 가능해지면서 함수를 주소로 호출(call-by-reference)할 수 있게 된다.

값으로 호출하기(Call-by-value)

지금까지는 함수를 값으로 호출(call-by-value)하였다. 6.2절에서 정의한 함수 호출의 의미를 다시 생각해보자.

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \qquad \{x \mapsto l\}\rho', \{l \mapsto v\}\sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}\ l \notin \mathrm{Dom}(\sigma_2)$$

함수가 호출되어 몸통식 E를 계산할 때, 함수의 형식 인자(x)가 의미하는 메모리 주소(l)에 실제 인자로 주어진 식(E_2)의 값(v)을 저장하고 있다. 이러한 함수 인자 전달 방식을 값에 의한 호출(call-by-value)라고 부른다. 여기서 핵심은 함수의 인자를 넘길 때 인자 값을 저장할 메모리를 새롭게 할당해서 그 곳에 값을 복사하여 저장한다는 것이다. 새로운 주소를 만들어 값을 전달하였기 때문에 함수 몸통에서 인자의 값을 변경하여도 함수 호출 지점에서의 인자의 값이 변경되지 않는다. 예를 들어, 아래 프로그램을 보자.

```
let f = fun (x) (x := 1; 1)
in let a = 2
   in let b = (f a)
       in a + b
```

함수 `f`는 인자 `x`의 값을 1로 변경한 후 1을 반환하는 함수이다. 함수 `f`를 인자 `a`로 호출하고 있는데 `a`의 값을 전달하고 함수 내에서는 `x`가 새로운 주소를 가리키고 그 주소에 전달된 값을 저장하게 된다. 함수 몸통에서 `x`의 값을 1로 변경하더라도 새로 할당된 메모리 주소에 저장된 값이 변경될 뿐 호출 지점의 `a`값이 변경되지 않는다. 따라서 `a + b`의 값은 3이 된다.

주소로 호출하기(Call-by-reference)

주소 전달 호출 방식에서는 새로운 주소를 할당하고 인자값을 복사하는 대신 인자값이 저장되어 있는 주소를 넘긴다. 예를 들어, 위의 예제를 주소 전달 호출로 함수 `f`를 호출하면 `a`의 주소를 인자

로 넘기게 되고 인자 x가 a와 동일한 주소를 의미하게 된다. 따라서 함수 몸통에서 x를 변경하면 a의 값이 변경된다.

값 전달 호출외에도 주소 전달 호출을 지원하기 위해서 다음과 같이 언어의 문법구조를 확장하자.

$$
\begin{array}{rcll}
E & \rightarrow & \ldots & \\
 & | & E_1\ E_2 & \text{값 전달 호출} \\
 & | & E\ \langle y \rangle & \text{주소 전달 호출}
\end{array}
$$

값 전달 호출($E_1\ E_2$)과 구별하기 위해서 주소 전달 호출을 의미할 때에는 다른 문법($E\ \langle y \rangle$)을 사용하기로 하자. 주소 전달 호출은 인자로 변수가 주어지는 경우에만 동작한다. 인자로 주소를 전달할 수 있어야 하는데, 현재 우리 언어에서는 변수만 주소를 뜻하기 때문이다. 따라서 언어의 문법구조에서 주소 전달 호출을 할 때에는 함수의 인자로 임의의 식이 올 수 없고 변수만 올 수 있도록 강제하였다. 예를 들어, 위 예제 프로그램에서 함수 f를 주소 전달 호출로 부르려면 다음과 같이 프로그램을 작성한다.

```
let f = fun (x) (x := 1; 1)
in let a = 2
   in let b = (f <a>)
       in a + b
```

주소 전달 호출의 의미는 다음과 같다.

$$
\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \{x \mapsto \rho(y)\}\rho', \sigma_1 \vdash E \Rightarrow v', \sigma_2}{\rho, \sigma_0 \vdash E_1\ \langle y \rangle \Rightarrow v', \sigma_2}
$$

호출할때 함수의 형식 인자 x가 인자로 주어진 변수가 뜻하는 주소를 가리키도록 변경한다는 점이 값 전달 호출과 다르다. 호출 지점에서 실제 인자의 값을 계산할 필요없이 변수의 주소만 넘기고 있다.

프로그래밍 언어가 주소 전달 호출을 지원하면서 서로 다른 두 변수가 같은 주소를 의미하는 것이 가능해진다. 예를 들어 아래 프로그램을 생각하자.

```
let a = 1
in let f = fun (x) (fun (y) (y := 2; x + y))
   in ((f <a>) <a>)
```

함수 몸통에서 변수 x, y, a는 모두 같은 주소를 의미하게 된다. 따라서 대입문 y := 2에 의해 y의 값을 변경하면 x와 a가 의미하는 값도 모두 2가 된다. 주소 전달 호출로 인해서 변수 x, y, a가 모두 동일한 메모리 주소를 뜻하는 별명(aliasing)이 된 셈이다. 프로그램에서 별명을 만들 수 있게 되면 프로그램의 실행을 예측하기가 어려워진다. 겉으로 보기에 어떤 변수의 값을 바꾸는 것처럼 보여도 전혀 상관 없어 보이는 다른 변수의 값이 변경되어 있는 일이 일어날 수 있으므로, 발생 가능한 모든 상황을 점검해보아야 하는 것이다. 어떤 대입문이 어떤 변수의 값에 영향을 주는지를 생각할 때 이전에는 정의되는 변수만 생각하면 충분했지만 이제는 겉으로 보기에 상관없어 보이는 변수들도 값이 바뀔 수 있음을 고려해야 하기 때문에 훨씬 복잡해졌다.

지연 계산법 (Lazy evaluation)

참고로 함수 호출 방식의 다른 축으로 함수 인자의 계산을 최대한 미루는 지연 계산법 또는 소극적인 계산법(lazy evaluation)이라 불리우는 방식이 있다. 앞에서 살펴본 두 호출 방식은 모두 함수 호출전에 인자를 계산하는 적극적인 방식(eager evaluation)이었다. 하지만 지연 계산법에서는 함수의 인자가 사용되는 시점에 계산된다. 따라서 인자가 함수의 몸통에서 사용되지 않으면 그 값이 계산되지 않는다. 예를 들어 아래 프로그램을 보자.

```
letrec forever(x) = (forever x)  (* infinite loop *)
in (fun x (1)) (forever 0)
```

인자의 값에 상관없이 1을 반환하는 함수에 인자 `(forever 0)`을 주어 호출하였다. 함수 `f`를 호출하기 전에 인자값을 계산하면 위 프로그램은 끝나지 않는다. 하지만 지연 계산법으로 위 프로그램을 실행하면 인자 `(forever 0)`을 먼저 계산하지 않고 호출된 함수의 몸통인 `1`을 실행한다. 함수의 몸통에서 인자 `x`를 사용하지 않으므로 식 `(forever 0)`은 계산되지 않고, 위 프로그램은 종료하게 된다. 이처럼 적극적인 계산법(eager evaluation)에서는 끝나지 않는 프로그램이 소극적인 계산법(lazy evaluation)에서는 끝날 수 있다.

지연 계산법은 Haskell과 같은 함수형 언어에서 주로 사용되지만 명령형 언어에서는 잘 사용되지 않는다. 인자값이 계산되는 시점이 동적으로 변하게 되면서 프로그램 구문들이 실행 되는 순서를 예상하기 어려워지기 때문이다. 특히 계산 순서에 따라 프로그램의

의미가 달라지는 명령형 언어에서 프로그램의 복잡도를 증가시킬 수 있다.

6.3 구현

이 장에서 정의한 언어의 실행기를 구현해보자.

첫 번째 방식

먼저 첫 번째 방식의 언어를 OCaml로 다음과 같이 정의할 수 있다.

```
type program = exp
and exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
  | DIV of exp * exp
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
  | NEWREF of exp (* memory allocation *)
  | DEREF of exp  (* dereference *)
  | SETREF of exp * exp (* assignment *)
  | SEQ of exp * exp (* sequence *)
and var = string
```

값, 환경, 메모리를 다음과 같이 구현하자.

```
type value =
    Int of int
  | Bool of bool
  | Loc of loc
  | Closure of var * exp * env
  | RecClosure of var * var * exp * env
and env = var -> value
and loc = int
and mem = loc -> value

let empty_env =
  fun x -> raise (Failure "Environment is empty")
let extend_env (x,v) e =
  fun y -> if x = y then v else (e y)
let apply_env e x = e x

let empty_mem =
  fun _ -> raise (Failure "Memory is empty")
let extend_mem (l,v) m =
  fun y -> if l = y then v else (m y)
let apply_mem m l = m l

let counter = ref 0
let new_location () = counter:=!counter+1;!counter
```

메모리 주소를 정수로 표현하였고 `new_location()`을 호출할 때마다 새로운 메모리 주소를 반환하도록 하였다. 아래 함수 타입을 가지는 위 언어에 대한 실행기를 구현해보자

```
eval : exp -> env -> mem -> value * mem
```

두 번째 방식

두 번째 방식의 언어는 다음과 같이 정의할 수 있다.

```
type program = exp
and exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
  | DIV of exp * exp
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp    (* call-by-value *)
  | CALLREF of exp * var (* call-by-reference *)
  | SET of var * exp
  | SEQ of exp * exp
and var = string
```

값, 환경, 메모리를 다음과 같이 구현하자.

```
type value =
    Int of int
  | Bool of bool
  | Closure of var * exp * env
  | RecClosure of var * var * exp * env
and env = var -> loc
and loc = int
and mem = loc -> value

let empty_env =
```

```
  fun x -> raise (Failure "Environment is empty")
let extend_env (x,v) e =
  fun y -> if x = y then v else (e y)
let apply_env e x = e x

let empty_mem =
  fun _ -> raise (Failure "Memory is empty")
let extend_mem (l,v) m =
  fun y -> if l = y then v else (m y)
let apply_mem m l = m l

let counter = ref 0
let new_location () = counter:=!counter+1;!counter
```

아래 함수 타입을 가지는 위 언어에 대한 실행기를 구현해보자

```
eval : exp -> env -> mem -> value * mem
```

7

포인터와 메모리 관리

이 장에서는 포인터와 레코드를 사용할 수 있고 메모리 관리를 할 수 있도록 언어를 확장해본다. 그림 7.1의 언어를 대상으로 하자. 변수의 값을 변경할 수 있고 값 전달 및 주소 전달에 기반한 함수 호출을 지원하는 언어이다. 의미 공간은 다음과 같다.

$$\begin{aligned} \mathit{Val} &= \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} \\ \mathit{Procedure} &= \mathit{Var} \times E \times \mathit{Env} \\ \rho \in \mathit{Env} &= \mathit{Var} \to \mathit{Loc} \\ \sigma \in \mathit{Mem} &= \mathit{Loc} \to \mathit{Val} \end{aligned}$$

주요 실행 규칙은 그림 7.2와 같다.

7.1 레코드

명령형 언어에서 레코드(record)[1]는 이름 지은 메모리 주소들의 모음을 가리킨다. 예를 들어, 아래 프로그램을 보자.

```
let person = { age := 2, birth := 201803 }
in  person.age + person.birth
```

1) C 언어에서는 구조체(structure)라고 부른다.

E	$\rightarrow$	n	정수
	$\mid$	x	변수
	$\mid$	E `+` E	사칙연산
	$\mid$	`iszero` E	테스트
	$\mid$	`if` E_1 `then` E_2 `else` E_3	조건식
	$\mid$	`let` $x = E_1$ `in` E_2	변수 정의
	$\mid$	`fun` x E	함수 정의
	$\mid$	E_1 E_2	값 전달 함수 호출
	$\mid$	E $\langle y \rangle$	주소 전달 함수 호출
	$\mid$	x `:=` E	대입문
	$\mid$	$E_1; E_2$	순서문

그림 7.1: 대상 언어의 문법구조

레코드 `person`을 `age`와 `birth`로 이름 지은 두 메모리 주소들의 모음으로 정의하였다. 레코드 필드(field)는 `person.age`와 같이 접근할 수 있다.

참고로 명령형 언어에서 배열(array)은 메모리 주소들의 이름을 자연수로 붙인 레코드로 이해할 수 있다. 예를 들어, 아래 예제를 보자.

```
let arr[3] = { 1, 2, 3 }
in  arr[0] + arr[1] + arr[2]
```

배열 `arr`은 메모리 주소 세 개의 모음이며, 각 주소들을 `arr[0]`, `arr[1]`, `arr[2]`와 같이 자연수 인덱스를 써서 접근하고 있다.

$$\frac{}{\rho, \sigma \vdash n \Rightarrow n, \sigma} \qquad \frac{}{\rho, \sigma \vdash x \Rightarrow \sigma(\rho(x)), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \quad \{x \mapsto l\}\rho, \{l \mapsto v_1\}\sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let}\ x\ \texttt{=}\ E_1\ \texttt{in}\ E_2 \Rightarrow v, \sigma_2}\ l \notin \mathrm{Dom}(\sigma_1)$$

$$\frac{}{\rho, \sigma \vdash \texttt{fun}\ x\ E \Rightarrow (x, E, \rho), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash x\ \texttt{:=}\ E \Rightarrow v, \{\rho(x) \mapsto v\}\sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \quad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \quad \{x \mapsto l\}\rho', \{l \mapsto v\}\sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}\ l \notin \mathrm{Dom}(\sigma_2)$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \quad \{x \mapsto \rho(y)\}\rho', \sigma_1 \vdash E \Rightarrow v', \sigma_2}{\rho, \sigma_0 \vdash E_1\ \langle y \rangle \Rightarrow v', \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \quad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2}{\rho, \sigma_0 \vdash E_1; E_2 \Rightarrow v_2, \sigma_2}$$

그림 7.2: 대상 언어의 의미구조

문법구조

레코드를 추가하기 위하여 언어의 문법구조를 그림 7.3과 같이 확장하자. 레코드를 만들고 사용하는 문법들을 추가하였다. 레코드는 임의의 개수의 필드를 가질 수 있지만 의미구조 정의의 편의를 위해서 필드가 없거나 두 개인 경우만 생각하자. $E.x$는 E가 뜻하는 레코드의 필드 x의 값을 의미한다. $E_1.x$:= E_2는 레코드 필드 $E_1.x$가 의미하는 메모리 주소에 담긴 값을 E_2의 값으로 변경하는

$$
\begin{array}{rcll}
E & \to & \vdots & \\
 & | & \{\} & \text{빈 레코드} \\
 & | & \{x \texttt{ := } E_1, y \texttt{ := } E_2\} & \text{레코드} \\
 & | & E.x & \text{필드 접근} \\
 & | & E_1.x \texttt{ := } E_2 & \text{필드 값 변경}
\end{array}
$$

그림 7.3: 레코드를 위한 문법구조 확장

레코드 대입문이다.

의미구조

의미공간을 다음과 같이 확장하자. 이제 레코드가 값이 될 수 있으므로 값 집합에 레코드를 포함시켰다.

$$
\begin{aligned}
Val &= \mathbb{Z} + Bool + Procedure + Record \\
Procedure &= Var \times E \times Env \\
r \in Record &= Field \to Loc \\
\rho \in Env &= Var \to Loc \\
\sigma \in Mem &= Loc \to Val
\end{aligned}
$$

레코드(r)는 이름을 붙인 메모리 주소들의 모음이므로 필드 이름에서 주소로 가는 함수로 정의하였다. 즉, 레코드 값은 다음과 같이 생겼다.

$$\{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\}$$

각 레코드 필드 x_i를 메모리 주소 l_i로 매핑하는 함수이다.

의미구조 정의는 그림 7.4와 같다. 첫 번째 규칙에서 빈 레코드

$$\frac{}{\rho, \sigma \vdash \{\} \Rightarrow \emptyset, \sigma}$$

$$\frac{\rho, \sigma \vdash E_1 \Rightarrow v_1, \sigma_1 \quad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2 \quad l_1, l_2 \notin \mathrm{Dom}(\sigma_2)}{\rho, \sigma \vdash \{x \texttt{ := } E_1, y \texttt{ := } E_2\} \Rightarrow \left\{ \begin{array}{l} x \mapsto l_1, \\ y \mapsto l_2 \end{array} \right\}, \{l_1 \mapsto v_1, l_2 \mapsto v_2\}\sigma_2}$$

$$\frac{\rho, \sigma \vdash E \Rightarrow r, \sigma_1}{\rho, \sigma \vdash E.x \Rightarrow \sigma_1(r(x)), \sigma_1}$$

$$\frac{\rho, \sigma \vdash E_1 \Rightarrow r, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma \vdash E_1.x \texttt{ := } E_2 \Rightarrow v, \{r(x) \mapsto v\}\sigma_2}$$

그림 7.4: 레코드를 위한 의미구조 확장

{}는 아무것도 정의되지 않은 함수, 즉 빈 레코드 값 $\emptyset$를 만들어 낸다.

두 번째 규칙은 필드를 가지고 있는 레코드 $\{x \texttt{ := } E_1, y \texttt{ := } E_2\}$의 값을 만들어내는 규칙이다. 먼저 식 E_1과 E_2의 값을 계산하여 값 v_1과 v_2를 만든다. 그 후의 메모리를 σ_2라고 하면, σ_2에서 각 필드의 값을 저장할 메모리 주소들인 l_1, l_2를 새롭게 할당한다($l_1, l_2 \notin \mathrm{Dom}(\sigma_2)$). 최종적으로 레코드 값 $\{x \mapsto l_1, y \mapsto l_2\}$를 만들어내고, 메모리 σ_2의 l_1, l_2 주소에 값 v_1, v_2를 저장한다.

세 번째 규칙은 레코드의 필드($E.x$)가 의미하는 값을 가져오는 방법을 정의하고 있다. 먼저 E를 계산하여 레코드 값 r을 얻는다. 필드 x의 값을 가지고 오기 위해서 먼저 필드 x가 의미하는 주소인 $r(x)$를 찾는다. 그 다음 현재 메모리에서 그 주소에 담긴 값을 찾는다($\sigma_1(r(x))$).

마지막 규칙은 레코드 필드값을 갱신하는 규칙이다. 먼저 E_1을

계산하면 레코드 값(r)이 나와야 한다. 그 다음 E_2의 값 v를 계산하여 현재 메모리 σ_2에서 레코드 필드 x의 주소 $r(x)$가 값 v를 가리키도록 값을 변경한다. 식 $E_1.x$:= E_2의 값은 v로 정의하였다.

