SNU 4190.310 Programming Languages

Lecture Notes

Kwangkeun Yi[1]

School of Computer Science & Engineering

Seoul National University

2006

[1]Homepage: `cse.snu.ac.kr/~kwang`

머리말

이 노트는 컴퓨터 프로그래밍 언어에 대한 것입니다. 이 분야에서 축적된 성과와 문제들의 해결과정에 대한 이야기입니다. 학부 프로그래밍 언어 강의용입니다.

두개의 축

이야기의 내용은 두개의 축을 따라 구성되 있습니다. 가로 축은 프로그래밍 언어 자체에 대한 이야기 입니다. 세로 축은 생각하는 방식에 대한 이야기 입니다. 프로그래밍 언어를 공부할 때 사용하는 생각의 방식에 대한. 이 두 축을 기준으로 분포되는 좌표들이 내용을 구성합니다.

염려

바라건데 다음의 것들은 빠뜨리지 않고 전달하려고 합니다: 오랫동안 살아남을 기본기, 단단한 핵심.

그리고, 모국어로 강의한 내용입니다. 물론 전문 용어는 항상 영어를 *italic* 글꼴로 병기합니다.

> *그리고 내가 내 스승의 언어인 라틴어 대신에 나의 언어인 프랑스어로 쓰기로 한 것은, 태어나면서 가지고 있는 순수 이성만을 사용하는 사람들이 옛날 책만을 믿는 사람들보다 더 바르게 내 의견을 이해하리라고 믿고 있기 때문이다. 나는 양식을 가지고 꾸준히 노력하는 사람들만이 내 심판*

관이 되기를 바라거니와 그런 사람들은 내가 통속의 언어로 내 논거를 설명했다고 해서 이 논거를 들으려 하지 않을 만큼 라틴어를 편애하지는 않을 것이라고 믿는다. – 데카르트, *방법서설* (마지막에서 두번째 단락)

차 례

1 장 소개 **9**

1.1 프로그래밍 언어는 미개하다 10

1.2 프로그래밍 언어의 위치 11

2 장 기본기 **15**

2.1 *귀납법inductive definition* 15

2.1.1 집합의 정의 15

2.1.2 증명의 방법 24

2.2 *형식 논리formal logic* 27

2.2.1 모양과 뜻 27

2.2.2 추론규칙 29

2.2.3 안전한 혹은 완전한 31

3 장 모양과 뜻 **33**

3.1 *문법구조syntax* 33

3.1.1 요약된 혹은 구체적인 35

3.2 *의미구조semantics* 38

3.2.1 과정을 드러내는 38

3.2.2 궁극을 드러내는 48

4 장 기계 중심의 언어 **57**

4.1 주어진 기계 57

4.2 언어 키우기 0: K--- 59
4.2.1 변수: 메모리 주소에 이름 붙이기 59
4.2.2 이름의 두가지 의미 63
4.2.3 이름의 유효범위 64
4.2.4 환경 66
4.2.5 자유로운 혹은 묶여있는 69
4.2.6 설탕구조 70
4.3 언어 키우기 I: K-- 71
4.3.1 프로시저: 명령문에 이름 붙이기 71
4.3.2 이름의 유효 범위 73
4.3.3 프로시저 호출 방법 75
4.3.4 재귀 호출 76
4.3.5 명령문과 식의 통합 77
4.3.6 메모리 관리 80
4.4 언어 키우기 II: K- 82
4.4.1 레코드: 구조가 있는 데이타, 혹은 메모리 뭉치 82
4.4.2 포인터: 메모리 주소를 데이타로 85
4.4.3 메모리 관리 87
4.5 언어 키우기 III: K 94
4.5.1 실행되서는 않될 프로그램들 94
4.5.2 오류 검증의 문제 95
4.5.3 타입 시스템: 소개 99
4.5.4 K = K- + 타입 시스템 100
4.5.5 K- 타입 시스템의 논리적 문제 106
4.5.6 K- 타입 시스템의 구현 문제 112
4.5.7 이름 공간, 같은 타입 117
4.6 정리 118

5 장 값 중심의 언어 **121**

5.1 언어의 모델 121

5.1.1 *람다 계산법Lambda Calculus* 123

5.1.2 람다로 프로그램하기 130

5.2 언어 키우기 0: M0 133

5.2.1 *소극적인 실행lazy evaluation*과 *적극적인 실행eager evaluation* 134

5.2.2 시작 언어: 값은 오직 함수 136

5.3 언어 키우기 I: M1 138

5.3.1 다른 값: 정수, 참/거짓 138

5.4 언어 키우기 II: M2 140

5.4.1 구조가 있는 값: *쌍pair* 140

5.5 언어 키우기 III: M3 143

5.5.1 메모리 주소 값: 명령형 언어의 모습 143

5.6 오류 검증의 문제 145

5.6.1 *정적 타입 시스템static type system* 147

5.6.2 *형식 논리 시스템formal logic* 리뷰 149

5.7 *단순 타입 시스템simple type system* 152

5.7.1 단순 타입 추론 규칙 153

5.7.2 추론 규칙의 안전성 157

5.7.3 추론 규칙의 구현 160

5.7.4 안전한 그러나 경직된 170

5.7.5 M3로 확장하기 172

5.8 *다형 타입 시스템polymorphic type system* 175

5.8.1 다형 타입 추론 규칙 179

5.8.2 추론 규칙의 안전성 185

5.8.3 추론 규칙의 구현 185

5.8.4 M3로 확장하기 186

5.9 정리 190

6 장 번역과 실행 **193**

6.1 프로그램 입력 193

6.2 프로그램 실행 194

6.2.1 *실행기interpreter* 194

6.2.2 *번역기compiler* 195

6.2.3 자가발전 196

6.3 번역: *불변성질invariant* 유지하기 197

6.4 *가상의 기계virtual machine* 200

1 장

소개

이 강의를 통해서 다음과 같은 질문들에 대한 답을 익히거나, 좋은 답을 만들어내기 위해서 필요한 소양을 닦게 된다.

다양한 프로그래밍 언어들이 품고있는 공통된 원리들은 무엇인가? 현재의 프로그래밍 언어들은 얼마만큼 미개한가? 좀더 나아지기 위해서 필요한 것들은 무엇인가? 새로운 프로그래밍 환경을 효과적으로 운용할 수 있는 언어는 무엇인가?

이러한 질문들에 대한 답은 왜 필요한 것일까?

- 고난도 소프트웨어 시스템을 손쉽게 다룰 수 있는 방법은 프로그래밍 언어의 내용을 공부하면서 효과적으로 익혀진다. 소프트웨어 방법론 중에는 프로그래밍 언어 이론의 뒷받침이 없이는 공허해지는 것이 대부분이기 때문이다.

- 대부분의 소프트웨어들이 컴퓨터 언어를 처리하는 부품을 가지게 되는데, 이 부품들이 프로그래밍 언어에서 축적한 이론과 실제를 올바로 이해하고 설계될 필요가 있다. 성급한 상식의 수준에서 고안되는 프로그래밍 언어 시스템은 어느순간 무너져 버리는 다리와 같이 위태로울 수 밖에 없기 때문이다.

- 프로그래밍 언어의 연구성과들 덕택에 소프트웨어 개발이라는 것이 급

속도로 수학적이고 엄밀한 기술이 되어가고 있는데, 이에 우리가 프로그래밍 언어를 어떻게 해야 할 것인지를 고민해야 한다. 우리 소프트웨어 산업의 경쟁력을 창출할 근간으로써.

프로그래밍 언어의 성과, 문제, 그리고 그 생각의 틀을 이해하는 것이 *컴퓨터 과학/공학computer science, computer engineering*을 전공하는 데 중요한 밑거름이다.

1.1 프로그래밍 언어는 미개하다

현재의 프로그래밍 언어는 얼마나 미개할까?

프로그래밍 언어로 짜여진 소프트웨어를 기계가 실행해 준다. 우리가 고안한 기계가 소프트웨어를 자동으로 실행해 준다. 대단하고 놀라운 일이다.

더 놀라운 일은, 하나의 기계가 실행할 수 있는 프로그램들이 무수히 많이 있을 수 있다는 것이다. 이런 기계는 이 세상에 없었다. 자동차라는 기계는 "가고 서는" 프로그램만 실행해 줄 뿐이다. 자동차 페달을 밟아서 문서를 편집하거나 동영상을 상영하거나 에어백을 통제하는 등의 일을 할 수 는 없다.