예를 들어, 아래 프로그램이 의미구조 규칙에 따라 실행되는 과정을 살펴보자. 레코드를 이용하여 나무 구조를 표현하는 프로그램이다.

```
let tree = { left := {}, v := 1, right := {} }
in  tree.right := { left := {}, v := 2, right := 3 }
```

먼저 빈 환경과 메모리에서 레코드 생성식

```
{ left := {}, v := 1, right := {} }
```

을 계산하면 아래와 같은 레코드 값이 만들어지며

$$\{\texttt{left} \mapsto l_1, \texttt{v} \mapsto l_2, \texttt{right} \mapsto l_3\}$$

메모리 상태가 다음과 같이 바뀐다.

$$\{l_1 \mapsto \emptyset, l_2 \mapsto 1, l_3 \mapsto \emptyset\}$$

새로운 메모리 주소 l_1, l_2, l_3를 할당하였고 각 필드가 이들을 가리키도록 레코드 값을 생성하였다. 메모리의 주소 l_1과 l_3에는 빈 레코드 값($\emptyset$)이 저장되고, 필드 v가 의미하는 주소 l_2에는 정수값 1이 저장된다. 변수 tree를 레코드 값 r로 정의했으므로 let tree = ...

를 지난 후 환경과 메모리는 다음과 같다.

$$\begin{aligned} \rho &= \{\texttt{tree} \mapsto l_4\} \\ \sigma &= \{l_1 \mapsto \emptyset, l_2 \mapsto 1, l_3 \mapsto \emptyset, l_4 \mapsto \left\{ \begin{array}{rcl} \texttt{left} & \mapsto & l_1, \\ \texttt{v} & \mapsto & l_2, \\ \texttt{right} & \mapsto & l_3 \end{array} \right\}\} \end{aligned}$$

환경에서 변수 tree는 메모리 주소 l_4를 의미하고, 메모리 주소 l_4에는 레코드 값이 담겨 있다. 레코드의 필드들은 각각 주소 l_1, l_2, l_3를 의미하고, 그 주소에는 각각 값 $\emptyset, 1, \emptyset$가 담겨 있다. 이러한 환경과 메모리에서 레코드 대입문

```
tree.right := { left := {}, v := 2, right := 3 }
```

을 계산해보자. 이를 위해 먼저 tree.right가 의미하는 메모리 주소를 계산해야 한다. 현재 환경과 메모리에서 변수 tree가 의미하는 값을 찾으면 레코드 $\{\texttt{left} \mapsto l_1, \texttt{v} \mapsto l_2, \texttt{right} \mapsto l_3\}$가 계산된다. 이 레코드에서 필드 right가 의미하는 주소인 l_3에 레코드 생성식

```
{ left := {}, v := 2, right := 3 }
```

을 계산하여 만들어지는 값을 저장하면 된다. 위 레코드 생성식을 계산하면 다음과 같이 새로운 메모리 주소와 함께 레코드 값이 만들어진다.

$$\{\texttt{left} \mapsto l_5, \texttt{v} \mapsto l_6, \texttt{right} \mapsto l_7\}$$

이 값이 전체 프로그램의 값으로 반환되며, 실행이 끝난 후의 환경과 메모리 상태는 아래와 같다.

$$
\begin{aligned}
\rho &= \{\texttt{tree} \mapsto l_4\} \\
\sigma &= \left\{
\begin{array}{l}
l_1 \mapsto \emptyset, \\
l_2 \mapsto 1, \\
l_3 \mapsto \{\texttt{left} \mapsto l_5, \texttt{v} \mapsto l_6, \texttt{right} \mapsto l_7\}, \\
l_4 \mapsto \{\texttt{left} \mapsto l_1, \texttt{v} \mapsto l_2, \texttt{right} \mapsto l_3\}, \\
l_5 \mapsto \emptyset, \\
l_6 \mapsto 2, \\
l_7 \mapsto 3
\end{array}
\right\}
\end{aligned}
$$

메모리 주소들 사이의 링크 구조를 따라 만들어지는 나무 구조가 메모리 내에 만들어졌고 이를 변수 `tree`로부터 접근할 수 있게 되었다.

7.2 포인터

명령형 언어에서 포인터 변수는 메모리 주소를 값으로 가지는 변수이다. 포인터를 지원하기 위해서 프로그래밍 언어는 주소값을 생성하는 문법과 포인터 변수의 값을 참조하는 문법을 제공해야 한다.

예를 들어 아래 프로그램들을 보자.

- ```
  let x = 1 in
    let y = &x in
      *y := *y + 2
  ```
  ```

변수 y를 변수 x의 주소(&x)를 가리키는 포인터로 정의하였다. 포인터 변수를 참조(*y)하면 y가 가리키는 메모리 주소에 담긴 값, 즉 변수 x에 담긴 값을 의미한다. 따라서 *y + 2의 값은 3이 된다. 대입문 *y := *y + 2는 포인터 y가 가리키는 주소에 담긴 값을 3으로 변경하게 된다. 따라서 위 프로그램을 실행한 후의 x값은 3으로 바뀌게 된다.

- ```
  let x = { left := {}, v := 1, right := {} } in
    let y = &x.v
      *y := *y + 2
  ```

  변수 y를 구조체 필드 x.v의 주소를 가리키는 포인터로 정의하였다. 마지막 구문 (*y := *y + 2)을 실행하면 x.v의 값이 3으로 바뀐다.

- ```
  let f = fun (x) (*x := *x + 1) in
    let a = 1 in
      (f &a); a
  ```

  함수 f의 인자 x는 포인터 변수이고 함수 몸통식에서 x가 가리키는 주소에 담긴 값을 1 증가시키고 있다. 변수 a의 초기값이 1이고 함수 f의 인자로 a의 주소값을 넘기고 있으므로 최종적으로 a의 값은 2가 된다.

- ```
  let f = fun (x) (&x) in
    let p = (f 1) in
      *p + 2
  ```

  함수 f는 인자 x의 주소를 반환하는 함수이다. 함수 호출 f 1을 실행하면 인자 x의 주소가 반환되고 포인터 p가 그 주소
  ```

를 가리키게 된다. 따라서 프로그램을 실행한 결과는 3이 된다.[2)]

이처럼 프로그래밍 언어가 포인터를 지원하면 메모리 주소가 프로그램에서 가장 자유롭게 사용할 수 있는 값(first-class value)이 된다. 즉, 메모리 주소를 변수에 저장할 수 있고, 함수의 인자로 전달할 수 있고, 함수로부터 반환할 수 있게 된다.

문법구조

포인터를 지원하기 위해서 언어를 다음과 같이 확장하자.

$$
\begin{array}{rcl}
E & \rightarrow & \cdots \\
& | & \texttt{new}\ E \\
& | & \texttt{\&}x \\
& | & \texttt{\&}E.x \\
& | & \texttt{*}E \\
& | & \texttt{*}E_1\ \texttt{:=}\ E_2
\end{array}
$$

처음 세 구문은 메모리 주소를 만드는 식들이다. `new` E는 새로운 메모리 주소를 할당하고 그 주소에 E를 저장한다. `&`x, `&`$E.x$는 메모리 주소의 이름으로부터 그 주소를 가져온다. `*`E, `*`E_1 `:=` E_2는 메모리 주소에 저장된 값을 가져오거나 그 주소에 담긴 값을 변경한다.

2) 함수가 반환될 때 접근 불가능한 메모리가 자동으로 해제되지 않는다고 가정하였다.

의미구조

의미공간은 다음과 같다. 주소값(Loc)이 값에 포함되었다.

$$\begin{aligned} Val &= \mathbb{Z} + Bool + Procedure + Record + Loc \\ Procedure &= Var \times E \times Env \\ r \in Record &= Field \to Loc \\ \rho \in Env &= Var \to Loc \\ \sigma \in Mem &= Loc \to Val \end{aligned}$$

의미구조 정의는 다음과 같다.

$$\frac{\rho, \sigma \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma \vdash \texttt{new}\ E \Rightarrow l, \{l \mapsto v\}\sigma_1}\ l \notin \mathrm{Dom}(\sigma_1)$$

$$\frac{}{\rho, \sigma \vdash \texttt{\&}x \Rightarrow \rho(x), \sigma}$$

$$\frac{\rho, \sigma \vdash E \Rightarrow r, \sigma_1}{\rho, \sigma \vdash \texttt{\&}E.x \Rightarrow r(x), \sigma_1}$$

$$\frac{\rho, \sigma \vdash E \Rightarrow l, \sigma_1}{\rho, \sigma \vdash *E \Rightarrow \sigma_1(l), \sigma_1}$$

$$\frac{\rho, \sigma \vdash E_1 \Rightarrow l, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma \vdash *E_1\ \texttt{:=}\ E_2 \Rightarrow v, \{l \mapsto v\}\sigma_2}$$

첫 번째 규칙은 메모리에서 새로운 주소 l을 할당하고 그 주소에 식 E의 값을 저장하고 있다. 메모리 할당식 `new` E의 값은 새롭게 할당된 주소값 l이다. 두 번째, 세 번째 규칙은 변수(x)와 레코드 필드($E.x$)가 나타내는 메모리 주소를 가져온다. 변수의 경우 환경

에서, 레코드 필드의 경우에는 레코드 값에서 찾으면 된다. 네 번째 규칙은 E가 메모리 주소를 나타낼 때 그 주소에 담긴 값을 가져오는 방법을 정의하고 있다. 먼저 E를 계산하면 주소값 l이 나와야 하고, 메모리에서 l의 값을 찾으면 된다. 마지막 규칙은 E_1이 나타내는 메모리 주소에 담긴 값을 E_2의 값으로 변경하는 규칙이다. E_1을 계산하면 주소값 l이 나와야 하며, 메모리 σ_2에서 l의 값을 E_2의 값인 v로 변경하고 있다.

위 의미구조에서 포인터 참조식 $*E$의 의미가 사용되는 위치에 따라 달라지고 있다. 마지막 규칙은 대입문의 왼쪽에서 참조식 $*E$가 사용되는 경우이다. 이 경우 $*E$는 E가 의미하는 주소값을 뜻한다. 네 번째 규칙은 $*E$ 자체가 식으로 사용되는 경우이다. 이때 $*E$는 E가 의미하는 주소에 담긴 값을 뜻한다. 동일하게 생긴 식 $(*E)$이라고 해도 사용되는 위치에 따라 그 의미는 다를 수 있다.

예제 1 다음 프로그램이 실행되는 과정을 살펴보자.

```
let x = 1 in
  let y = &x in
    *y := *y + 2
```

두 `let`식들을 실행한 후의 환경과 메모리를 $\rho = \{\mathtt{x} \mapsto l_1, \mathtt{y} \mapsto l_2\}$, $\sigma = \{l_1 \mapsto 1, l_2 \mapsto l_1\}$이라고 하면 마지막 대입문이 실행되는 과정

은 다음과 같다.

$$\dfrac{\rho,\sigma \vdash \texttt{y} \Rightarrow l_1,\sigma \quad \dfrac{\rho,\sigma \vdash \texttt{y} \Rightarrow l_1,\sigma}{\rho,\sigma \vdash \texttt{*y} \Rightarrow 1,\sigma} \quad \rho,\sigma \vdash \texttt{2} \Rightarrow 2,\sigma}{\rho,\sigma \vdash \texttt{*y := *y + 2} \Rightarrow 3,\{l_1 \mapsto 3\}\sigma}$$

예제 2 다음 프로그램이 실행되는 과정을 살펴보자.

```
let f = proc (x) (*x := *x + 1) in
  let a = 1 in
    (f &a)
```

두 `let`식들을 실행한 후의 환경과 메모리를 $\rho = \{\texttt{f} \mapsto l_1, \texttt{a} \mapsto l_2\}$, $\sigma = \{l_1 \mapsto (\texttt{x}, \texttt{*x:=*x+1}, \emptyset), l_2 \mapsto 1\}$이라고 하자. 여기서 함수 호출 `(f &a)`을 하면 함수값과 인자를 다음과 같이 계산하게 되고

$$\dfrac{}{\rho,\sigma \vdash \texttt{f} \Rightarrow (\texttt{x}, \texttt{*x:=*x+1}, \emptyset),\sigma} \qquad \dfrac{}{\rho,\sigma \vdash \texttt{\&a} \Rightarrow l_2,\sigma}$$

환경 $\emptyset$과 메모리 σ를 인자에 대해서 다음과 같이 확장한 후

$$\rho = \{\texttt{x} \mapsto l_3\}, \qquad \sigma = \{l_3 \mapsto l_2, l_1 \mapsto (\texttt{x}, \texttt{*x:=*x+1}, \emptyset), l_2 \mapsto 1\}$$

다음과 같이 함수 몸통을 계산하게 된다.

$$\dfrac{\rho,\sigma \vdash \texttt{x} \Rightarrow l_2,\sigma \quad \dfrac{\rho,\sigma \vdash \texttt{x} \Rightarrow l_2,\sigma}{\rho,\sigma \vdash \texttt{*x} \Rightarrow 1,\sigma} \quad \rho,\sigma \vdash \texttt{1} \Rightarrow 1,\sigma}{\rho,\sigma \vdash \texttt{*x:=*x+1} \Rightarrow 2,\{l_2 \mapsto 2\}\sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \quad \{x \mapsto l\}\rho, \{l \mapsto v_1\}\sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v, \sigma_2} \; l \notin \mathrm{Dom}(\sigma_1)$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \quad \{x \mapsto l\}\rho', \{l \mapsto v\}\sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1 \;\; E_2 \Rightarrow v', \sigma_3} \; l \notin \mathrm{Dom}(\sigma_2)$$

$$\frac{\rho, \sigma \vdash E_1 \Rightarrow v_1, \sigma_1 \quad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2 \quad l_1, l_2 \notin \mathrm{Dom}(\sigma_2)}{\rho, \sigma \vdash \{x \texttt{ := } E_1, y \texttt{ := } E_2\} \Rightarrow \{ \begin{array}{c} x \mapsto l_1, \\ y \mapsto l_2 \end{array} \}, \{l_1 \mapsto v_1, l_2 \mapsto v_2\}\sigma_2}$$

$$\frac{\rho, \sigma \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma \vdash \texttt{new } E \Rightarrow l, \{l \mapsto v\}\sigma_1} \; l \notin \mathrm{Dom}(\sigma_1)$$

그림 7.5: 메모리를 소모하는 실행 규칙들

7.3 메모리 관리

지금까지 정의한 의미구조를 보면 메모리가 새롭게 할당되는 곳이 네 경우가 있다. `let`식, 함수 호출식, 레코드 생성식, 메모리 할당식을 계산할 때마다 새로운 메모리 주소를 할당한다(그림 7.5).

하지만 지금까지 정의한 의미구조를 보면 프로그램 실행중에 메모리가 할당된 후 해제되는 경우가 없다. 따라서 언젠가는 메모리가 가득차게 되고 프로그램 실행이 더 이상 불가능해지는 순간이 오게 된다. 예를 들어, 다음과 같이 재귀함수를 작성하면 영원히 돌지 않고 메모리 부족으로 프로그램이 비정상 종료하게 된다.[3)]

```
letrec forever(x) = (forever x) in (forever (new 0))
```

3) 이 장의 언어 정의에서 재귀함수를 고려하지 않았지만 쉽게 확장 가능하다.

메모리 주소를 재활용할 수 있도록 언어를 확장하자. 대부분의 프로그래밍 언어들은 아래 두 가지 방법중 하나를 택한다.

1. 수동 메모리 재활용: 프로그래머에게 메모리 관리를 맡기는 것이다. 주로 시스템 프로그래밍 언어들 (C, C++)이 그렇다. 이들 언어에서는 메모리 관리를 직접 하기 때문에 좀 더 세밀하게 프로그래머가 원하는대로 메모리를 관리하는 것이 가능하다. 하지만 메모리를 수동으로 관리하는 것은 매우 어렵고 여러가지 다른 문제를 야기할 수 있다.

2. 자동 메모리 재활용: Java와 같은 언어들은 메모리 재활용을 자동으로 지원한다. 프로그래머는 메모리를 할당하기만 하고 해제는 프로그래밍 언어가 자동으로 해 주는 것이다. 메모리 관리 실수로 인한 오류가 일어나지 않는다는 것은 장점이다. 하지만 프로그래머가 직접 관리하는 것보다 비효율적일 수 있고 프로그램 실행 중간에 메모리 재활용을 수행해야 하므로 부담이 따른다.

7.3.1 수동 메모리 재활용

수동 메모리 재활용은 간단하다. 언어에서 메모리 해제문을 지원하면 된다.

$$\begin{aligned} E &\rightarrow \cdots \\ &\mid \texttt{free}\ E \end{aligned}$$

`free` E는 E가 의미하는 주소를 현재 메모리에서 삭제하는 명령이다. 다음과 같이 의미를 정의할 수 있다.

$$\frac{\rho, \sigma \vdash E \Rightarrow l, \sigma_1}{\rho, \sigma \vdash \texttt{free}\ E \Rightarrow l, \sigma_1 \setminus l}\ l \in \mathrm{Dom}(\sigma_1)$$

여기서 $\sigma \setminus l$은 메모리 σ에서 주소 l에 해당하는 엔트리를 삭제한 메모리를 의미한다. 그 결과를 σ'라고 할 때 σ'은 다음 두 조건을 만족하는 메모리이다: 1) $\mathrm{Dom}(\sigma') = \mathrm{Dom}(\sigma) \setminus \{l\}$ (여기서 $A \setminus B$는 집합 A와 B의 차집합을 뜻한다), 2) $\forall l \in \mathrm{Dom}(\sigma') : \sigma'(l) = \sigma(l)$. 메모리 해제문의 값은 주소 l로 정의하였는데, 더 이상 쓰이지 않을 값이므로 사실 아무 값으로 정의해도 무방하다.

수동 메모리 재활용은 이처럼 매우 간단하다. 하지만 모든 메모리 관리를 프로그래머가 직접 하는 것은 어려운 일이다. 세 가지 문제가 있을 수 있다.

- 메모리를 너무 늦게 해제하거나 아예 해제하지 않을 수 있다. 메모리 누수(memory-leak)라고 한다.
- 메모리를 미처 사용하기 전에 해제할 수 있다(use-after-free 또는 dangling pointer 오류).
- 동일한 메모리를 여러번 해제할 수 있다(double-free).

수동 메모리 재활용을 지원하는 언어인 C/C++로 작성된 프로그램에서 이들 세 가지 오류들은 매우 흔하며 소프트웨어 보안 취약점의 주요 원인이 된다.

7.3.2 자동 메모리 재활용

프로그래밍 언어에서 자동으로 메모리 재활용을 지원하는 방법을 고안해보자. 다음과 같은 방식을 생각할 수 있을 것이다.

1. 메모리가 부족해지면 프로그램 실행을 잠시 멈춘다.
2. 현재 메모리에서 앞으로 더 이상 사용되지 않을 메모리 주소들을 모두 모은다.
3. 이렇게 모은 주소들을 현재 메모리에서 모두 삭제한다.