그런데, 문제는 프로그래밍 언어로 짜여진 소프트웨어라는 인공물이 매우 위태로운 상태에서 실행된다는 점이다. 제대로 작동하는 물건인지를 확인하지 않고 실행되게 된다. 그 확인을 미리 자동으로 해 주는 기술이 아직은 미개하다.

다른 분야와 비교해 보자. 제대로 작동할 지를 미리 검증할 수 없는 기계설계는 없다. 제대로 서있을 지를 미리 검증할 수 없는 건축설계는 없다. 인공물들이 자연세계에서 문제 없이 작동할 지를 미리 엄밀하게 분석하는 수학적 기술들은 잘 발달해 왔다. 뉴튼 역학, 미적분 방정식, 통계 역학등등이 그러한 기술들일 것이다.

소프트웨어라는 인공물에 대해서는 어떠한가? 작성한 소프트웨어가 제대로 실행될 지를 미리 엄밀하게 확인해 주는 기술들은 있는가?

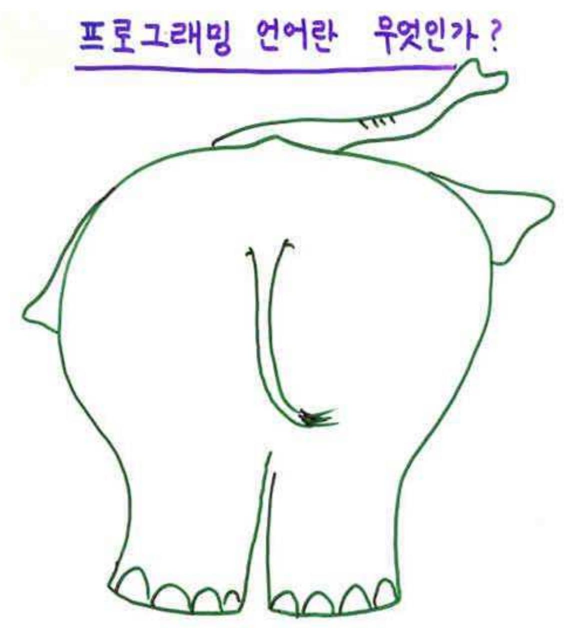
이 문제는 아직도 해결되지 않은 문제이다. 이 문제에 대한 답을 프로그래밍 언어에서도 해줘야 한다. 이 방향에서 지금 까지 연구된 것들이 아직은 미

흡하다.

지금까지 성공적인 연구결과가 어떤 모습으로 프로그래밍 언어에 장착되고 있는지 이 강의를 통해 살펴 볼 것이다. 그렇게 얻어진 시각으로 소프트웨어 기술의 현재수준을 간파하고, 보다나은 미래기술을 열어 갈 씨앗을 품게 되길.

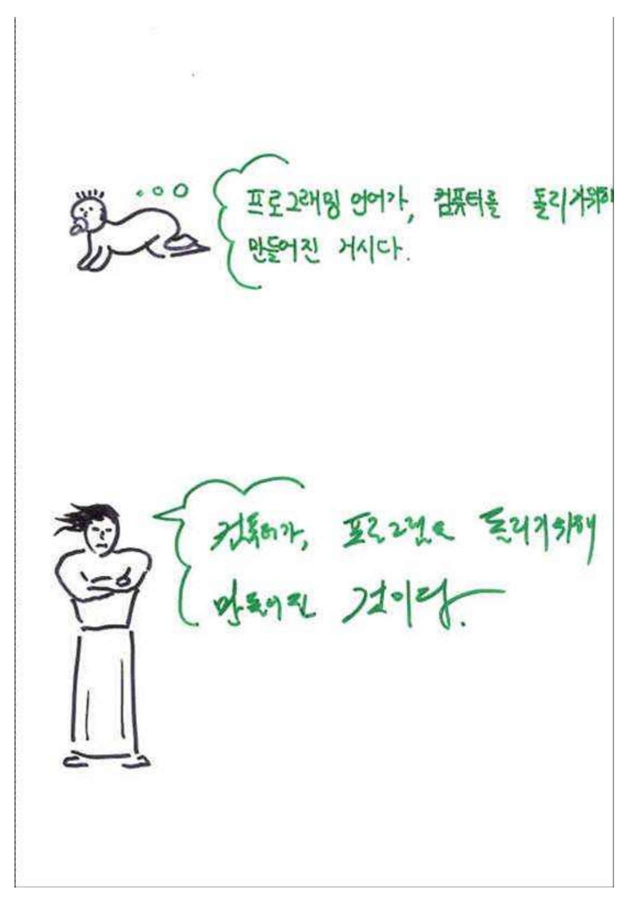
1.2 프로그래밍 언어의 위치

우선 프로그래밍 언어는 무엇인가? 에 대해서 다양하게 이야기 될 수 있다. 컴퓨터라는 기계에게 명령하는 도구, 프로그래머들 사이의 소통의 수단, 상위레벨의 디자인을 표현하는 도구, 알고리즘을 기술하는 표기법, 생각한 바를 실행시킬 수 있도록 표현하는 도구, 등등.