예를 들어 아래 프로그램을 보자.

```
let f = fun (x) (x+1) in
  let a = f 0 in
    a + 1
```

두 번째 `let`의 몸통식(`a + 1`)을 계산하기 직전 환경과 메모리가 다음과 같다고 하자.

$$\rho = \{\texttt{f} \mapsto l_1, \texttt{a} \mapsto l_3\}, \sigma = \{l_1 \mapsto (\texttt{x}, \texttt{x+1}, \emptyset), l_2 \mapsto 0, l_3 \mapsto 1\}$$

이 때, 현재 메모리에 속한 주소들 가운데 앞으로 사용될 것은 변수 `a`가 의미하는 l_3뿐이다. 따라서 σ에서 l_1, l_2를 제거하여 $\sigma' = \{l_3 \mapsto 1\}$와 같이 메모리를 재활용하여 식 `a + 1`을 계산해도 된다.

자동 메모리 재활용의 원리는 이처럼 단순하지만 위의 방식을 일반적인 프로그래밍 언어에 대해서 구현하는 것은 불가능하다. 두 번째 단계, 미래에 사용되지 않을 주소만 정확하게 골라내는 것이

계산 불가능(undecidable)하기 때문이다. 그런 알고리즘이 존재한다면 이를 이용하여 정지 문제(halting problem)를 풀 수 있게 된다. 정지 문제란 입력으로 주어진 프로그램을 실행시키면 유한한 시간에 종료하는지 또는 무한히 도는지를 판단하는 문제로서 알고리즘이 존재할 수 없는, 그래서 컴퓨터로 정확하게 풀 수 없는 대표적인 문제이다. 현 시점 이후로 사용되지 않을 메모리 주소들만 빠짐없이 골라내는 알고리즘이 존재한다고 하고 이를 G라고 하자. 알고리즘 G를 이용하여 정지 문제를 푸는 알고리즘 H를 다음과 같이 만들 수 있다.

1. H는 입력 프로그램 P를 받아서 다음과 같이 새 프로그램을 구성한다.

   `let x = new() in` P`; x`

   변수 `x`에 메모리를 할당하고 프로그램 P를 실행한 후 `x`를 반환하는 프로그램이다. 이 때 변수 `x`는 P에 등장하지 않는 새로운 변수라고 하자.

2. 위 프로그램을 실행하면서 P를 실행하기 직전에 G를 실행시켜서 앞으로 사용되지 않을 메모리 주소들을 모두 모은다. 그 집합을 S라고 하자.

3. 이제 다음과 같이 P에 대한 종료 여부를 판단할 수 있다.

   - 만약 S가 `x`의 주소값을 포함하지 않는다면 P가 끝난다는 뜻이다. P가 끝나야 `x`가 사용되기 때문이다.

- 만약 S가 x의 주소값을 포함한다면, P가 끝나지 않는다는 뜻이다. 그래야 x가 사용되지 않는다.

정지 문제를 푸는 알고리즘 H는 존재하지 말아야 하므로 알고리즘 G가 존재한다고 가정한 것이 잘못되었음을 알 수 있다.

계산 불가능한 문제는 어림잡아 푸는 방법밖에 없다. 따라서 자동 메모리 재활용 기술들은 앞으로 사용되지 않을 메모리 주소를 완벽하게 골라내는 것을 포기하고, 어림잡는다. 가장 많이 사용되는 방식은 앞으로 사용되지 않을 메모리 주소들의 집합을 현재 환경으로부터 접근할 수 없는 메모리 주소들의 집합으로 어림잡는 것이다. 앞의 예제를 다시 보자.

```
let f = fun (x) (x+1) in
  let a = f 0 in
    a + 1
```

식(a + 1)을 계산하기 직전 환경과 메모리는 다음과 같다:

$$\rho = \{\texttt{f} \mapsto l_1, \texttt{a} \mapsto l_3\}, \sigma = \{l_1 \mapsto (\texttt{x}, \texttt{x+1}, \emptyset), l_2 \mapsto 0, l_3 \mapsto 1\}$$

현재 환경에서 변수 f와 a가 정의되어 있고, 각각 메모리 주소 l_1과 l_3를 가리키고 있다. 따라서 현재 환경에서 접근 가능한 주소들은 l_1, l_3이고 접근 불가능한 주소는 l_2이다. 현재 메모리에서 접근 불가능한 주소 l_2를 제거하여 $\sigma' = \{l_1 \mapsto (\texttt{x}, \texttt{x+1}, \emptyset), l_3 \mapsto 1\}$로 메모리를 재활용한 후 식 a + 1을 계산한다.

위의 어림잡는 방식은 안전(sound)하다. 현재 환경으로부터 접근할 수 없는 주소들의 집합(위의 예의 경우 $\{l_2\}$)은 앞으로 쓰이지 않을 주소들의 집합(위의 예의 경우 $\{l_1, l_2\}$)의 부분집합이기 때문이다. 앞으로 사용될 주소들은 절대로 재활용되지 않는다는 점에서 안전하다. 하지만 위 방식은 완전(complete)하지는 않다. 현재 환경에서 접근가능하지만 미래에 안 쓰이는 주소가 있을 수 있다. 위의 예에서 주소 l_1이 그렇다.

현재 환경으로부터 접근 가능한 메모리 주소를 찾을때, 메모리 내에서 만들어지는 링크 구조들을 모두 고려해야 한다. 예를 들어, 다음과 같은 환경(ρ)과 메모리(σ)에서 위의 방식으로 메모리 재활용을 해 보자.

$$\rho = \left\{ \begin{array}{l} \mathtt{x} \mapsto l_1 \\ \mathtt{y} \mapsto l_2 \end{array} \right\} \qquad \sigma = \left\{ \begin{array}{l} l_1 \mapsto 0 \\ l_2 \mapsto \{\mathtt{a} \mapsto l_3, \mathtt{b} \mapsto l_1\} \\ l_3 \mapsto l_4 \\ l_4 \mapsto (\mathtt{x}, E, \{\mathtt{z} \mapsto l_5\}) \\ l_5 \mapsto 0 \\ l_6 \mapsto l_7 \\ l_7 \mapsto l_6 \end{array} \right\}$$

먼저 환경에서 직접 접근이 가능한 주소 $\{l_1, l_2\}$를 모은다. 메모리에서 l_2에 담긴 값은 레코드 $\{\mathtt{a} \mapsto l_3, \mathtt{b} \mapsto l_1\}$이므로 현재 환경과 주소 l_2를 통해서 l_3와 l_1에 추가적으로 접근할 수 있다. 이들 주소들까지 모으면 접근 가능한 주소들의 집합은 $\{l_1, l_2, l_3\}$이 된다.

이전 집합에 비해서 l_3가 새로 추가되었으므로 메모리에서 l_3를 통해 추가적으로 접근 가능한 주소가 있는지를 살펴본다. l_3에는 주소값 l_4가 담겨 있으므로 접근가능한 메모리 집합을 $\{l_1, l_2, l_3, l_4\}$로 확장한다. 다시 새롭게 추가된 주소 l_4에 담긴 값을 보자. 함수값 $(\mathtt{x}, E, \{\mathtt{z} \mapsto l_5\})$이 담겨 있는데, 이 함수가 호출될 경우 저장된 환경 $\{\mathtt{z} \mapsto l_5\}$을 통해 주소 l_5에 접근 가능하다. 따라서 l_5도 집합에 추가한다: $\{l_1, l_2, l_3, l_4, l_5\}$. 다시 l_5에 대해 위 과정을 반복한다. 이 경우 l_5에는 정수값이 담겨 있으므로 추가적으로 접근가능한 주소는 더 이상 없고, 최종적으로 $\{l_1, l_2, l_3, l_4, l_5\}$가 현재 환경에서 접근가능한(reachable) 주소들의 전체 집합임을 알 수 있다. 메모리에서 이들만 남기고 나머지 주소들을 재활용하면 다음과 같다.

$$\sigma' = \left\{ \begin{array}{l} l_1 \mapsto 0 \\ l_2 \mapsto \{\mathtt{a} \mapsto l_3, \mathtt{b} \mapsto l_4\} \\ l_3 \mapsto l_4 \\ l_4 \mapsto (\mathtt{x}, E, \{\mathtt{z} \mapsto l_5\}) \\ l_5 \mapsto 0 \end{array} \right\}$$

현재 환경을 통해 접근가능한 메모리 주소들을 정확하게 정의해보자. $\mathsf{Reach}(\rho, \sigma)$를 메모리 σ의 주소들 중에서 환경 ρ을 통해서 접근 가능한 주소들의 집합이라고 하자. 그림 7.6의 규칙을 만족하는 가장 작은 집합으로 정의할 수 있다.

첫 번째 규칙은 현재 환경에서 직접 접근가능한 주소들이 모두 집합 $\mathsf{Reach}(\rho, \sigma)$에 포함되어야 함을 뜻한다. 두 번째 규칙은 l이

$$\frac{}{\rho(x) \in \mathsf{Reach}(\rho, \sigma)}\ x \in \mathrm{Dom}(\rho)$$

$$\frac{l \in \mathsf{Reach}(\rho, \sigma) \qquad \sigma(l) = l'}{l' \in \mathsf{Reach}(\rho, \sigma)}$$

$$\frac{l \in \mathsf{Reach}(\rho, \sigma) \qquad \sigma(l) = \{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\}}{\{l_1, \ldots, l_n\} \subseteq \mathsf{Reach}(\rho, \sigma)}$$

$$\frac{l \in \mathsf{Reach}(\rho, \sigma) \qquad \sigma(l) = (x, E, \rho')}{\mathsf{Reach}(\rho', \sigma) \subseteq \mathsf{Reach}(\rho, \sigma)}$$

그림 7.6: 환경에서 접근 가능한 메모리 주소들

접근가능한 주소이고, 메모리에서 l에 담긴 값이 다른 주소 l'일때, l'도 접근가능한 주소 집합에 포함되어야 함을 뜻한다. 비슷하게 세 번째 규칙은 l이 주소들의 모음인 레코드일때 각 필드가 의미하는 주소들이 모두 접근가능한 주소 집합에 속해야 함을 뜻한다. 마지막 규칙은 l에 담긴 값이 함수값일 때, 그 함수의 몸통이 계산될때의 환경으로부터 접근 가능한 메모리 주소들($\mathsf{Reach}(\rho', \sigma)$)도 모두 모아야 함을 의미한다.

GC를 자동 메모리 재활용(garbage collection)을 수행하는 함수라고 하자. 다음과 같이 정의할 수 있다.

$$\mathsf{GC}(\rho, \sigma) = \sigma|_{\mathsf{Reach}(\rho,\sigma)}$$

여기서 $\sigma|_X$는 메모리 σ에서 집합 X에 있는 주소들만 남기고 나머지 주소들은 삭제한 메모리를 뜻한다. GC를 사용하는 방법은 계산

후 할일(continuation)이 없는 임의의 식 E에 대해서 그 식을 계산하기 전에 호출하면 된다.

$$\rho, \mathsf{GC}(\rho, \sigma) \vdash E \Rightarrow v, \sigma'$$

예를 들어, 앞의 예제 프로그램

```
let f = fun (x) (x+1) in
  let a = f 0 in
    a + 1
```

에서 식 `a + 1`은 계산 후에 할일이 남아 있지 않으므로 실행 전에 GC를 호출해도 된다. 하지만 함수 호출식 `f 0` 또는 함수 몸통식 `x+1`을 계산할 때에는 할일이 남아 있으므로 GC를 호출하지 않는다.

7.4 구현

지금까지의 언어를 OCaml로 아래와 같이 정의할 수 있다.

```
type program = exp
and exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
  | DIV of exp * exp
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
```

```
  | LETREC of var * var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
  | CALLREF of exp * var
  | ASSIGN of var * exp
  | SEQ of exp * exp
  | EMPTYREC                          (* {} *)
  | REC of var * exp * var * exp      (* { x:=E1, y:=E2 *)
  | FIELD of exp * var                (* E.x *)
  | FIELDASSIGN of exp * var * exp    (* E1.x := E2 *)
  | NEW of exp                        (* new E *)
  | ADDROF of var                     (* &x *)
  | DEREF of exp                      (* *E *)
  | STORE of exp * exp                (* *E1 := E2 *)
and var = string
```

1. 위 언어에 대한 값(value), 환경(env), 메모리(mem)를 구현해보자.

2. 아래 타입의 실행 함수를 구현해보자.

```
eval: program -> env -> mem -> mem
```

3. 아래 타입의 자동 메모리 재활용기 gc를 구현해보자.

```
gc: env -> mem -> mem
```

4. 메모리 재활용기 gc를 위에서 정의한 실행기 eval에 장착해보자.

8

정적 타입 시스템

지금까지 프로그래밍 언어의 실행 의미를 정의하고 실행기를 구현해 보았다. 이 장에서는 프로그램을 실행시키지 않고 프로그램의 실행 의미를 예측하는 방안을 고안해보자. 일반적으로 정적 프로그램 분석(static program analysis) 또는 정적 분석(static analysis)이라고 불리우는 기술이다. 정적 분석의 여러 갈래 중에서 최근 프로그래밍 언어들이 많이 활용하고 있는 정적 타입 시스템(static type system)의 원리를 이해하고 우리 언어에 장착해보자.

정적 타입 시스템의 목표는 타입 오류를 가지는 프로그램을 실행 전에 미리 모두 걸러내는 것이다. 예를 들어, 아래 프로그램들은 모두 타입 오류를 가지고 있고 타입 시스템에 의해 걸러진다.

- `let x = iszero 0 in (x+3)`

  덧셈 $E_1 + E_2$의 의미구조 규칙에 의하면 E_1과 E_2의 값은 모두 정수 타입이어야 한다. 이 예제에서 `x`값은 부울 타입의 값 *true*이므로 `x+3`의 의미가 정의되지 않는다.

- `if 3 then 88 else 99`

  의미구조에 의하면 조건식 `if` E_1 `then` E_2 `else` E_3에서 E_1의 값은 *true* 또는 *false*이어야 한다. 이 예제에서는 정수값

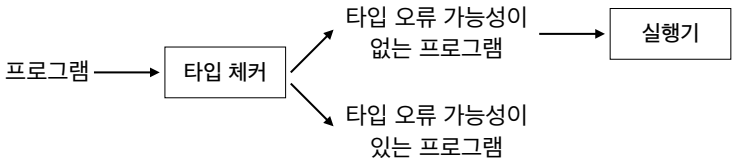


그림 8.1: 타입 시스템이 장착된 프로그래밍 언어 시스템

이 계산되므로 실행 중에 의미가 정의되지 않고 타입 오류가 발생한다.

- `(fun x (3 x)) 1`

  함수 호출식 `(3 x)`에서 정수값을 함수로 취급하여 호출하고 있다. 역시 의미구조에 규칙에서 적용할 규칙을 찾을 수 없고 실행 중에 타입 오류가 발생한다.

실제 프로그래밍 언어에서 타입 오류는 복잡하고 다양한 상황에서 발생할 수 있다. 이러한 오류들을 모두 예측하며 프로그램을 작성하는 것은 어려운 일이다. 정적 타입 시스템은 타입 오류를 컴파일 단계에서 자동으로 모두 찾아 줌으로써 안전한 프로그램을 손쉽게 개발할 수 있게 해 준다.

그림 8.1은 정적 타입 시스템이 장착되어 있는 프로그래밍 언어에서 프로그램이 실행되는 과정을 나타낸다. 먼저 입력 프로그램을 타입 체커(type checker)가 분석하여 프로그램의 실행 중에 타입 오류가 발생할 가능성이 있는지를 따져본다. 만약 타입 오류의 가능성이 발견된 경우, 타입 체커는 그 프로그램을 거부하게 되

$$
\begin{array}{rcl}
E & \rightarrow & n \\
& | & x \\
& | & E_1 \texttt{ + } E_2 \\
& | & \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \\
& | & \texttt{iszero } E \\
& | & \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \\
& | & \texttt{fun } x\ E \\
& | & \texttt{letrec } f(x) = E \texttt{ in } E \\
& | & E_1\ E_2
\end{array}
$$

그림 8.2: 대상 언어의 문법구조

고 입력 프로그램은 실행되지 못한다. 따라서 타입 체커를 통과하고 실행기에 의해 실행되는 프로그램은 타입 오류 가능성이 없는 것들이다. 타입 체커를 통과한 프로그램은 절대로 실행 중에 타입 오류가 발생하지 않음이 보장된다.

8.1 대상 언어

타입 시스템을 설계할 대상으로 그림 8.2의 언어를 생각하자. 함수형 언어의 주요 특징만 가지고 있는 언어이다. 의미공간은 다음과 같다.

$$
\begin{array}{rcl}
\mathit{Val} & = & \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} + \mathit{RecProcedure} \\
\mathit{Procedure} & = & \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \\
\mathit{RecProcedure} & = & \mathit{Var} \times \mathit{Var} \times \mathit{Exp} \times \mathit{Env}
\end{array}
$$

실행 규칙은 그림 8.3과 같다.

$$\frac{}{\rho \vdash n \Rightarrow n} \qquad \frac{}{\rho \vdash x \Rightarrow \rho(x)}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2}$$

$$\frac{\rho \vdash E \Rightarrow 0}{\rho \vdash \texttt{iszero } E \Rightarrow \mathit{true}} \qquad \frac{\rho \vdash E \Rightarrow n}{\rho \vdash \texttt{iszero } E \Rightarrow \mathit{false}}\ n \neq 0$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{true} \qquad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{false} \qquad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \{x \mapsto v_1\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v}$$

$$\frac{}{\rho \vdash \texttt{fun } x\ E \Rightarrow (x, E, \rho)}$$

$$\frac{\{f \mapsto (f, x, E_1, \rho)\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{letrec } f(x) \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow (x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad \{x \mapsto v\}\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

$$\frac{\begin{array}{c}\rho \vdash E_1 \Rightarrow (f, x, E, \rho') \qquad \rho \vdash E_2 \Rightarrow v \\ \{x \mapsto v, f \mapsto (f, x, E, \rho')\}\rho' \vdash E \Rightarrow v'\end{array}}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

그림 8.3: 대상 언어의 의미구조

8.2 타입

먼저 '타입'이 무엇인지 정의하자. 프로그래밍 언어에서 타입은 특정 값들의 집합을 뜻한다. 예를 들어, 정수 타입(int)은 정수 전체

집합($\mathbb{Z}$)을 의미하고, 부울(bool) 타입은 집합 $\{true, false\}$를 뜻한다. 이처럼 타입이란 프로그램에서 사용되는 값들을 그 종류별로 요약(abstraction)한 것이라 할 수 있다. 정수값 $1, 2, 3, \ldots$ 등을 일일이 구별하지 않고 요약하여 int라고 부르는 식이다.

일반적으로 타입은 값을 요약한 것이고 다양한 요약 수준에서 타입을 정의할 수 있지만[1] 대부분의 프로그래밍 언어에서는 다음과 같은 수준에서 타입을 정의하는 것이 보통이다.[2]

$$\frac{}{\text{int}} \qquad \frac{}{\text{bool}} \qquad \frac{T_1 \quad T_2}{T_1 \to T_2}$$

전체 타입들의 집합을 귀납적으로 정의하였다. int와 bool은 각각 정수와 부울 타입을 뜻한다. 함수는 인자와 결과값의 타입으로 정의하였다. T_1과 T_2가 임의의 타입일 때 $T_1 \to T_2$는 T_1에서 T_2로 가는 함수 타입을 뜻한다. 모든 정수를 'int'로 요약한 것처럼 모든 함수들을 'function'으로 요약할 수도 있지만 이럴 경우 타입 시스템의 표현력이 너무 떨어져서 결국 유용하지 않은 타입 시스템이 만들어지게 된다.

함수 타입의 몇 가지 예를 들어 보자.

---

1) 예를 들어, 모든 정수들을 단순히 int라고 부르는 대신 자연수의 타입은 $\text{int}_{\geq 0}$, 음의 정수의 타입은 $\text{int}_{<0}$라고 더 세밀하게 구분할 수도 있다.

2) 타입을 매우 일반적으로 사용하는 프로그래밍 언어가 궁금하다면 Coq (https://coq.inria.fr)을 살펴보자. Coq은 OCaml과 비슷한 함수형 언어이지만 지원하는 타입의 표현력이 매우 풍부하여 프로그램의 임의의 성질을 타입으로 표현할 수 있다. Coq의 타입 체커는 프로그램이 해당 성질을 가지는지를 자동으로 확인해주는데 이러한 관점에서 Coq을 증명 보조기(proof assistant)라고 부른다.

- int → int: 정수를 입력으로 받아서 정수를 반환하는 함수들의 집합을 의미한다. 예를 들어

```
fun n (n+1), fun n (let x = 1 in x + n)
```

  등이 이 타입에 속하는 함수들이다.

- bool → int: 참/거짓값을 입력으로 받아서 정수를 반환하는 함수들을 뜻한다. 예를 들어, `fun b (if b then 0 else 1)` 과 같은 함수가 이 타입에 속한다.

- int → (int → bool): 정수를 입력으로 받아서 int → bool 타입의 함수를 반환하는 함수들을 뜻한다. 예를 들어,

```
fun n (fun m (iszero (m + n)))
```

  와 같은 함수들이 이 타입에 속한다.

- (int → int) → (bool → bool): int → int 타입의 함수를 입력으로 받아서 bool → bool 타입의 함수를 반환하는 함수를 뜻한다. 예를 들어, 다음 함수가 이 타입에 속한다.

```
fun f (fun b (if b then b else (iszero (f 1))))
```

  더 간단한 예로 `fun f (fun b (b))`도 이 타입에 속한다.

- (int → int) → (bool → (bool → int)): int → int 타입의

함수를 인자로 받아서 (bool → (bool → int)) 타입의 함수를 반환하는 함수 타입이다. 예를 들어,

```
fun f (fun b1 (fun b2 (f 1)))
```

와 같은 함수가 이 타입에 속한다.

8.3 타입 환경

의미구조 정의와 마찬가지로 식 E의 타입은 E에 속한 자유 변수들의 타입을 알아야 결정할 수 있다. 예를 들어, 식 `x+1`의 타입은 변수 `x`의 타입이 int일 경우에는 int이지만 다른 경우에는 정의되지 않는다. 현재 선언된 변수들이 어떤 타입을 가지고 있는지를 나타내는 자료구조를 타입 환경(type environment)이라고 부른다. 실행 환경과 비슷하게 타입 환경은 변수에서 타입으로 가는 함수로 정의할 수 있다.

$$\Gamma \in \mathit{TyEnv} = \mathit{Var} \rightarrow \mathit{Type}$$

여기서 TyEnv는 타입 환경들의 전체 집합을 뜻하고, Γ는 하나의 타입 환경을 나타낸다. 실행 환경에 대한 표기법과 유사하게 다음과 같이 타입 환경에 대한 표기법을 정의하자.

- $\emptyset$: 빈 타입 환경을 뜻한다.

- $\{x \mapsto t\}\Gamma$: 변수 x가 타입 t를 가지도록 타입 환경 Γ를 확장한 새로운 타입 환경을 뜻한다.
- $\Gamma(x)$: 변수 x가 타입 환경 Γ에서 가지는 타입을 뜻한다.

참고로 타입이 값들의 집합을 의미하듯이 타입 환경은 실행 환경들의 집합을 뜻한다. 예를 들어, 타입 환경

$$\{x \mapsto \text{int}, y \mapsto \text{bool}\}$$

은 다음과 같이 무한히 많은 실행 환경을 유한하게 요약한 것이다.

$$\begin{gathered}
\vdots \\
\{x \mapsto 0, y \mapsto \mathit{true}\}, \\
\{x \mapsto 1, y \mapsto \mathit{true}\}, \\
\{x \mapsto 2, y \mapsto \mathit{true}\}, \\
\vdots \\
\{x \mapsto 0, y \mapsto \mathit{false}\}, \\
\{x \mapsto 1, y \mapsto \mathit{false}\}, \\
\{x \mapsto 2, y \mapsto \mathit{false}\}, \\
\vdots
\end{gathered}$$

이러한 관점에서 정적 타입 시스템을 타입이라고 하는 요약된 값을 가지고 프로그램을 실행시키는 요약 실행기(abstract interpreter)라고 직관적으로 이해해도 된다. 예를 들어, 식 $1 + 2$에 대

한 실제 실행은 3을 계산하지만, 요약 실행은 정수 1과 2를 각각 타입 int로 요약한 후 타입의 세계에서 덧셈을 수행하여 그 결과로 int를 계산하는 것이다. 프로그램의 실제 실행 의미를 실행 전에 정확하게 계산하는 것은 불가능(undecidable)하지만 타입으로 요약한 후 계산하는 것은 가능(decidable)할 수 있다.

8.4 타입 추론 규칙

"타입환경 Γ에서 식 E가 타입 t를 가진다"를 다음과 같이 표현하기로 하자.

$$\Gamma \vdash E : t$$

위 사실들을 추론하는 규칙을 고안해보자. E의 각 경우마다 하나씩 추론 규칙을 귀납적으로 정의하면 된다.

- $E = n$인 경우: 정수 n의 타입은 타입 환경과 상관없이 항상 int이다. 이를 추론 규칙으로 표현하면 다음과 같다.

$$\frac{}{\Gamma \vdash n : \text{int}}$$

  타입 환경 Γ가 주어졌을 때, 임의의 정수 n의 타입을 항상 int로 결론지을 수 있다는 뜻이다.

- x인 경우: 변수 x의 타입은 주어진 타입 환경에 의해 결정된

다. 추론 규칙으로 표현하면 다음과 같다.

$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

$\Gamma(x)$가 정의되는 경우, 즉 타입 환경 Γ가 변수 x에 대한 정보를 포함하는 경우에만 타입 추론이 가능하다.

- E_1 + E_2인 경우: 덧셈식 E_1 + E_2의 결과값은 int 타입이다. 하지만 항상 타입을 가지지는 못하고 식 E_1과 E_2가 모두 int 타입인 경우만 타입을 가진다. 다음의 추론 규칙으로 표현할 수 있다.

$$\frac{\Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 \texttt{ + } E_2 : \text{int}}$$

- `iszero` E인 경우: `iszero` E의 계산결과는 항상 참/거짓 값이므로 bool타입을 가진다. 하지만 타입을 가지는 경우는 E가 정수값을 계산하는 경우이다. 추론 규칙으로 표현하면 다음과 같다.

$$\frac{\Gamma \vdash E : \text{int}}{\Gamma \vdash \texttt{iszero } E : \text{bool}}$$

- `if` E_1 `then` E_2 `else` E_3인 경우: 먼저 E_1의 타입이 bool이라고 하자. 또한 E_2와 E_3의 타입이 동일하다고 가정하고 그 타입을 t라고 하자. 그렇다면 전체 조건식의 타입을 t라고 할 수 있다. 이를 추론 규칙으로 표현하면 다음과 같다.

$$\frac{\Gamma \vdash E_1 : \text{bool} \quad \Gamma \vdash E_2 : t \quad \Gamma \vdash E_3 : t}{\Gamma \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 : t}$$

이 규칙의 경우 E_2와 E_3의 타입이 같은 경우만 고려하였다. 따라서 다음과 같이 조건식의 각 경우에서 서로 다른 타입의 값을 생성하는 프로그램에 대해서는 타입을 추론하지 못하게 된다.

```
if true then 1 else false
```

OCaml의 타입 시스템도 동일한 제약을 가지는데, 식 E_2와 E_3의 타입이 다른 경우까지 고려하면 정적 타입 체킹의 비용이 너무 비싸지기 때문이다.

- `let` x `=` E_1 `in` E_2인 경우: 타입 환경 Γ에서 E_1의 타입을 t_1으로 추론할 수 있다고 가정해보자. 그리고 타입 환경 $\{x \mapsto t_1\}\Gamma$에서 E_2의 타입을 t_2로 추론할 수 있다고 하자. 그렇다면 전체식 `let` x `=` E_1 `in` E_2의 타입을 t_2로 결론지을 수 있다. 이 과정을 추론 규칙으로 다음과 같이 표현할 수 있다.

$$\frac{\Gamma \vdash E_1 : t_1 \quad \{x \mapsto t_1\}\Gamma \vdash E_2 : t_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ E_1\ \texttt{in}\ E_2 : t_2}$$

- E_1 E_2인 경우: 이 경우 의미가 정의되려면 일단 식 E_1이 함수값을 만들어내는 식이어야 한다. 그 함수값의 타입이 $t_1 \to t_2$ 라고 하자. 그리고 E_2의 타입이 인자 타입 t_1과 동일하다고 하자. 그렇다면 함수 호출식 E_1 E_2를 문제없이 실행할 수 있고 그 결과값의 타입은 함수의 결과 타입인 t_2가 된다. 다

음의 추론 규칙으로 표현할 수 있다.

$$\frac{\Gamma \vdash E_1 : t_1 \to t_2 \quad \Gamma \vdash E_2 : t_1}{\Gamma \vdash E_1\ E_2 : t_2}$$

- `fun` x E인 경우: 인자 x의 타입이 t_1이라고 가정한 후 E의 타입을 t_2로 추론할 수 있다고 하자. 그렇다면 주어진 함수식의 타입을 $t_1 \to t_2$라고 결론 내릴 수 있다. 추론 규칙으로 표현하면 다음과 같다.

$$\frac{\{x \mapsto t_1\}\Gamma \vdash E : t_2}{\Gamma \vdash \texttt{fun}\ x\ E : t_1 \to t_2}$$

  추론 규칙을 고안하는 시점에서 x의 타입 t_1을 어떻게 찾을지 고민할 필요는 없다. 위의 추론 규칙은 단지 x가 타입 t_1을 가진다고 가정한 경우에 논리적으로 어떤 성질을 추론할 수 있는지를 나타내고 있을 뿐이다.

- `letrec` $f(x) = E_1$ `in` E_2인 경우: 재귀 함수의 인자와 결과 타입을 각각 t_2, t_1이라고 하고 인자 x의 타입을 t_2라고 가정한 상황에서 함수의 몸통식 E_1의 타입을 t_1으로 추론할 수 있다고 해보자. 그리고 그러한 t_1, t_2에 대해서 (f가 $t_2 \to t_1$ 타입임을 가정한 후) E_2의 타입을 t로 추론할 수 있다고 하면, 전체식의 타입을 t로 결론지을 수 있다. 추론 규칙으로

표현하면 다음과 같다.

$$\frac{\{x \mapsto t_2, f \mapsto (t_2 \to t_1)\}\Gamma \vdash E_1 : t_1 \quad \{f \mapsto (t_2 \to t_1)\}\Gamma \vdash E_2 : t}{\Gamma \vdash \texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2 : t}$$

이 규칙에서도 t_1, t_2가 구체적으로 무엇인지는 중요하지 않다. x의 타입을 t_2, f의 타입을 $t_1 \to t_1$이라고 가정할 때 전체식의 타입이 무엇이 되어야 하는지에 대한 논리적 추론 관계를 표현한 것이다.

지금까지 정의한 타입 추론 규칙들을 정리하면 그림 8.4와 같다.

그림 8.1의 타입 체커는 입력으로 주어진 프로그램 E가 추론 규칙에 의해서 어떤 타입 t를 가질 수 있는지를 분석한다. 위 추론 규칙으로 어떤 타입 t와 빈 타입 환경($\emptyset$)에 대해서 아래 사실을 증명할 수 있는지를 확인해 보는 것이다.

$$\emptyset \vdash E : t$$

이 사실을 추론하는 데 성공하면 타입 체커는 프로그램 E의 타입이 t이고 따라서 타입 오류가 없다고 결론 내린다. 위 사실을 추론하는 데 실패하는 경우 타입 체커는 E가 타입을 가질 수 없고 따라서 타입 오류가 있을 수 있다고 결론 짓고 실행을 거부하게 된다.

$$\frac{}{\Gamma \vdash n : \text{int}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 \texttt{ + } E_2 : \text{int}}$$

$$\frac{\Gamma \vdash E : \text{int}}{\Gamma \vdash \texttt{iszero } E : \text{bool}}$$

$$\frac{\Gamma \vdash E_1 : \text{bool} \quad \Gamma \vdash E_2 : t \quad \Gamma \vdash E_3 : t}{\Gamma \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 : t}$$

$$\frac{\Gamma \vdash E_1 : t_1 \quad \{x \mapsto t_1\}\Gamma \vdash E_2 : t_2}{\Gamma \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 : t_2}$$

$$\frac{\Gamma \vdash E_1 : t_1 \to t_2 \quad \Gamma \vdash E_2 : t_1}{\Gamma \vdash E_1 \ E_2 : t_2}$$

$$\frac{\{x \mapsto t_1\}\Gamma \vdash E : t_2}{\Gamma \vdash \texttt{fun } x \ E : t_1 \to t_2}$$

$$\frac{\begin{array}{c}\{x \mapsto t_2, f \mapsto (t_2 \to t_1)\}\Gamma \vdash E_1 : t_1 \\ \{f \mapsto (t_2 \to t_1)\}\Gamma \vdash E_2 : t\end{array}}{\Gamma \vdash \texttt{letrec } f(x) = E_1 \texttt{ in } E_2 : t}$$

그림 8.4: 타입 추론 규칙

이러한 관점에서 타입 체커는 프로그램 E가 주어졌을 때

$$\frac{\vdots}{\emptyset \vdash E : t}$$

꼴의 증명을 시도하는 자동 증명기(automatic theorem prover)라 할 수 있다.

몇 가지 예를 들어 보자. 식 `iszero (1 + 2)`는 타입 bool의 값

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\texttt{x} \mapsto \text{int}, \texttt{y} \mapsto \text{bool}\} \vdash \texttt{y} : \text{bool} \quad \{\texttt{x} \mapsto \text{int}, \texttt{y} \mapsto \text{bool}\} \vdash \texttt{x} : \text{int} \quad \{\texttt{x} \mapsto \text{int}, \texttt{y} \mapsto \text{bool}\} \vdash \texttt{1} : \text{int}}{\{\texttt{x} \mapsto \text{int}, \texttt{y} \mapsto \text{bool}\} \vdash \texttt{if y then x else 1} : \text{int}}}{\{\texttt{x} \mapsto \text{int}\} \vdash \texttt{fun y (if y then x else 1)} : \text{bool} \to \text{int}}}{\emptyset \vdash \texttt{fun x (fun y (if y then x else 1))} : \text{int} \to (\text{bool} \to \text{int})}$$

그림 8.5: 타입 추론 과정의 예

을 만들어내며, 실행 중에 타입 오류가 발생하지 않는데, 다음과 같이 타입 추론 규칙에 의해서 $\emptyset \vdash$ `iszero (1 + 2)` : bool을 증명할 수 있기 때문이다.

$$\dfrac{\dfrac{\emptyset \vdash \texttt{1} : \text{int} \quad \emptyset \vdash \texttt{2} : \text{int}}{\emptyset \vdash \texttt{1 + 2} : \text{int}}}{\emptyset \vdash \texttt{iszero (1 + 2)} : \text{bool}}$$

식 `(fun x (x)) 1`의 타입도 다음과 같이 추론할 수 있다.

$$\dfrac{\dfrac{\{\texttt{x} \mapsto \text{int}\} \vdash \texttt{x} : \text{int}}{\emptyset \vdash \texttt{fun x (x)} : \text{int} \to \text{int}} \quad \emptyset \vdash \texttt{1} : \text{int}}{\emptyset \vdash \texttt{(fun x (x)) 1} : \text{int}}$$

타입 체커는 이러한 증명 나무를 자동으로 생성해보고 타입 오류가 없음을 확인해낸다.

식 `fun x (fun y (if y then x else 1))`의 경우도 타입이 맞는 프로그램이며 그림 8.5와 같이 그 타입이 int → (bool → int)임을 증명할 수 있다.

반면에 타입 오류가 있는 프로그램은 위의 추론 규칙에 의해서 타입을 가지지 않는다. 예를 들어, 식 `(fun x (3 x)) 1`의 타입은 위의 규칙으로 추론되지 않는다.

지금까지 디자인한 타입 추론 규칙의 특징을 살펴보자.

타입 추론은 유일하지 않다 먼저 프로그램 E가 주어졌을 때 위 규칙에 의해 결정되는 타입은 유일하지 않을 수 있다. 예를 들어,

- `fun x (x)`에 대해서 무한히 많은 타입을 부여할 수 있다. 임의의 타입 t에 대해서 $t \to t$ 꼴의 타입이면 모두 함수 `fun x x`의 타입이라 할 수 있다. 몇 가지 예를 보이면 다음과 같다.

$$\frac{\{\texttt{x} \mapsto \text{int}\} \vdash \texttt{x} : \text{int}}{\emptyset \vdash \texttt{fun x x} : \text{int} \to \text{int}}$$

$$\frac{\{\texttt{x} \mapsto \text{bool}\} \vdash \texttt{x} : \text{bool}}{\emptyset \vdash \texttt{fun x x} : \text{bool} \to \text{bool}}$$

$$\frac{\{\texttt{x} \mapsto (\text{int} \to \text{int})\} \vdash \texttt{x} : \text{int} \to \text{int}}{\emptyset \vdash \texttt{fun x x} : (\text{int} \to \text{int}) \to (\text{int} \to \text{int})}$$

$$\vdots$$

- `fun f (f 3)`: 임의의 타입 t에 대해서 $(\text{int} \to t) \to t$ 타입을

가진다. $t = \text{bool}$인 경우 다음과 같이 추론된다.

$$\dfrac{\dfrac{\{\texttt{f} \mapsto \text{int} \to \text{bool}\} \vdash \texttt{f} : \text{int} \to \text{bool} \quad \{\texttt{f} \mapsto \text{int} \to \text{bool}\} \vdash \texttt{3} : \text{int}}{\{\texttt{f} \mapsto \text{int} \to \text{bool}\} \vdash \texttt{f 3} : \text{bool}}}{\emptyset \vdash \texttt{fun f (f 3)} : (\text{int} \to \text{bool}) \to \text{bool}}$$

- `fun f (fun x (f (f x)))`: 임의의 타입 t에 대해서 $((t \to t) \to (t \to t))$ 타입을 가진다.