여담으로 오해 하나를 바로 잡자. 프로그래밍 언어가 컴퓨터를 돌리기위해 만들어진 것이 아니다. 컴퓨터가 프로그램을 돌리기위해 만들어진 것이다. 역사적으로 그렇다. 만들어진 컴퓨터를 사용하기위해 고안한 것이 프로그래밍 언어가 아니고, 그 전에 이미 논리학자와 수학자들이 프로그래밍 언어를 고안했다. 그리고 나서 전기회로로 구현된 디지탈 컴퓨터가 발명되었다.

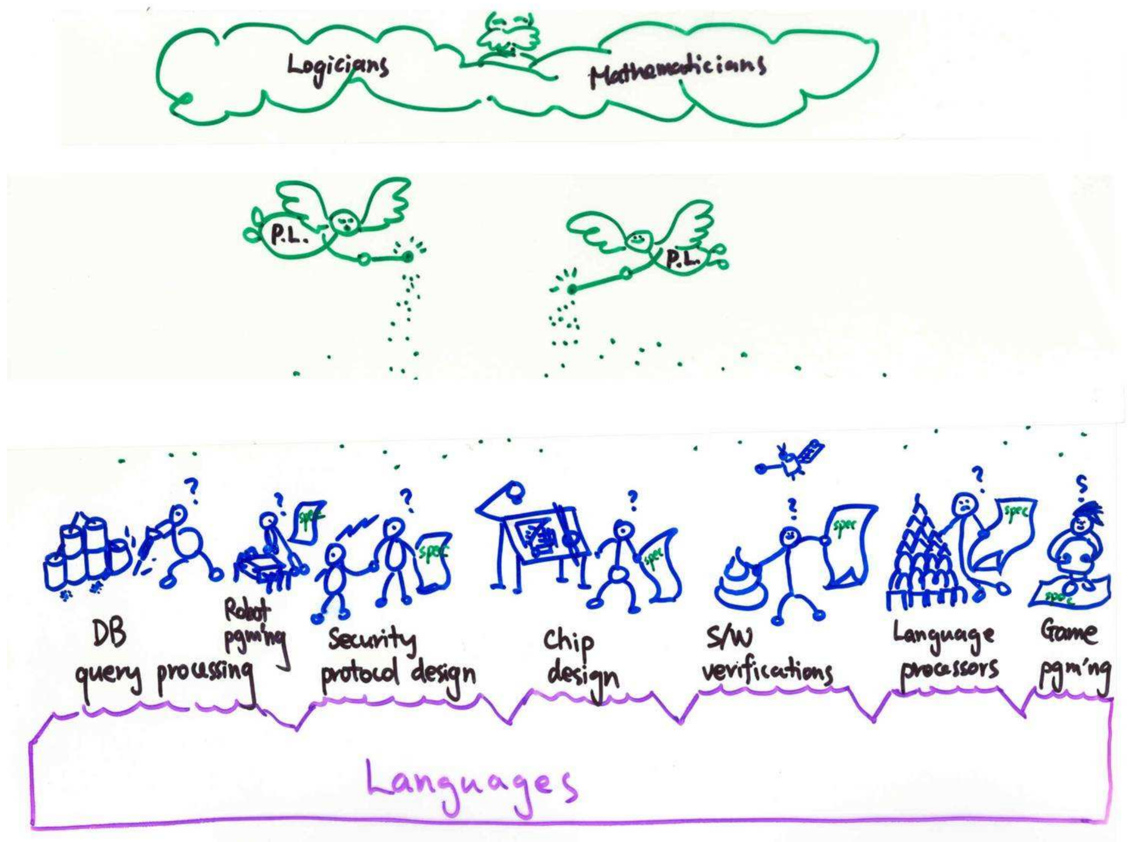
1930년대 말부터, 일군의 수학자들은 과연 "기계적으로 계산 가능한 함수"가 무엇일까를 고민했다. 하나의 아이디어로, 언어를 하나 고안해서 그 언어로 정의될 수 있는 것들을 기계적으로 계산 가능한 함수라고 하자, 고 제안하였다. 정확하게 현재의 디지탈 컴퓨터로 실행될 수 있는 모든 프로그램을 작성할 수 있는 언어들이었다. 놀랍게도. 지금의 디지탈 컴퓨터는 그러한 함수들을 자동으로 빨리 실행시켜주는 도구일 뿐이다.