타입 추론은 안전하다 우리가 디자인한 타입 추론 규칙은 안전(sound)하다. 타입 오류가 있는 프로그램은 절대로 타입이 추론되지 않는다. 다시 말해 추론 규칙에 의해서 타입 추론에 성공하면 프로그램 실행 중에 타입 오류가 발생하지 않는다. 자유 변수가 없는 식 E와 빈 타입 환경에 대해서 위의 추론 규칙으로 $\emptyset \vdash E : t$를 추론할 수 있으면, E는 실행 중에 타입 오류를 발생시키지 않는 것이다. 또한 식 E의 계산이 종료되고 값 v를 계산한다면 그 값의 타입은 t임이 보장된다. 예를 들어,

- `(fun (x) x) 1`는 위의 규칙으로 타입(int)을 추론할 수 있으므로 타입 오류없이 잘 실행되며 그 결과는 정수값이다.
- `(fun (x) (x 3)) 4`는 타입 오류가 발생하는 프로그램이다. 이와 같이 타입 오류를 가지는 프로그램은 위의 규칙으로 타입을 추론할 수 없다.

이러한 타입 시스템의 안전성(soundness)은 엄밀하게 증명할 수 있다. 타입 시스템의 안전성은 프로그래밍 언어의 실행 규칙(그림 8.3)과 타입 추론 규칙(그림 8.4) 사이에 성립하는 성질이다. 두 규칙이 모두 수학적으로 정의되어 있으므로 둘 사이의 관계와 성질 또한 엄밀하게 정의하고 증명할 수 있을 것이다.[3)]

타입 추론은 완전하지 않다 우리가 디자인한 타입 시스템은 안전하지만 완전하지는 않다(incomplete). 타입 오류 없이 잘 도는 프로그램이지만 타입을 추론하지 못할 수 있다는 뜻이다. 예를 들어, 아래 두 프로그램은 의미구조에 따라 실행하면 아무 문제 없이 실행되지만 우리 타입 시스템에 따르면 타입을 가지지 않는다.

- `if iszero 1 then 2 else (iszero 3)`
- `(fun f (f f)) (fun x x)`

표현력이 완전한(Turing-complete) 일반적인 프로그래밍 언어를 대상으로 안전하고 완전한(sound and complete) 정적 타입 시스템을 설계하는 것은 불가능하다. 따라서 정적 타입 시스템은 항상 안전성 또는 완전성 둘 중 하나(또는 둘 다)를 포기해야 하는데 우리 타입 시스템의 경우 완전성을 포기하고 안전성은 포기하지 않는 방식을 택했다.

3) 사실 우리가 정의한 실행 규칙 스타일은 타입 시스템 증명에 용이하지 않다. 타입 시스템의 안전성을 어떻게 증명하는지 궁금하다면 다음 책을 참고해보자. Benjamin C. Pierce. Types and Programming Languages. MIT Press.

8.5 타입 체커 구현 방안

타입 체커는 프로그램 E를 입력으로 받아서 타입 오류가 없는 프로그램인지를 검증하는 알고리즘이다. 동작하는 원리는 E를 빈 타입 환경에서 타입 추론 규칙에 의해서 추론을 시도하여 추론이 가능하면 타입 오류가 없다고 판단하고, 타입을 추론하는 것이 불가능하면 타입 오류가 있다고 판단하여 실행을 거부하는 것이다. 앞서 설명한대로 타입 체커는 타입 추론 규칙을 이용하여 어떤 타입 t에 대해서 $\emptyset \vdash E : t$에 대한 증명을 찾는 자동 증명기인 것이다. 증명 나무가 존재하는 타입 t를 찾는 데 성공하면 입력 프로그램을 받아들이고, 아니면 거부하게 된다.

- 타입체커가 프로그램 E를 받아들이는 경우는 어떤 t에 대해서 $\emptyset \vdash E : t$를 증명하는 데 성공한 경우이다.
- 증명이 불가능하다면, E를 받아들이지 않는다.

그러한 타입 체커를 어떻게 구현할 수 있을까? 실행기를 구현할때와 같이 타입 추론 규칙을 재귀 함수 형태로 구현을 시도해보자. 타입 추론 규칙을 그림 8.6과 같은 함수로 구현해 볼 수 있을 것이다.

대부분의 케이스에 대해서 재귀 함수로 추론 규칙을 구현할 수 있지만 한 가지 문제가 있다. 아래 규칙을 재귀함수로 구현하는 것은 불가능하다.

$$\frac{\{x \mapsto t_1\}\Gamma \vdash E : t_2}{\Gamma \vdash \texttt{fun}\ x\ E : t_1 \to t_2}$$

```
let rec typecheck Γ E =
  match E with
  | n → int
  | x → Γ(x)
  | E1 + E2 →
    let t1 = typecheck Γ E1
    let t2 = typecheck Γ E2
      if t1 = int and t2 = int then int
      else raise TypeError
  | ···
```

그림 8.6: 재귀 함수로 시도한 타입 체커 구현

이 규칙은 함수의 인자 x의 적절한 타입 t_1을 찾을 수 있고, 이 때 E의 타입이 t_2이면 함수의 타입을 $t_1 \to t_2$라고 결론지을 수 있음을 뜻한다. 논리적으로 문제 없는 추론 규칙이지만 이를 구현할 때에는 적절한 타입 t_1을 찾을 수 있어야 한다. 하지만 위 규칙을 재귀 함수로 구현하는 과정에서 타입 t_1이 무엇이어야 하는지 알아낼 방도가 마땅치 않다. t_1은 함수식 `fun` x E의 부분식의 타입을 재귀적으로 추론해서 얻어지는 것이 아니라 위의 추론 규칙을 적용할 때 미리 추측해야 하는 타입이기 때문이다. 인자 x의 타입 t_1을 미리 추측하기 어렵기 때문에 위 추론 규칙을 재귀 함수 형태의 알고리즘으로 바로 변환할 수 없다. 비슷하게 아래 추론 규칙에서도 인자와 함수의 타입(t_1, t_2)을 미리 추측하기 어렵다.

$$\frac{\{x \mapsto t_2, f \mapsto (t_2 \to t_1)\}\Gamma \vdash E_1 : t_1 \quad \{f \mapsto (t_2 \to t_1)\}\Gamma \vdash E_2 : t}{\Gamma \vdash \texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2 : t}$$

이 문제를 해결하는 방안에 따라 프로그래밍 언어들에 장착된

타입 시스템을 두 가지로 분류할 수 있다.

- 타입 추론을 프로그래머에게 맡기는 방식: C, C++, Java 등의 언어에서 사용하는 방식이다. 이들 언어에서는 예를 들어 함수 타입을 생략하는 것이 불가능하고 항상 적어주어야 한다.

- 알고리즘을 통해서 자동으로 타입을 추론하는 방식: OCaml, Haskell 등의 언어에서 사용하는 방식이다. 이들 언어에서는 함수 타입을 생략해도 타입 시스템이 자동으로 적절한 타입을 추론해준다.

타입 추론을 프로그래머에게 맡기는 방식은 간단하다. 프로그램에 타입을 적을 수 있도록 다음과 같이 언어의 문법을 먼저 변경한다.

$$
\begin{array}{rcl}
E & \rightarrow & \vdots \\
 & | & \texttt{fun}\ (x : t)\ E \\
 & | & \texttt{letrec}\ t_1\ f(x : t_2) = E\ \texttt{in}\ E
\end{array}
$$

다른 부분들은 모두 동일하고 함수를 정의하는 문법만 변경하였다. 일반 함수를 정의할 때는 인자 타입 t를 적도록 하였고, 재귀 함수의 경우에는 인자 타입 t_2와 결과 타입 t_1을 모두 적도록 하였다. 예를 들어, 이제 다음과 같이 함수를 정의할 때 타입을 적는 것이 선택이 아닌 필수가 된다.

```
fun (x: int) (x+1)
```

```
letrec int double (x: int) =
```

```
  if iszero x then 0 else (double (x-1)) + 2
 in double 2

fun (f: (bool -> int))
  fun (n : int)
    (f (iszero n))
```

타입 추론 규칙에서 바뀌는 부분은 아래 두 경우이다.

$$\frac{\{x \mapsto t_1\}\Gamma \vdash E : t_2}{\Gamma \vdash \texttt{fun}\ (x : t_1)\ E : t_1 \to t_2}$$

$$\frac{\{x \mapsto t_2, f \mapsto (t_2 \to t_1)\}\Gamma \vdash E_1 : t_1 \quad \{f \mapsto (t_2 \to t_1)\}\Gamma \vdash E_2 : t}{\Gamma \vdash \texttt{letrec}\ t_1\ f(x : t_2) = E_1\ \texttt{in}\ E_2 : t}$$

함수 인자의 타입이 프로그램에 적혀 있으므로 이를 추론 규칙에서 그대로 사용하도록 하였다. 재귀 함수를 정의할 때에도 적혀 있는 인자와 결과 타입을 그대로 이용하고 있다.

이제 타입 추론 규칙을 쉽게 재귀 함수로 구현할 수 있다. 위의 두 경우에 대해서만 보이면 그림 8.7과 같다.

8.6 자동 타입 추론 알고리즘

사람의 도움 없이 타입을 자동으로 추론하는 타입 체커 알고리즘을 고안해보자. OCaml 등의 프로그래밍 언어에서 사용하는 알고리즘으로 주어진 프로그램을 분석하여 함수의 인자 및 결과 타입을 추론해낸다. 주어진 프로그램으로부터 타입 방정식을 생성하고 푸는 두 단계로 동작한다.

```
let rec typecheck Γ E =
  match E with
  | fun (x : t1) E1 →
    let t2 = typecheck ({x ↦ t1}Γ) E1
    in t1 → t2
  | letrec t1 f(x : t2) = E1 in E2
    let t' = typecheck ({x ↦ t2, f ↦ (t2 → t1)}Γ) E1
    in if t' = t1 then typecheck ({f ↦ (t2 → t1)}Γ) E2
       else raise TypeError
  | ···
```

그림 8.7: 타입 추론을 프로그래머에게 맡긴 후 재귀 함수로 구현한 타입 체커

8.6.1 타입 방정식 생성

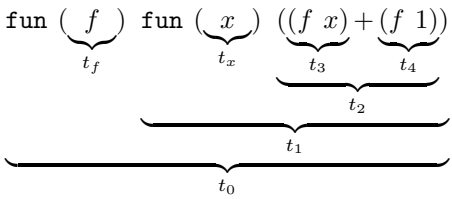
첫 단계에서는 프로그램의 각 부분식과 변수들에 대해서 타입 변수를 도입하고 이들이 만족해야 하는 조건을 방정식으로 표현한다.

예제 1 예를 들어 아래 프로그램을 생각하자.[4)]

$$\texttt{fun}\ (f)\ \texttt{fun}\ (x)\ ((f\ x) + (f\ 1))$$

4) 예제 1–4는 다음 책 7장의 예제들을 각색한 것들이다. Daniel P. Friedman and Mitchell Wand. Essentials of Programming Languages. MIT Press

프로그램의 각 부분식과 변수들에 대해서 다음과 같이 타입 변수를 도입하자.



예를 들어, t_f는 변수 f의 타입을 뜻하는 타입 변수이고, t_0는 전체 프로그램의 타입을 뜻하는 타입 변수, t_3는 부분식 $(f\ x)$의 타입을 뜻하는 타입 변수이다. 이와 같이 각 부분식과 변수들에 대하여 타입 변수들을 도입한 후에 이들이 만족해야 하는 조건들을 찾는다. 타입 변수들이 만족해야 하는 조건은 타입 추론 규칙으로부터 얻어진다. 예를 들어, 전체 프로그램은 함수 정의식이고 그 인자의 타입을 t_f, 결과 타입을 t_1이라고 하였으므로 타입 변수 t_0, t_1, t_f 사이에 다음과 같은 관계가 성립해야 한다.

$$t_0 = t_f \to t_1$$

t_0가 의미하는 타입이 t_f와 t_1으로 구성된 함수 타입과 같아야 함을 뜻한다. 비슷하게 t_1, t_x, t_2 사이에 다음과 같은 관계가 성립해야 한다.

$$t_1 = t_x \to t_2$$

또한 식 $((f\ x) + (f\ 1))$가 타입을 가지기 위해서는 그 타입이 int 이어야 하고, 부분식 $(f\ 3)$과 $(f\ x)$가 모두 int 타입이어야 한다. 이를 방정식으로 표현하면 다음과 같다.

$$t_2 = \text{int}, \quad t_3 = \text{int}, \quad t_4 = \text{int}$$

또한 함수 호출식 $(f\ 1)$을 보면 f가 함수 타입을 가져야 하고 그 인자 타입이 int, 결과 타입이 t_4여야 함을 알 수 있다.

$$t_f = \text{int} \to t_4$$

마지막으로 함수 호출식 $(f\ x)$에 의하면 t_f, t_x, t_3사이에 다음과 같은 관계가 성립해야 한다.

$$t_f = t_x \to t_3$$

지금까지 생성한 제약 조건들을 모두 모으면 그림 8.8의 왼쪽과 같은 연립 방정식이 된다. 미지수(타입 변수)와 방정식의 개수가 동일하므로 해가 존재한다면 모든 미지수의 값을 결정할 수 있는 경우이다. 방정식의 해는 그림 8.8의 오른쪽에 나타내었다. 해를 가지고 위 방정식의 미지수들을 해로 치환해보면 모든 방정식의 왼쪽, 오른쪽 항들이 일치하게 된다.

타입 방정식			타입 방정식의 해		
t_0	=	$t_f \to t_1$	t_0	=	$(\text{int} \to \text{int}) \to (\text{int} \to \text{int})$
t_1	=	$t_x \to t_2$	t_1	=	$\text{int} \to \text{int}$
t_2	=	int	t_2	=	int
t_3	=	int	t_3	=	int
t_4	=	int	t_4	=	int
t_f	=	$\text{int} \to t_4$	t_f	=	$\text{int} \to \text{int}$
t_f	=	$t_x \to t_3$	t_x	=	int

그림 8.8: 타입 방정식과 해

예제 2 아래 프로그램에 대해서

$$\underbrace{\texttt{fun}\ (\underbrace{f}_{t_f})\ (\underbrace{f\ 0}_{t_1})}_{t_0}$$

타입 방정식을 세우면 다음과 같다.

$$\begin{aligned} t_0 &= t_f \to t_1 \\ t_f &= \text{int} \to t_1 \end{aligned}$$

해는 다음과 같다.

$$\begin{aligned} t_0 &= (\text{int} \to t_1) \to t_1 \\ t_f &= \text{int} \to t_1 \end{aligned}$$

이 경우 미지수가 방정식보다 많으므로 모든 미지수의 해를 결정할 수 없다.

예제 3 아래 프로그램을 생각하자.

$$\underbrace{\texttt{if } \underbrace{x}_{t_x} \texttt{ then } \underbrace{(x+1)}_{t_1} \texttt{ else } 0}_{t_0}$$

타입 추론 규칙에 의하면 조건식 `if` E_1 `then` E_2 `else` E_3에서 E_1은 bool 타입이어야 하고 E_2와 E_3의 타입은 같아야 하며, 그 타입이 전체 조건식의 타입이 된다. 방정식으로 표현하면 다음과 같다.

$$\begin{aligned} t_x &= \text{bool} \\ t_1 &= t_0 \\ \text{int} &= t_0 \\ t_x &= \text{int} \\ t_1 &= \text{int} \end{aligned}$$

마지막 두 방정식은 $x+1$에서 생성되었다.

위 방정식은 해를 가지지 않는다. 첫 번째와 네 번째 방정식을 보면 서로 상충되는 조건을 나타내고 있기 때문이다.

예제 4 아래 프로그램에 대해서

$$\underbrace{\texttt{fun } \underbrace{(\ f\ )}_{t_f} \underbrace{(\texttt{iszero } \underbrace{(f\ f)}_{t_2})}_{t_1}}_{t_0}$$

방정식을 세우면 다음과 같다.

$$\begin{aligned} t_0 &= t_f \to t_1 \\ t_1 &= \text{bool} \\ t_2 &= \text{int} \\ t_f &= t_f \to t_2 \end{aligned}$$

마지막 방정식은 부분식 ($f\ f$)에서 생성되었다. f의 타입을 t_f라고 할 때, f의 인자로 f가 사용되고 있고 그 결과값이 `iszero`의 인자로 사용되기 때문이다.

위 방정식도 해를 가지지 않는다. 마지막 방정식을 보면 동일한 타입 변수 t_f가 $t_f = t_f \to \ldots$와 같은 형태로 쓰였다. 이러한 제약 조건을 만족하는 타입은 존재할 수 없다.

타입 방정식 생성 알고리즘

타입 방정식 생성 과정을 정확하게 기술해보자. 방정식에 타입 변수가 사용되므로 이전에 정의한 타입에 타입 변수를 추가하자.

$$\begin{aligned} T &\to \text{int} \\ &\mid \text{bool} \\ &\mid T \to T \\ &\mid \alpha\ (\in \mathit{TyVar}) \end{aligned}$$

여기서 α는 타입 변수, TyVar는 타입 변수들의 집합을 뜻한다.

타입 방정식은 다음과 같이 귀납적으로 정의할 수 있다.

$$\begin{array}{rcl} \mathit{TyEqn} & \rightarrow & \emptyset \\ & | & T_1 \doteq T_2 \ \wedge \ \mathit{TyEqn} \end{array}$$

여기서 $T_1 \doteq T_2$는 두 타입 T_1, T_2가 같음을 의미하는 방정식이다 (일반적인 $=$과 혼동을 피하기 위하여 두 타입의 같음을 나타내는 의미로 기호 $\doteq$를 사용하였다. 혼동되지 않을 때는 $\doteq$대신 $=$를 사용하기도 할 것이다). 타입 방정식은 이러한 개별 방정식들의 모음으로, 개별 방정식을 $\wedge$ (and)로 연결하였다.

프로그램으로부터 방정식을 도출하는 알고리즘의 타입은 다음과 같다.

$$\mathcal{V} : (\mathit{Var} \rightarrow T) \times E \times T \rightarrow \mathit{TyEqn}$$

알고리즘 $\mathcal{V}$는 세 가지 인자를 받는다. 첫 번째는 타입 환경 $\Gamma \in \mathit{Var} \rightarrow T$이고, 두 번째는 방정식을 도출할 프로그램 $e \in E$, 마지막은 주어진 프로그램이 가져야 하는 타입 $t \in T$이다. $\mathcal{V}(\Gamma, e, t)$는 식 e가 환경 Γ에서 타입 t를 가지기 위해서 성립해야 하는 조건들을 생성해낸다. 예를 들어,

- $\mathcal{V}(\{x \mapsto \text{int}\}, \texttt{x+1}, \alpha)$는 방정식 $\alpha \doteq \text{int}$를 생성한다. 환경 $\{x \mapsto \text{int}\}$에서 프로그램 `x+1`이 타입 α를 가지려면 조건 $\alpha \doteq \text{int}$가 성립해야 하기 때문이다.

$$
\begin{aligned}
\mathcal{V}(\Gamma, n, t) &= t \doteq \text{int} \\
\mathcal{V}(\Gamma, x, t) &= t \doteq \Gamma(x) \\
\mathcal{V}(\Gamma, e_1 + e_2, t) &= t \doteq \text{int} \ \wedge \ \mathcal{V}(\Gamma, e_1, \text{int}) \ \wedge \ \mathcal{V}(\Gamma, e_2, \text{int}) \\
\mathcal{V}(\Gamma, \texttt{iszero}\ e, t) &= t \doteq \text{bool} \ \wedge \ \mathcal{V}(\Gamma, e, \text{int}) \\
\mathcal{V}(\Gamma, \texttt{if}\ e_1\ e_2\ e_3, t) &= \mathcal{V}(\Gamma, e_1, \text{bool}) \ \wedge \ \mathcal{V}(\Gamma, e_2, t) \ \wedge \ \mathcal{V}(\Gamma, e_3, t) \\
\mathcal{V}(\Gamma, \texttt{let}\ x = e_1\ \texttt{in}\ e_2, t) &= \mathcal{V}(\Gamma, e_1, \alpha) \ \wedge \ \mathcal{V}([x \mapsto \alpha]\Gamma, e_2, t) \ (\text{new } \alpha) \\
\mathcal{V}(\Gamma, \texttt{fun}\ (x)\ e, t) &= t \doteq \alpha_1 \to \alpha_2 \ \wedge \ \mathcal{V}([x \mapsto \alpha_1]\Gamma, e, \alpha_2) \\
&\quad (\text{new } \alpha_1, \alpha_2) \\
\mathcal{V}(\Gamma, e_1\ e_2, t) &= \mathcal{V}(\Gamma, e_1, \alpha \to t) \ \wedge \ \mathcal{V}(\Gamma, e_2, \alpha) \ (\text{new } \alpha)
\end{aligned}
$$

그림 8.9: 타입 방정식 생성 알고리즘

- $\mathcal{V}(\emptyset, \texttt{fun}\ (x)\ (\texttt{if}\ x\ \texttt{then}\ 1\ \texttt{else}\ 2), \alpha \to \beta)$는 방정식

$$\alpha \doteq \text{bool} \ \wedge \ \beta \doteq \text{int}$$

  를 생성한다. 주어진 프로그램의 타입을 $\alpha \to \beta$라고 할 수 있으려면 $\alpha \doteq \text{bool}$과 $\beta \doteq \text{int}$가 만족되어야 하기 때문이다.

이와 같은 방정식을 생성하도록 알고리즘 $\mathcal{V}$를 재귀적으로 정의하면 그림 8.9과 같다.

- $\mathcal{V}(\Gamma, n, t)$: 타입 환경 Γ에서 정수 n이 타입 t를 가지려면 t가 int여야 한다($t \doteq \text{int}$).
- $\mathcal{V}(\Gamma, x, t)$: 타입 환경 Γ에서 변수 x가 타입 t를 가지려면 t가 타입 환경 Γ에서 x가 가지는 타입인 $\Gamma(x)$와 같아야 한다

$(t \doteq \Gamma(x))$.

- $\mathcal{V}(\Gamma, e_1 + e_2, t)$: 타입 환경 Γ에서 식 $e_1 + e_2$가 타입 t를 가진다면 t는 int이어야 한다($t \doteq \text{int}$). 그다음에 식 e_1과 e_2에 대해서 재귀적으로 타입 방정식을 생성한다($\mathcal{V}(\Gamma, e_1, \text{int})$, $\mathcal{V}(\Gamma, e_2, \text{int})$). 이 때 e_1과 e_2 모두 정수 타입이어야 하므로 $\mathcal{V}$의 마지막 인자로 int를 주었다.

- $\mathcal{V}(\Gamma, \texttt{iszero}\ e, t)$: 타입 환경 Γ에서 식 `iszero` e가 타입 t를 가지려면 t는 bool이어야 한다. 그리고 e에 대해서 재귀적으로 방정식을 생성한다($\mathcal{V}(\Gamma, e, \text{int})$).

- $\mathcal{V}(\Gamma, \texttt{if}\ e_1\ e_2\ e_3, t)$: 조건식의 경우 e_1의 타입은 bool이어야 하고 e_2와 e_3의 타입은 같아야 한다. 이러한 제약 조건을 담아서 타입 방정식을 재귀적으로 생성한다.

- $\mathcal{V}(\Gamma, \texttt{let}\ x = e_1\ \texttt{in}\ e_2, t)$: 먼저 식 e_1에 대하여 성립해야 하는 조건을 재귀적으로 생성한다($\mathcal{V}(\Gamma, e_1, \alpha)$). 이 때, 식 e_1이 만족해야 하는 특별한 조건이 없기 때문에 새로운 타입 변수 α를 생성하여 마지막 인자로 주었다. 재귀 호출 $\mathcal{V}(\Gamma, e_1, \alpha)$는 식 e_1을 분석하여 α가 만족해야 하는 조건들을 생성할 것이다. 그다음, 현재 타입 환경을 확장하여 변수 x가 α 타입을 가진다고 가정한 후 식 e_2에 대해서 방정식을 생성하였다. 식 e_2의 경우 그 타입이 전체 `let`식의 타입과 같아야 하므로 세 번째 인자로 t를 주었다.

- $\mathcal{V}(\Gamma, \texttt{fun}\ (x)\ e, t)$: 먼저 함수식 $\texttt{fun}\ (x)\ e$는 함수 타입을 가져야 한다. 함수의 인자와 결과 타입이 만족해야 하는 조건이 아직 없으므로 새로운 타입 변수 α_1, α_2를 생성하여 $t \doteq \alpha_1 \to \alpha_2$라고 하였다. 그다음에 현재 환경에서 변수 x의 타입을 인자 타입 α_1으로 가정한 후 함수의 몸통식 e_2에 대하여 방정식을 재귀적으로 생성한다. 이 때, 몸통식의 타입은 함수의 결과 타입과 같아야 하므로 세 번째 인자로 α_2를 주었다.

- $\mathcal{V}(\Gamma, e_1\ e_2, t)$: 식 e_1은 함수 타입이어야 하므로 타입 변수 α를 도입하여 e_1의 타입을 $\alpha \to t$로 두고 재귀적으로 방정식을 생성한다. 그다음 인자로 주어진 식 e_2에 대해서도 재귀적으로 방정식을 생성하는 데 그 타입은 함수의 인자 타입 α와 동일해야 한다.

자유변수가 없는 프로그램에 대해서 $\mathcal{V}$를 호출할 때에는 빈 환경과 새로운 타입 변수를 첫 번째, 세 번째 인자로 주면 된다. 예를 들어, $(\texttt{fun}\ (x)\ (x))\ 1$에 대해 타입 방정식을 생성하는 과정은 다음과 같다(α는 전체식의 타입을 뜻하는 타입 변수이다).

$$
\begin{aligned}
&\mathcal{V}(\emptyset, (\texttt{fun}\ (x)\ (x))\ 1, \alpha) \\
&= \mathcal{V}(\emptyset, \texttt{fun}\ (x)\ (x), \alpha_1 \to \alpha) \wedge \mathcal{V}(\emptyset, 1, \alpha_1) && (\text{new } \alpha_1) \\
&= \alpha_1 \to \alpha \doteq \alpha_2 \to \alpha_3 \wedge \mathcal{V}([x \mapsto \alpha_2], x, \alpha_3) \wedge \alpha_1 \doteq \text{int} && (\text{new } \alpha_2, \alpha_3) \\
&= \alpha_1 \to \alpha \doteq \alpha_2 \to \alpha_3 \wedge \alpha_2 \doteq \alpha_3 \wedge \alpha_1 \doteq \text{int}
\end{aligned}
$$

8.6.2 타입 방정식 풀기

이제 앞서 세운 타입 방정식의 해를 구하는 과정을 살펴보자. 아래 프로그램을 예로 들어 설명한다.

$$\underbrace{\texttt{fun}\ \underbrace{(f)}_{t_f}\ \texttt{fun}\ \underbrace{(x)}_{t_x}\ \underbrace{\underbrace{(\underbrace{(f}_{t_3}\ \underbrace{x)}_{t_3}}_{t_2} + \underbrace{\underbrace{(f}_{t_4}\ \underbrace{1))}_{t_4}}_{t_2}}_{t_1}}_{t_0}$$

위 프로그램에 대한 타입 방정식과 해는 그림 8.8과 같다.

타입 방정식의 해를 치환(substitution)이라고도 부르는데 해의 왼쪽에 있는 타입 변수를 오른쪽의 타입으로 치환해주는 함수로 볼 수 있기 때문이다. 그러한 함수를 타입 방정식의 각 타입 변수에 적용하면 각 방정식의 왼쪽, 오른쪽 항들이 동일해진다. 예를 들어, 그림 8.8의 첫 번째 방정식 $t_0 = t_f \to t_1$에서 각 타입 변수를 방정식의 해로 치환하면 다음과 같다.

$$(\text{int} \to \text{int}) \to (\text{int} \to \text{int}) = (\text{int} \to \text{int}) \to (\text{int} \to \text{int})$$

타입 방정식의 해는 동일화 알고리즘(unification algorithm)으로 구한다. 동일화 알고리즘은 빈 치환($\emptyset$)에서 시작하여 각 방정식을 순차적으로 치환으로 옮기는 방식으로 동작한다. 예를 들어, 첫 번째 방정식 $t_0 = t_f \to t_1$을 빈 치환으로 옮긴 후 방정식과 치환의

상태는 다음과 같다.

방정식	치환
$t_1 = t_x \to t_2$	$t_0 = t_f \to t_1$
$t_2 = \text{int}$	
$t_3 = \text{int}$	
$t_4 = \text{int}$	
$t_f = \text{int} \to t_4$	
$t_f = t_x \to t_3$	

그다음 방정식 $t_1 = t_x \to t_2$를 치환으로 옮기는데, 현재 옮기는 방정식의 왼쪽에 해당하는 변수(t_1)가 치환에 존재하는 경우 그 변수들을 방정식의 오른쪽($t_x \to t_2$)으로 변경한다. 따라서 $t_1 = t_x \to t_2$를 치환으로 옮기면 다음과 같은 상태가 된다.

방정식	치환
$t_2 = \text{int}$	$t_0 = t_f \to (t_x \to t_2)$
$t_3 = \text{int}$	$t_1 = t_x \to t_2$
$t_4 = \text{int}$	
$t_f = \text{int} \to t_4$	
$t_f = t_x \to t_3$	

동일한 방식으로 나머지 방정식들도 모두 치환으로 옮긴다. 하지만 만약 현재 옮기려는 방정식에 포함된 타입 변수에 대한 해가 현재 치환에 존재한다면 치환을 방정식에 적용하여 해당 변수를 그 해로

다시 쓴 후에 옮긴다. 예를 들어, 아래 상태를 보자.

방정식	치환
$t_f = \text{int} \to t_4$	$t_0 = t_f \to (t_x \to \text{int})$
$t_f = t_x \to t_3$	$t_1 = t_x \to \text{int}$
	$t_2 = \text{int}$
	$t_3 = \text{int}$
	$t_4 = \text{int}$

방정식 $t_f = \text{int} \to t_4$를 치환으로 옮기기 전에 변수 t_4를 int로 바꿔준다. 현재 치환에 타입 변수 t_4에 대한 해가 존재하기 때문이다. 방정식을 치환으로 옮기면 다음과 같은 상태가 된다.

방정식	치환
$t_f = t_x \to t_3$	$t_0 = (\text{int} \to \text{int}) \to (t_x \to \text{int})$
	$t_1 = t_x \to \text{int}$
	$t_2 = \text{int}$
	$t_3 = \text{int}$
	$t_4 = \text{int}$
	$t_f = \text{int} \to \text{int}$

그다음 방정식 $t_f = t_x \to t_3$를 처리하기 위해 먼저 치환을 적용하면 방정식이 다음과 같이 변경된다.

$$\text{int} \to \text{int} = t_x \to \text{int}$$

이와 같이 방정식의 왼쪽과 오른쪽이 모두 변수가 아닌 경우에는 방정식을 쪼갠다. 양쪽이 모두 함수 타입이고 이 둘이 서로 같으려면 두 인자 타입과 결과 타입이 각각 같아야 한다. 다음과 같이 방정식을 쪼갤 수 있다.

방정식	치환
$\text{int} = t_x$	$t_0 = (\text{int} \to \text{int}) \to (t_x \to \text{int})$
$\text{int} = \text{int}$	$t_1 = t_x \to \text{int}$
	$t_2 = \text{int}$
	$t_3 = \text{int}$
	$t_4 = \text{int}$
	$t_f = \text{int} \to \text{int}$

방정식 $\text{int} = t_x$와 같이 왼쪽항이 변수가 아닌 경우 양쪽을 바꾼 후 치환으로 이동시킨다. 마지막 방정식 $\text{int} = \text{int}$은 항상 성립하는 경우이므로 무시한다. 결국 최종적으로 그림 8.8의 오른쪽에 해당하는 해를 얻게 된다. 해에는 전체 프로그램의 타입(t_0)뿐 아니라 모든 부분식과 변수들의 타입이 추론되어 있음을 볼 수 있다.

동일화 알고리즘이 해를 구하지 못하는 경우도 있다. 아래 프로그램에 대해서

$$\underbrace{\texttt{if } \underbrace{x}_{t_x} \texttt{ then } \underbrace{(x+1)}_{t_1} \texttt{ else } 0}_{t_0}$$

방정식을 풀어보면 다음과 같이 모순적 상황에 봉착한다.

방정식	치환
$\text{bool} = \text{int}$	$t_x = \text{bool}$
$t_1 = \text{int}$	$t_1 = \text{int}$
	$t_0 = \text{int}$

방정식 $\text{bool} = \text{int}$를 만족하는 해는 존재할 수 없으므로 동일화 알고리즘은 더 이상 진행하지 않고 실패하게 된다. 주어진 프로그램에 대해서 타입을 추론할 수 없다는 뜻이다. 실제로 타입 오류가 있는 경우이다.

실행 중에 타입 오류가 발생하지 않지만 타입 추론에 실패하는 경우를 보자. 아래 프로그램에 대해서

$$\underbrace{\texttt{fun}\ (\underbrace{f}_{t_f})\ \underbrace{(\texttt{iszero}\ \underbrace{(f\ f)}_{t_2})}_{t_1}}_{t_0}$$

해를 구해보면 다음과 같은 상황을 만나게 된다.

방정식	치환
$t_f = t_f \to \text{int}$	$t_0 = t_f \to \text{bool}$
	$t_1 = \text{bool}$
	$t_2 = \text{int}$

우리가 현재 고려하고 있는 타입은 아래의 문법으로 만들어지는데

$$T \to \text{int} \mid \text{bool} \mid T \to T \mid \alpha$$

이 집합에 속하는 어떤 타입도 방정식 $t_f = t_f \to \text{int}$를 만족할 수 없다. 따라서 이런 형태의 방정식을 마주치면 동일화 알고리즘은 타입 추론에 실패하고 종료하게 된다. 하지만 이 경우 함수 f가 다른 함수를 인자로 받아서 정수값을 반환하는 함수이면 위 프로그램은 오류없이 실행된다. 예를 들어, f가 `fun (g) 1`이면 문제없이 실행된다. 위 프로그램은 타입 오류를 포함하지 않음에도 불구하고 우리가 설계한 타입 시스템의 한계 때문에 받아들이지 못하는 경우이다.

위의 예들을 종합하여 타입 방정식의 해를 구하는 과정을 기술하면 다음과 같다. 타입 방정식에 있는 개별 방정식 $T_1 = T_2$에 대해 아래 과정을 거쳐 치환으로 옮기는 작업을 반복한다.

- 먼저 현재 치환을 방정식에 적용하여 이미 알고 있는 타입 변수의 내용을 갱신한다.
- 그 결과 현재 방정식이 $\text{int} = \text{int}$와 같이 항상 성립하는 꼴이 된다면, 이를 무시하고 다음 방정식으로 넘어간다.
- 그렇기 않고 $\text{bool} = \text{int}$와 같이 모순이 생긴 경우에는 타입 추론에 실패하는 경우이므로 알고리즘을 종료한다.
- 현재 방정식의 양쪽항이 $\text{int} \to t_1 = t_2 \to \text{bool}$와 같이 모두

변수가 아닌 경우, 어느 한쪽 항이 변수가 될 때까지 방정식을 간략화 시킨다.

- 만약 왼쪽항이 변수가 아니면 양쪽항을 바꾼다.
- 만약 $t = \ldots t \ldots$와 같이 왼쪽항의 변수가 오른쪽에 등장하면 타입 추론에 실패하고 알고리즘을 종료한다.
- 그렇지 않은 경우, 현재 방정식을 치환으로 옮기고 치환에 존재하는 방정식의 왼쪽에 해당하는 변수를 오른쪽 항으로 바꿔준다.