실행해 주는 기계가 뭐가 되었던지간에, 프로그래밍 언어란 자동 실행을 염두에 두고 고안한 언어이다. 그 레벨이 기계어 수준이건 그 보다 훨씬 상위의

언어이건간에.

프로그래밍 언어의 효용을 조금 구체적으로 보자. 다음 그림에서 표현하였듯이, 모든 소프트웨어 시스템은 "언어"를 기반으로 한다. 소프트웨어가 외부 세계와 소통하는 데는, 어떤 형태든 항상 언어를 통해서 이루어지기 때문이다. 데이타베이스 시스템의 *질의 처리query processing*, 로봇 프로그래밍, 하드웨어 디자인, 커뮤니케이션 프로토콜 디자인, 소프트웨어 검증, 프로그래밍 언어 시스템(*번역기compiler*나 *실행기interpreter*), 게임 프로그래밍 등등 "언어"없이 작동하는 소프트웨어 시스템은 없다. 모든 것들은 언어 처리 시스템을 품고 있다.



따라서 모든 소프트웨어 시스템의 경쟁력의 일부분은 프로그래밍 언어 분야의 연구 성과에 기초해서 발전하게 된다. 프로그래밍 언어의 첨단 기술에 대한 이해가 없이는 로봇 프로그래밍 시스템/하드웨어 칩 디자인 시스템/프로토콜 디자인 시스템/소프트웨어 검증 시스템/게임 프로그래밍 시스템/데이타

베이스 시스템 등을 제대로 디자인하거나 구현하기가 어렵다. 모두 프로그래밍 언어 분야의 "축복"을 지렛대로 삼아 발전할 수 있다. 그리고, 프로그래밍 언어 분야 자체는 또, 논리학과 수학의 축복 아래에서만 단단하게 발전할 수 있다.

2 장

기본기

프로그래밍 언어를 이야기 하는데 기본적으로 사용하는 어휘들과 말하는 방식이 있다. 그 어휘와 방식의 기본은 수학이다. 이 장에서는 앞으로 우리가 사용할 어휘와 말하는 방식을 알아보고, 그 사용 예들을 살펴보자.

2.1 *귀납법inductive definition*

2.1.1 집합의 정의

집합을 정의하는 방법에는 조건제시법과 원소나열법, 그리고 하나가 더 있다. 귀납법이다.

귀납법은 즐겁다. 유한하게 무한한 것을 정의할 수 있기 때문이다. 유한한 갯수의 유한한 규칙들로 무한히 많은 원소를 가지는 집합을 정의하는 방법이 귀납법이다.

귀납법은 대개 증명의 한 방법이라고 배우지만, 사실 그 증명이 수긍이 가는 이유는 집합을 정의하는 귀납법의 얼개를 따라서 증명해 나가는 것이기 때문이다.

귀납 증명에 대해서는 다음장(2.1.2)에서 살펴보기로 하고 귀납이 집합을 정의하는 데 사용되는 것에 대해서 우선 살펴보자.

귀납법으로 집합을 정의하는 방법은, 그 이름이 의미하는데로, 되돌아서 바

치는 것이다(되돌아서 return 歸, 바치다 dedicate 納). 되돌아서 만들어 바친다는 것은, 그 집합의 원소를 가지고 그 집합의 원소를 만든다는 것이다. 귀납법으로 정의되는 집합은 몇개의 규칙들로 구성되고, 그 규칙들이 귀납적이다: 무엇무엇이 이미 집합의 원소들이라면 그것들로 어떻게어떻게 하면 다시 집합의 원소가 된다, 는 식이다.