타입 방정식 풀이 알고리즘

위 과정을 정확하게 알고리즘으로 기술해보자. 치환은 다음과 같이 타입 변수에서 타입으로 가는 함수로 정의할 수 있다.

$$S \in Subst = TyVar \to T$$

치환 S를 타입에 적용하는 과정은 다음과 귀납적으로 정의된다. 치환에 존재하는 타입 변수들을 모두 해당 타입으로 바꾸는 연산이다.

$$\begin{aligned}
S(\text{int}) &= \text{int} \\
S(\text{bool}) &= \text{bool} \\
S(\alpha) &= \begin{cases} t & \text{if } \alpha \mapsto t \in S \\ \alpha & \text{otherwise} \end{cases} \\
S(T_1 \to T_2) &= S(T_1) \to S(T_2)
\end{aligned}$$

예를 들어, 다음과 같은 치환을

$$S = \{t_1 \mapsto \text{int}, t_2 \mapsto \text{int} \to \text{int}\}$$

타입 $(t_1 \to t_2) \to (t_3 \to \text{int})$에 적용하는 과정은 다음과 같다.

$$\begin{aligned}
&S((t_1 \to t_2) \to (t_3 \to \text{int})) \\
&= S(t_1 \to t_2) \to S(t_3 \to \text{int}) \\
&= (S(t_1) \to S(t_2)) \to (S(t_3) \to S(\text{int})) \\
&= (\text{int} \to (\text{int} \to \text{int})) \to (t_3 \to \text{int})
\end{aligned}$$

방정식 $t_1 \doteq t_2$을 현재 치환으로 옮기는 과정을 아래 함수 unify로 정의할 수 있다.

$$\mathsf{unify} : T \times T \times \mathit{Subst} \to \mathit{Subst}$$

$\mathsf{unify}(T_1, T_2, S)$는 방정식 $T_1 \doteq T_2$를 처리하여 치환 S로 옮겨준다. 그림 8.10에 unify를 정의하였다. 명시적으로 나타내지 않았지만 $\mathsf{unify}(\alpha, t, S)$의 경우 S를 확장(extend)하는 것은 기존 치환에 현재 방정식의 정보를 갱신하는 과정을 포함한다고 가정하자. 몇 가지 예를 들면 다음과 같다.

- $\mathsf{unify}(\alpha, \text{int} \to \text{int}, \emptyset) = \{\alpha \mapsto \text{int} \to \text{int}\}$
- $\mathsf{unify}(\alpha, \text{int} \to \alpha, \emptyset) = \text{fail}$
- $\mathsf{unify}(\alpha \to \beta, \text{int} \to \text{int}, \emptyset) = \{\alpha \mapsto \text{int}, \beta \mapsto \text{int}\}$

$$
\begin{aligned}
\mathsf{unify}(\mathrm{int}, \mathrm{int}, S) &= S \\
\mathsf{unify}(\mathrm{bool}, \mathrm{bool}, S) &= S \\
\mathsf{unify}(\alpha, \alpha, S) &= S \\
\mathsf{unify}(\alpha, t, S) &= \begin{cases} \text{fail} & \alpha \text{ occurs in } t \\ \text{extend } S \text{ with } \alpha \doteq t & \text{otherwise} \end{cases} \\
\mathsf{unify}(t, \alpha, S) &= \mathsf{unify}(\alpha, t, S) \\
\mathsf{unify}(t_1 \to t_2, t_1' \to t_2', S) &= \text{let } S' = \mathsf{unify}(t_1, t_1', S) \text{ in} \\
&\quad \text{let } S'' = \mathsf{unify}(S'(t_2), S'(t_2'), S') \text{ in} \\
&\quad\quad S'' \\
\mathsf{unify}(\_, \_, \_) &= \text{fail}
\end{aligned}
$$

그림 8.10: 동일화 알고리즘

- $\mathsf{unify}(\alpha \to \beta, \mathrm{int} \to \alpha, \emptyset) = \{\alpha \mapsto \mathrm{int}, \beta \mapsto \mathrm{int}\}$.

마지막 예제가 어떻게 동작하는지 이해하는 것이 중요하다. 그림 8.10의 알고리즘에서 $\mathsf{unify}(t_1 \to t_2, t_1' \to t_2', S)$에 해당하는 경우인데 S''를 구할 때 t_2와 t_2'에 대해서 S'을 먼저 적용해야 하는 이유를 보여준다.

타입 방정식 전체를 받아서 해를 구하는 알고리즘을 unifyall이라고 하자. 다음과 같이 정의된다.

$$\mathsf{unifyall} : \mathit{TyEqn} \to \mathit{Subst} \to \mathit{Subst}$$

$$
\begin{aligned}
\mathsf{unifyall}(\emptyset, S) &= S \\
\mathsf{unifyall}((t_1 \doteq t_2) \wedge u, S) &= \text{let } S' = \mathsf{unify}(S(t_1), S(t_2), S) \\
&\quad \text{in } \mathsf{unifyall}(u, S')
\end{aligned}
$$

타입 방정식 u가 주어졌을 때, unifyall의 두 번째 인자로는 항상 빈 치환을 주면 된다. 방정식 u를 받아서 해를 계산하는 알고리즘 $\mathcal{U}$를 다음과 같이 정의하자.

$$\mathcal{U}(u) = \mathsf{unifyall}(u, \emptyset)$$

최종 타입 추론 알고리즘

지금까지 정의한 알고리즘 $\mathcal{V}$와 $\mathcal{U}$를 이용하여 타입 체킹 알고리즘 typecheck를 다음과 같이 정의할 수 있다.

$$\begin{aligned}&\mathsf{typecheck}(E) = \\ &\quad \text{let } S = \mathcal{U}(\mathcal{V}(\emptyset, E, \alpha)) \quad (\text{new } \alpha) \\ &\quad \text{in } S(\alpha)\end{aligned}$$

가장 먼저 새로운 타입 변수(α)를 생성하고 이를 주어진 프로그램 E의 타입으로 가정한 후 방정식을 생성($\mathcal{V}(\emptyset, E, \alpha)$)하고 푼다($\mathcal{U}$). 방정식이 풀리는 경우 치환($S$)이 구해진다. 치환 S에서 α에 해당하는 타입($S(\alpha)$)을 찾아 반환한다.

타입 추론 알고리즘의 안전성과 완전성

위의 타입 추론 알고리즘은 타입 추론 규칙에 대해서 안전하면서 완전하다. 추론 알고리즘의 역할은 $\emptyset \vdash E : t$가 증명 가능한지를 판단하는 것인데, 모든 증명 가능한 $\emptyset \vdash E : t$에 대해서 알고리즘이 증명을 해낼 수 있고(완전성), 알고리즘이 추론에 성공하면 추

론 규칙에 의해서도 반드시 증명된다(안전성)는 뜻이다. 타입 추론 규칙을 충실하게 구현한 알고리즘임을 뜻한다.

이러한 타입 추론 알고리즘의 안전성과 완전성은 앞서 설명한 추론 규칙의 안전성과 완전성과는 다른 개념이다. 타입 추론 규칙의 안전성과 완전성은 실행 규칙에 대해서 정의되는 개념이지만 알고리즘의 경우는 타입 추론 규칙에 대해서 정의되는 개념이다.

실행 규칙 $\xleftrightarrow{\text{안전/불완전}}$ 타입 추론 규칙 $\xleftrightarrow{\text{안전/완전}}$ 타입 추론 알고리즘

8.7 다형 타입 시스템

지금까지의 타입 시스템을 단순 타입 시스템(simple type system)이라고 부른다. 단순 타입 시스템은 OCaml 등의 언어에서 실제로 쓰이는 타입 시스템에 비해서 정교함이 떨어지는데, 주된 차이점은 다형 타입(polymorphic type)을 지원하지 않는다는 것이다. 예를 들어, 아래 프로그램은 타입 오류가 없지만 우리 타입 시스템은 받아들이지 못하고 OCaml 타입 시스템은 받아들이는 대표적인 예이다.

```
let f = fun (x) x in
  if (f (iszero 0)) then (f 1) else (f 2)
```

단순 타입 시스템이 위 프로그램에 대해서 타입 체크를 하지 못하는 이유를 분석해보자.

- `let f = fun (x) x`에 대하여 타입 방정식을 생성할 때, 타입 변수 t_x와 t_f를 도입하여 방정식 $t_f = t_x \to t_x$를 생성한다.

이 방정식만 놓고 보면 함수 `f`의 타입이 만족해야 하는 제약은 한 가지밖에 없다. 변수 `x`의 타입을 t_x라고 할 때, `f`의 타입이 $t_x \to t_x$ 형태여야 한다는 것이다. 다시 말해, 인자 타입 t_x가 만족해야 하는 특별한 제약은 없고 단지 함수의 인자 타입과 결과 타입이 같아야 한다는 제약만 생성하였다.

- 직관적으로 이러한 제약 조건하에서 위의 프로그램은 타입 체크가 되어야 한다. `f`가 사용될 때마다 위의 제약 조건을 지키면서 전체 프로그램이 타입을 가지도록 할 수 있기 때문이다. 첫 번째 함수 호출식 `(f (iszero 0))`의 경우 $t_f = \text{bool} \to \text{bool}$이라고 하고, 두 번째와 세 번째 호출식에서 $t_f = \text{int} \to \text{int}$라고 하면, `f`에 대한 위의 제약 조건을 항상 만족하면서 조건식에 대한 타입 추론 규칙이 요구하는 제약 조건들도 모두 만족된다. 따라서 전체 프로그램의 타입을 int로 결론 내리는 데 논리적으로 문제가 없다.

- 그럼에도 불구하고 단순 타입 시스템은 타입 추론에 실패하는 데, 그 이유는 `f`가 사용되는 서로 다른 지점들에서 동일한 타입 변수 t_x를 사용하여 방정식을 생성하기 때문이다. 첫 번째 함수 호출 지점에서 `f`의 인자로 부울 타입의 값이 주어지므로 방정식 $t_x = \text{bool}$이 생성되고, 두 번째와 세 번째 호출 지점에서는 `f`의 인자로 정수 타입의 값이 주어졌으므로 방정식 $t_x = \text{int}$가 생성된다. 이 두 방정식을 결합하면 모순$(\text{bool} = \text{int})$이 발생하고 타입 추론에 실패하게 된다.

위 문제를 해결하는 자연스런 방법은 f가 사용되는 서로 다른 지점마다 새로운 타입 변수를 도입하는 것이다. 첫 번째 호출 지점에서 $t_f = t_x \to t_x$라고 했으면 두 번째 호출 지점에서는 새로운 타입 변수 t'_x을 도입하여 $t_f = t'_x \to t'_x$로 두고 방정식을 생성하는 것이다. 그 결과 $t_x = \text{bool}$과 $t'_x = \text{int}$와 같은 방정식이 만들어지며 논리적으로 관련없는 두 방정식이 서로 엮여서 모순을 만들어내는 문제를 피할 수 있다. 이와 같이 서로 다른 함수 호출 지점을 구별하는 방식을 정적 분석에서 일반적으로 문맥 구분 분석(context-sensitive analysis)이라고 부르며, 타입 시스템의 경우에는 다형 타입 시스템(polymorphic type system)이라고 부른다. 이러한 타입 시스템을 장착한 언어에서는 임의의 타입에 대하여 동작하는 함수(polymorphic function)를 정의하고 사용할 수 있게 된다.

단순 타입 시스템에서 다형 타입 시스템으로의 확장에 대한 기본 아이디어는 간단하지만 정교함의 정도에 따라서 다양한 타입 시스템을 디자인할 수 있다. OCaml에서 사용하는 타입 시스템은 let 다형 타입 시스템(let-polymorphic type system)이라 불리는 것으로서 let식에 의해 정의되는 함수만 다형 함수로 일반화하는 시스템이다.

8.8 구현

이 장에서 설계한 타입 시스템을 구현해보자. OCaml로 정의한 다음 언어를 생각하자.

```
type exp =
```

```
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
and var = string
```

타입, 타입 환경, 치환은 다음과 같이 구현할 수 있다.

```
type typ =
  TyInt
  | TyBool
  | TyFun of typ * typ
  | TyVar of tyvar
and tyvar = string

let tyvar_num = ref 0
let fresh_tyvar () =
  (tyvar_num := !tyvar_num + 1;
   (TyVar ("t" ^ string_of_int !tyvar_num)))

(* type environment : var -> type *)
module TEnv = struct
  type t = var -> typ
  let empty =
    fun _ -> raise (Failure "Type Env is empty")
  let extend (x,t) tenv =
    fun y -> if x = y then t else (tenv y)
  let find tenv x = tenv x
end
```

```
(* substitution *)
module Subst = struct
  type t = (tyvar * typ) list
  let empty = []
  let find x subst = List.assoc x subst

  let rec apply : typ -> t -> typ
  =fun typ subst ->
    match typ with
    | TyInt -> TyInt
    | TyBool -> TyBool
    | TyFun (t1,t2) ->
      TyFun (apply t1 subst, apply t2 subst)
    | TyVar x ->
      try find x subst
      with _ -> typ

  let extend tv ty subst =
    (tv,ty) ::
    (List.map (fun (x,t) ->
      (x, apply t [(tv,ty)])) subst)
end
```

타입 방정식을 다음과 같이 정의하자.

```
type typ_eqn = (typ * typ) list
```

타입 방정식을 생성하고 해를 구하는 과정을 아래 함수들로 구현해보자.

```
let rec gen_equations : TEnv.t -> exp -> typ -> typ_eqn
=fun tenv e ty -> (* TODO *)

let solve : typ_eqn -> Subst.t
=fun eqn -> (* TODO *)
```

최종 타입 체커 typecheck는 이들 함수를 가지고 쉽게 구현할 수 있다.

```
let typecheck : exp -> typ
=fun exp ->
  let new_tv = fresh_tyvar () in
  let eqns = gen_equations TEnv.empty exp new_tv in
  let subst = solve eqns in
  let ty = Subst.apply new_tv subst in
    ty
```

타입 시스템 확장

이 장에서는 간단한 언어에 대하여 타입 시스템을 설계하고 구현해보았다. 이를 확장하여 5장에서 정의한 언어 Fun에 대해서 타입 추론 규칙을 설계하고 타입 체커를 구현해보자.

9

람다 계산식

프로그래밍 언어의 기원은 알론조 처치(Alonzo Church)가 1936년에 발표한 람다 계산법(lambda calculus)이다. 람다 계산법은 프로그래밍 언어의 필수 기능만 가지고 있는 가장 작고 단순한 형태의 프로그래밍 언어라고 할 수 있다. 람다 계산식을 소개하며 이 책을 마친다.

9.1 람다 계산식

4장에서 `let`식은 다음과 같이 함수 호출식으로 다시 쓸 수 있는 설탕구조(syntactic sugar)라고 했다.

$$\texttt{let}\ x = E_1\ \texttt{in}\ E_2 \overset{desugar}{\Longrightarrow} (\texttt{fun}\ x\ E_2)\ E_1$$

설탕구조는 프로그래밍 언어에서 필수는 아니다. 프로그래밍 언어가 `let`을 지원하지 않는다 하더라도 작성할 수 있는 프로그램의 개수가 줄어들지는 않는다.

한 가지 흥미로운 질문이 있을 수 있다. 프로그래밍 언어에서 설탕구조를 모두 제거한다면 어떤 모습이 될까? 예를 들어, 4장에서 정의한 그림 9.1의 언어를 생각해보자. 이 언어의 문법구조 중 어떤 것들이 설탕일까? 설탕구조를 모두 제거한, 필수 요소들만 가

지는 프로그래밍 언어는 어떻게 생겼을까?

놀랍게도 프로그래밍 언어를 구성하는 대부분이 설탕이다. 그림 9.1의 언어에서 정수(n)도 설탕이고, 모든 사칙 연산도 설탕이다. 조건식, 재귀 함수도 설탕이어서 이들 없이도 임의의 조건과 반복을 표현할 수 있다. 설탕이 아닌 문법구조는 변수(x), 함수 정의(`fun` x E), 함수 호출(E_1 E_2)뿐이다. 모든 설탕구조를 제거한 후의 프로그래밍 언어는 다음과 같이 생겼다.

$$\begin{array}{rcl} E & \rightarrow & x \\ & | & \texttt{fun}\ x\ E \\ & | & E_1\ E_2 \end{array}$$

이 언어가 람다 계산식이다. 일반 프로그래밍 언어로 작성할 수 있는 모든 프로그램을 람다 계산식으로 작성할 수 있으므로 람다 계산식은 튜링 완전성(Turing-completeness)을 가진 가장 작은 프로그래밍 언어라 할 수 있다.

문법구조

람다 계산식은 매우 간단한 문법구조와 의미구조를 가진다. 전통적인 람다 계산식은 다음과 같은 문법구조를 가진다.

$$\begin{array}{rcll} E & \rightarrow & x & \text{변수} \\ & | & \lambda x.E & \text{함수 정의} \\ & | & E_1\ E_2 & \text{함수 호출} \end{array}$$

$$
\begin{array}{rcl}
E & \rightarrow & n \\
& | & x \\
& | & E_1 \texttt{ + } E_2 \\
& | & E_1 \texttt{ - } E_2 \\
& | & E_1 \texttt{ * } E_2 \\
& | & \texttt{iszero } E \\
& | & \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \\
& | & \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \\
& | & \texttt{letrec } f(x) \texttt{ = } E_1 \texttt{ in } E_2 \\
& | & \texttt{fun } x\ E \\
& | & E_1\ E_2
\end{array}
$$

그림 9.1: 4장에서 정의한 프로그래밍 언어

함수 정의식 `fun` $x\ E$를 $\lambda x.E$와 같이 표현한 것만 다르다. 예를 들어, 아래 식들은 모두 위의 문법으로 만들 수 있는 람다식들이다.

$$
\begin{array}{cccc}
 & x & y & z \\
\lambda x.x & & \lambda x.y & \lambda x.\lambda y.x \\
x\ y & (\lambda x.x)\ z & x\ \lambda y.z & ((\lambda x.x)\ \lambda x.x)
\end{array}
$$

괄호를 충분히 사용하지 않으면 람다식에서 함수 몸통의 범위가 명확하지 않을 수 있다. 예를 들어 람다식 $\lambda x.\lambda y.x\ y$가 $\lambda x.\lambda y.(x\ y)$와 $(\lambda x.\lambda y.x)\ y$ 중에서 무엇을 의미하는지 애매하다. 이런 경우 최대한 함수의 몸통으로 해석하는 것이 관례이다. 람다식 $\lambda x.\lambda y.x\ y$를 $\lambda x.\lambda y.(x\ y)$로 해석하는 것이다.

람다식에 대해서도 자유 변수(free variable)와 묶인 변수(bound variable)를 정의할 수 있다. 어떤 변수 x에 대해서 그 변수를 도입한 λx가 존재하면 묶인 변수라고 부른다. 그 외의 변수들은 자유

변수이다. 예를 들어,

- 람다식 $\lambda y.(x\ y)$에서 x는 자유변수이고 y는 묶인 변수이다.
- 람다식 $\lambda z.\lambda x.\lambda y.(y\ z)$에서 변수 y와 z는 모두 묶인 변수이다.
- 람다식 $(\lambda x.x)\ x$에서 첫 번째 x는 묶인 변수, 두 번째 x는 자유 변수이다.