2.1.1.1 그 집합은 이것이다

조금 일반적인 톤으로 이야기해보면 이렇다. 하나의 규칙은 짝으로 구성된다: 가정들 X와 결론 x. 그 규칙이 말하는 바는 "X에 있는 것들이 정의하려는 집합에 모두 있으면, x도 있어야 한다" 는 것이다. 이러한 규칙들이 모여서 하나의 집합을 정의한다. 그 집합은, 규칙들이 요구하는 원소들은 모조리 포함하고 있는 집합 가운데 가장 작은 집합으로 한다. 최소의 집합이라고 고집하는 이유는, 규칙들이 요구하는 원소들은 모두 포함하지만 그 밖의 것들은 제외하고 싶어서이다.

좀 더 명확하게 가보자. 가정들 X와 결론 x의 짝 (X, x)로 표현되는 규칙들의 모임을 Φ라고 하자.

그 규칙을 만족하는 원소들을 모두 품고 있는 집합을 "Φ에 대해서 닫혀있다" 혹은, "Φ-닫힘"이라고 한다. 즉, 집합 A가 Φ-닫힘, 은

$$(X, x) \in \Phi \text{ 이고 } X \subseteq A \text{ 이면 } x \in A$$

인 경우를 말한다. 이제, Φ가 정의하는 집합은 모든 Φ-닫힌 집합들의 교집합으로 정의된다:

$$\bigcap \{A | A\text{는 } \Phi\text{-닫힘}\}.$$

그러한 집합은 Φ-닫힌 집합이면서 가장 작다. (왜?)

Example 1 자연수의 집합은 다음 두개의 규칙들로 정의된다:

$$(\emptyset, 0) \quad (\{n\}, n+1)$$

즉, 0은 자연수이고, n이 자연수라면 $n+1$도 자연수이다. 이 규칙들에 대해서 닫혀있는 집합들은 우리가 알고 있는 자연수의 집합 $\mathbb{N}$ 뿐 아니라 유리수의 집합 $\mathbb{Q}$도 있다. 하지만 가장 작은 그러한 집합은 $\mathbb{N}$ 이다.□

Example 2 영문 소문자 알파벳으로 만들어 지는 스트링의 집합은 다음의 규칙들로 정의된다:

$$(\emptyset, \epsilon) \quad (\{\alpha\}, x\alpha \text{ for } x \in \{\mathsf{a}, \cdots, \mathsf{z}\})$$

즉, ϵ은 스트링이고, s가 스트링 이라면 그 앞에 영문 소문자를 하나 붙여도 스트링이다. 이 규칙들에 대해서 닫혀있는 가장 작은 집합이 이 규칙들이 정의하는 "스트링"이라는 집합이다. □

2.1.1.2 표기법

귀납법의 규칙을 나타내는 편리한 표기로 프로그래밍 언어에서는 다음 두 가지 것을 주로 쓴다. 위의 자연수 집합은:

$$n \;\rightarrow\; 0 \mid n+1$$

혹은

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

이고, 위의 스트링 집합은:

$$\begin{array}{rcl} \alpha & \rightarrow & \epsilon \\ & \mid & x\alpha \quad (x \in \{\mathsf{a}, \cdots, \mathsf{z}\}) \end{array}$$

혹은

$$\frac{}{\epsilon} \qquad \frac{\alpha}{x\alpha}\; x \in \{\mathsf{a}, \cdots, \mathsf{z}\}$$

으로 표현한다.

그리고, 귀납 규칙들은 반드시 되돌아서 표현될 필요는 없다. 집합이 유한하다면 재귀가 필요없이 원소나열법과 같이 주욱 쓰면 된다. 예를 들어, 집합