람다식의 계산

람다식 E를 계산하려면 다음 두 단계를 반복하면 된다.

1. 주어진 람다식 E에서 다음과 같은 형태의 부분식을 찾는다.

$$(\lambda x.E_1)\ E_2$$

   함수 $\lambda x.E_1$을 인자 E_2를 가지고 호출하는 형태인데, 이러한 부분식을 레덱스(redex[1])라고 부른다.

   예를 들어, 람다식 $(\lambda x.x)\ y$는 전체식이 레덱스인 경우이고, 람다식 $\lambda x.((\lambda y.y)\ z)$에서는 부분식 $(\lambda y.y)\ z$가 레덱스이다.

2. 레덱스를 다음과 같이 다시 쓴다. β-계산(β-reduction)이라 불리는 과정이다.

$$(\lambda x.E_1)\ E_2 \rightarrow [x \mapsto E_2]E_1$$

1) reducible expression

여기서 $[x \mapsto E_2]E_1$은 식 E_1에 등장하는 변수 x를 식 E_2로 치환한 식을 뜻한다. 예를 들어, 위의 두 람다식들은 다음과 같이 계산된다.

$$(\lambda x.x)\ y \to [x \mapsto y]x = y$$
$$\lambda x.((\lambda y.y)\ z) \to \lambda x.([y \mapsto z]y) = \lambda x.z$$

위 과정을 레덱스가 더 이상 존재하지 않을 때까지 반복한다. 레덱스가 존재하지 않아서 더 이상 계산이 불가능한 람다식을 정상식(normal term)이라 부른다.

치환 $[x \mapsto E_2]E_1$을 정확하게 정의해보자. E_1의 구조에 따라 다음과 같이 재귀적으로 정의된다.

$$\begin{aligned}
[x \mapsto E_2]x &= E_2 \\
[x \mapsto E_2]y &= y \quad (x \neq y) \\
[x \mapsto E_2](\lambda y.E_1') &= \lambda z.[x \mapsto E_2]([y \mapsto z]E_1') \quad (\text{new } z) \\
[x \mapsto E_2](E_1'\ E_1'') &= ([x \mapsto E_2]E_1'\ [x \mapsto E_2]E_1'')
\end{aligned}$$

E_1이 함수 정의식$(\lambda y.E_1')$인 경우가 중요하다. E_1'에서 변수 x를 E_2로 치환할 때, 묶여 있는 변수가 치환되서는 안되고 의도치 않게 변수가 새롭게 묶여서도 안 된다. 이를 위해 함수의 인자 이름 y를 전체 람다식에 등장하지 않는 새로운 이름 z로 변경한 후 치환 $[x \mapsto E_2]$를 적용하고 있다.

예를 들어, 람다식 $(\lambda x.(\lambda x.x))\ y$의 계산은 다음과 같이 진행된

다.

$$\begin{aligned}(\lambda x.(\lambda x.x))\ y &\rightarrow [x \mapsto y](\lambda x.x) \\ &= \lambda z.[x \mapsto y]([x \mapsto z]x) \quad (\text{new } z) \\ &= \lambda z.[x \mapsto y](z) \\ &= \lambda z.z\end{aligned}$$

$[x \mapsto y](\lambda x.x)$를 계산할 때 $\lambda x.x$ 내의 x는 자유 변수가 아니므로 치환 $[x \mapsto y]$에 의해 이름이 바뀌면 안 된다. 이를 방지하고자 치환을 함수의 몸통에 적용하기 전에 묶여 있는 변수의 이름을 새로운 이름으로 변경하였다. 함수에서 인자의 이름은 의미에 영향을 주지 않으므로 최종 결과가 $\lambda z.z$이든 $\lambda x.x$이든 의미는 동일하다. 위의 치환 과정에서 새로운 이름 z를 도입하지 않았다면 다음과 같이 계산되었을 것이다.

$$\begin{aligned}(\lambda x.(\lambda x.x))\ y &\rightarrow [x \mapsto y](\lambda x.x) \\ &= \lambda x.[x \mapsto y](x) \\ &= \lambda x.y\end{aligned}$$

최종 결과로 인자를 그대로 반환하는 함수 $\lambda x.x$가 계산되어야 하는데, 다른 의미를 가지는 함수 $\lambda x.y$로 잘못 계산되었다.

이와 같이 인자 이름을 새로 지으면 의도치 않게 변수가 묶이는 일도 방지할 수 있다. 예를 들어, 람다식 $(\lambda x.(\lambda y.(x\ y)))\ y$를 계산할 때 묶인 변수 y의 이름을 새로 짓지 않으면 계산 결과는 $\lambda y.(y\ y)$가 된다. 하지만 $\lambda y.(x\ y)$ 내의 y와 인자로 주어진 y는 서로 관련이

없는 변수들이다. 관련 없는 변수가 치환으로 인해 묶여서 관련이 생겼으니 잘못된 결과이다. $\lambda y.(x\ y)$에서 y를 새로운 이름 z로 바꾸고 계산하면 $\lambda z.(y\ z)$를 얻고 자유 변수가 의도치 않게 묶인 변수가 되는 일을 방지할 수 있다.

람다식 E에 여러 개의 레덱스가 존재할 수 있다. 예를 들어, 아래 람다식은 세 개의 레덱스를 포함하고 있다.

$$\overbrace{\lambda x.x\ \ \overbrace{(\lambda x.x\ \ \overbrace{(\lambda z.\underbrace{(\lambda x.x)}_{redex3}\ z)}^{redex2})}^{redex1}}$$

일반적으로 여러 개의 레덱스 중에서 어떤 것을 선택하여 먼저 계산하느냐에 따라 람다식의 계산이 정상식에 도달하여 끝날 수도 있고 끝나지 않을 수도 있다. 정상식이 존재하는 람다식의 경우, 그 정상식에 항상 도달하려면 정상 순서(normal-order)로 계산하면 된다. 정상 순서는 가장 바깥, 가장 왼쪽에 있는 레덱스를 먼저 계산하는 방법이다. 예를 들어, 위 람다식을 정상 순서로 계산하는 과정은 다음과 같다. 각 단계에서 계산되는 레덱스를 밑줄로 표시하였다.

$$\begin{aligned}
&\underline{\lambda x.x\ (\lambda x.x\ (\lambda z.(\lambda x.x)\ z))}\\
&\rightarrow (\lambda x.x\ \underline{(\lambda z.(\lambda x.x)\ z)})\\
&\rightarrow (\lambda z.\underline{(\lambda x.x)}\ z)\\
&\rightarrow \lambda z.z
\end{aligned}$$

계산 순서가 달라지더라도 람다식의 계산이 끝나는 경우라면 그 결

과는 항상 같다.[2] 예를 들어, 위 람다식을 가장 안쪽의 레덱스부터 계산해도 그 결과는 $\lambda z.z$가 된다.

$$\begin{aligned}
&\lambda x.x\ (\lambda x.x\ (\lambda z.\underline{(\lambda x.x)\ z}))\\
&\rightarrow \lambda x.x\ (\underline{\lambda x.x\ (\lambda z.z)})\\
&\rightarrow \underline{\lambda x.x\ (\lambda z.z)}\\
&\rightarrow \lambda z.z
\end{aligned}$$

9.2 람다식으로 변환하기

그림 9.1의 언어에서 설탕구조를 모두 제거하면 람다 계산식이 된다고 하였다. 그림 9.1의 언어로 작성된 프로그램 E가 주어졌을때 이를 동일한 의미를 가지는 람다 계산식으로 변환하는 과정을 정의해보자. 식 E를 람다식으로 변환한 결과를 $\underline{E}$로 표현하기로 하자. 변환 규칙을 그림 9.2와 같이 재귀적으로 정의할 수 있다.

먼저 부울값 *true*와 *false*를 각각 함수 $\lambda t.\lambda f.t$와 $\lambda t.\lambda f.f$로 정의하였다. *true*는 두 개의 인자 t와 f를 받아서 첫 번째 인자 t를 반환하는 함수로, *false*는 두 번째 인자 f를 반환하는 함수로 표현하기로 약속한 것이다. 조건식 `if` E_1 `then` E_2 `else` E_3는 $\underline{E_1}$ $\underline{E_2}$ $\underline{E_3}$와 같이 변환하였다. E_1을 변환하여 계산하면 궁극적으로 부울값을 의미하는 함수 $\lambda t.\lambda f.t$ 또는 $\lambda t.\lambda f.f$가 얻어질 것이다. 따라서 E_1의 원래 의미가 *true*였다면 $\underline{E_2}$가 계산될 것이고, E_1의 의미가 *false*라면 $\underline{E_3}$가 계산될 것이다. 예를 들어, 식 `if` *true* `then` 0 `else` 1을

2) Church-Rosser Theorem 또는 Confluence Theorem으로 알려져 있다.

$$
\begin{array}{rcl}
\underline{\mathit{true}} & = & \lambda t.\lambda f.t \\
\underline{\mathit{false}} & = & \lambda t.\lambda f.f \\
\underline{0} & = & \lambda s.\lambda z.z \\
\underline{1} & = & \lambda s.\lambda z.(s\ z) \\
\underline{n} & = & \lambda s.\lambda z.(s^n\ z) \\
\underline{x} & = & x \\
\underline{E_1 + E_2} & = & (\lambda n.\lambda m.\lambda s.\lambda z.m\ s\ (n\ s\ z))\ \underline{E_1}\ \underline{E_2} \\
\underline{\texttt{iszero}\ E} & = & (\lambda m.m\ (\lambda x.\underline{\mathit{false}})\ \underline{\mathit{true}})\ \underline{E} \\
\underline{\texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3} & = & \underline{E_1}\ \underline{E_2}\ \underline{E_3} \\
\underline{\texttt{let}\ x = E_1\ \texttt{in}\ E_2} & = & (\lambda x.\underline{E_2})\ \underline{E_1} \\
\underline{\texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2} & = & \underline{\texttt{let}\ f = Y\ (\lambda f.\lambda x.E_1)\ \texttt{in}\ E_2} \\
\underline{\texttt{fun}\ x\ E} & = & \lambda x.\underline{E} \\
\underline{E_1\ E_2} & = & \underline{E_1}\ \underline{E_2}
\end{array}
$$

그림 9.2: 람다 계산식으로 변환하는 규칙

변환하고 계산하면 다음과 같이 0을 의미하는 람다식이 계산된다.

$$
\begin{array}{rcl}
\underline{\texttt{if}\ \mathit{true}\ \texttt{then}\ 0\ \texttt{else}\ 1} & = & \underline{\mathit{true}}\ \underline{0}\ \underline{1} \\
& = & (\lambda t.\lambda f.t)\ \underline{0}\ \underline{1} \\
& \rightarrow & (\lambda f.\underline{0})\ \underline{1} \\
& \rightarrow & \underline{0}
\end{array}
$$

n이 자연수인 경우를 생각해보자.[3] 그림 9.2에서 자연수 n을 람다식 $\lambda s.\lambda z.(s^n\ z)$로 표현하였다. 여기서 $s^n\ z$는 다음과 같이 정

3) 정수는 자연수를 가지고 인코딩할 수 있는데 여기서는 생략한다.

$$
\begin{array}{rcl}
\underline{1+2} & = & (\lambda n.\lambda m.\lambda s.\lambda z.m\ s\ (n\ s\ z))\ \underline{1}\ \underline{2} \\
& \rightarrow & (\lambda m.\lambda s.\lambda z.m\ s\ (\underline{1}\ s\ z))\ \underline{2} \\
& \rightarrow & \lambda s.\lambda z.\underline{2}\ s\ (\underline{1}\ s\ z) \\
& = & \lambda s.\lambda z.\underline{2}\ s\ (\lambda s.\lambda z.(s\ z)\ s\ z) \\
& \rightarrow & \lambda s.\lambda z.\underline{2}\ s\ (\lambda z.(s\ z)\ z) \\
& \rightarrow & \lambda s.\lambda z.\underline{2}\ s\ (s\ z) \\
& = & \lambda s.\lambda z.(\lambda s.\lambda z.(s\ (s\ z)))\ s\ (s\ z) \\
& \rightarrow & \lambda s.\lambda z.(\lambda z.(s\ (s\ z)))\ (s\ z) \\
& \rightarrow & \lambda s.\lambda z.s\ (s\ (s\ z)) \\
& = & \underline{3}
\end{array}
$$

그림 9.3: $1+2$를 람다식으로 변환하고 계산하는 과정

의되는 람다식이다.

$$
\begin{array}{rcl}
s^0\ z & = & z \\
s^n\ z & = & s\ (s^{n-1}\ z)
\end{array}
$$

예를 들어, 자연수 0은 두 인자 s와 z를 받아서 z를 반환하는 함수로 인코딩된다. 자연수 1은 인자 s를 z에 한 번 적용하는 함수로, 2는 s를 z에 두 번 적용하는 함수로 인코딩된다. 일반적으로 n은 s를 z에 n번 적용하는 함수가 된다. 이와 같이 자연수를 표현하기로 약속하면 덧셈 $E_1 + E_2$는 아래와 같이 변환할 수 있다.

$$
\underline{E_1 + E_2} = (\lambda n.\lambda m.\lambda s.\lambda z.m\ s\ (n\ s\ z))\ \underline{E_1}\ \underline{E_2}
$$

예를 들어, 식 $1+2$를 람다식으로 변환하고 계산하면 그림 9.3과 같이 3을 뜻하는 람다식이 얻어진다.

식 `iszero` E는 $(\lambda m.m\ (\lambda x.\underline{\mathit{false}})\ \underline{\mathit{true}})\ \underline{E}$로 변환한다. E의

타입은 정수이므로 E를 람다식으로 변환($\underline{E}$)하고 계산하면 궁극적으로 $\lambda s.\lambda z.(s^k\ z)$꼴의 함수가 나올 것이다. 이를 m이라 하고 함수 호출식 $m\ (\lambda x.\underline{\mathit{false}})\ \underline{\mathit{true}}$를 계산해보자. k가 0이면 $\underline{\mathit{true}}$가, k가 0이 아니면 $\underline{\mathit{true}}$를 $\lambda x.\underline{\mathit{false}}$에 k번 적용한 결과를 얻는다. 그 결과는 임의의 k에 대해서 $\underline{\mathit{false}}$이다.

임의의 재귀 함수 $\texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2$를 아래의 비재귀 함수로 변환할 수 있다.

$$\texttt{let}\ f = Y\ (\lambda f.\lambda x.E_1)\ \texttt{in}\ E_2$$

여기서 Y는 Y-콤비네이터(combinator)라고 불리는 다음과 같이 생긴 람다식을 나타낸다.

$$Y = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

예를 들어, 팩토리얼(factorial)을 계산하는 재귀 함수

$$\texttt{letrec}\ f(n) = \text{if}\ n = 0\ \text{then}\ 1\ \text{else}\ n * f(n-1)$$

를 다음의 비재귀 함수로 변환할 수 있다.

$$\texttt{let}\ \text{fact} = Y(\lambda f.\lambda n.\text{if}\ n = 0\ \text{then}\ 1\ \text{else}\ n * f(n-1))$$

이렇게 정의한 후 fact n을 계산하면 $n!$이 나오게 된다.

$$
\begin{aligned}
&\text{fact } 1 \\
&= (Y\ F)\ 1 \\
&= \underline{(\lambda f.((\lambda x.f(x\ x))(\lambda x.f(x\ x)))\ F)}\ 1 \\
&= ((\lambda x.F(x\ x))(\lambda x.F(x\ x)))\ 1 \\
&= \underline{(G\ G)}\ 1 \\
&= (F\ (G\ G))\ 1 \\
&= \underline{(\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * (G\ G)(n-1))\ 1} \\
&= \underline{\text{if } 1 = 0 \text{ then } 1 \text{ else } 1 * (G\ G)(1-1))} \\
&= \underline{\text{if false then } 1 \text{ else } 1 * (G\ G)(1-1))} \\
&= \underline{1 * (G\ G)}(1-1) \\
&= 1 * (F\ (G\ G))(1-1) \\
&= 1 * \underline{(\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * (G\ G)(n-1))(1-1)} \\
&= 1 * \text{if } \underline{(1-1)} = 0 \text{ then } 1 \text{ else } (1-1) * (G\ G)((1-1)-1) \\
&= 1 * \text{if } \underline{0 = 0} \text{ then } 1 \text{ else } (1-1) * (G\ G)((1-1)-1) \\
&= 1 * \underline{\text{if } \mathit{true} \text{ then } 1 \text{ else } (1-1) * (G\ G)((1-1)-1)} \\
&= 1 * 1
\end{aligned}
$$

그림 9.4: 정상 순서로 fact 1을 계산하는 과정. Y, F, G의 정의를 이용하여 바꿔쓰는 단계가 아닌 경우, β-계산이 적용되는 레덱스를 밑줄로 표시하였다.

예를 들어, 그림 9.4은 fact 1을 정상 순서로 계산하는 과정을 보여준다. 계산 과정에서 F와 G는 아래 식들을 뜻한다.

$$
\begin{aligned}
F &= \lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1) \\
G &= \lambda x.F(x\ x)
\end{aligned}
$$

핵심적인 과정은 식 $G\ G$를 계산(β-reduction)하여 자기 자신을 품

고 있는 식인 $F\ (G\ G)$를 만들어내는 과정이다.

$$\begin{aligned} G\ G &= (\lambda x.F(x\ x))\ (\lambda x.F(x\ x)) \\ &\rightarrow F((\lambda x.F(x\ x))\ (\lambda x.F(x\ x))) = F(G\ G) \end{aligned}$$

Y-콤비네이터가 이러한 재귀적인 특성을 이미 가지고 있기 때문에 재귀 함수 없이 반복을 표현할 수 있다.

9.3 구현

이 장에서 대상으로 한 언어(그림 9.1)와 람다 계산식을 다음과 같이 OCaml 자료형으로 정의하자.

```
  type exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
and var = string

type lambda =
  | LVAR of var
  | LPROC of var * lambda
  | LCALL of lambda * lambda
```

1. 정상 순서로 람다식을 계산하는 함수 reduce를 작성해보자.

```
reduce : lambda -> lambda
```

2. 대상 언어를 람다식으로 변환하는 함수 translate를 작성해 보자.

```
translate : exp -> lambda
```

예를 들어,

```
reduce(translate (ADD(CONST 1, CONST 2)))
```

를 계산하면 3을 뜻하는 람다식인

```
LPROC ("s",
 LPROC ("z",
  LCALL (LVAR "s",
    LCALL (LVAR "s",
      LCALL (LVAR "s", LVAR "z")))))
```

가 나와야 한다(변수 이름들은 다를 수 있다).