$\{1,2,3\}$을 규칙들로 표현하면

$$(\emptyset,1) \quad (\emptyset,2) \quad (\emptyset,3)$$

혹은

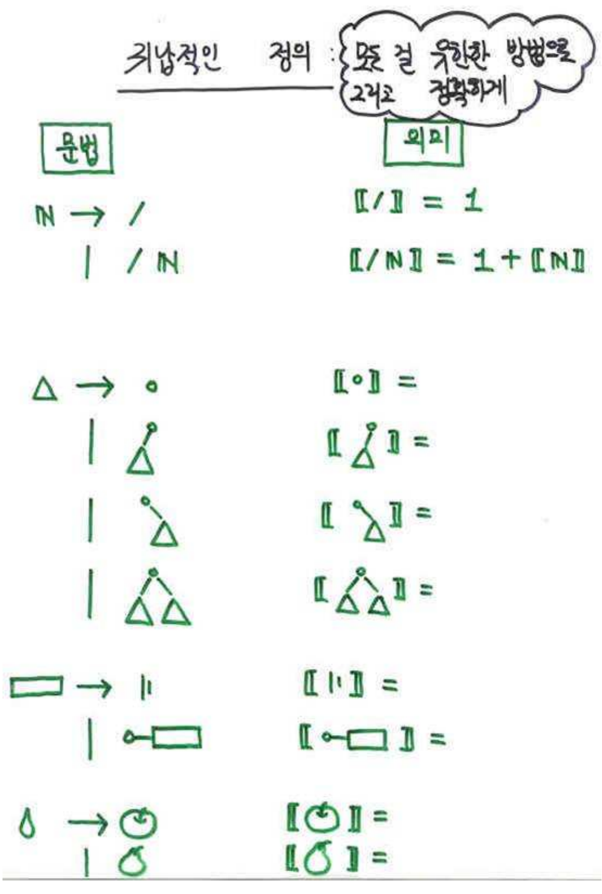
$$x \to 1 \mid 2 \mid 3$$

혹은

$$\overline{1} \quad \overline{2} \quad \overline{3}$$

가 된다.

다음 그림은 자연수 집합, *두갈래 나무binary tree* 집합, 리스트 집합, 사과와 배만 가지는 집합을 귀납적으로 정의한 것이다.



Example 3 리스트의 집합도 다음 두개의 규칙들로 정의된다:

$$\frac{}{\texttt{nil}} \qquad \frac{\ell}{\circ\!-\!\ell}$$

혹은

$$\ell \ \rightarrow\ \texttt{nil} \mid \circ\!-\!\ell$$

즉, nil은 리스트이고, ℓ이 리스트라면 그 앞에 하나의 노드를 붙인 $\circ\!-\!\ell$도 리스트이다. 이 규칙들에 대해서 닫혀있는 가장 작은 집합이 이 규칙들이 정의하는 "리스트"라는 집합이다. □

Example 4 말단에 정수를 가지는 *두갈래 나무binary tree*들의 집합은 다음 네개의 규칙들로 정의된다:

$$\frac{}{n}\ n \in \mathbb{Z} \qquad \frac{t}{\texttt{N}(t, \texttt{nil})}$$

$$\frac{t}{\texttt{N}(\texttt{nil}, t)} \qquad \frac{t_1 \quad t_2}{\texttt{N}(t_1, t_2)}$$

혹은

$$\begin{array}{rcl} t & \rightarrow & n \quad (n \in \mathbb{Z}) \\ & \mid & \texttt{N}(t, \texttt{nil}) \\ & \mid & \texttt{N}(\texttt{nil}, t) \\ & \mid & \texttt{N}(t, t) \end{array}$$

즉, 임의의 정수 n은 두갈래 나무이고, t가 두갈래 나무라면 그것을 왼편이나 오른편에 매단 것도 두갈래 나무이고, 두개의 두갈래 나무를 각각 왼편과 오른편에 매단 것도 두갈래 나무이다. 이 규칙들에 대해서 닫혀있는 가장 작은 집합이 이 규칙들이 정의하는 "두갈래 나무"라는 집합이다. □

Example 5 정수식들의 집합은 다음의 규칙들로 정의된다:

$$\frac{}{n}\ n \in \mathbb{N} \qquad \frac{e}{-e}$$

$$\frac{e_1 \quad e_2}{e_1{+}e_2} \qquad \frac{e_1 \quad e_2}{e_1{*}e_2}$$

혹은

$$\begin{array}{rcll} e & \to & n & (n \in \mathbb{N}) \\ & \mid & -e & \\ & \mid & e + e & \\ & \mid & e * e & \end{array}$$

즉, 자연수 n은 정수식이고, e가 정수식 이라면 그 앞에 음의 부호를 붙여도 정수식이고, 두개의 정수식 사이에 덧셈 부호나 곱셈 부호를 끼워넣어도 정수식이다. 이 규칙들에 대해서 닫혀있는 가장 작은 집합이 이 규칙들이 정의하는 "정수식"이라는 집합이다. □

Example 6 다음의 규칙들이 만들어내는 $\langle A, a\rangle$쌍들의 집합은 무엇일까?

$$\frac{}{\langle A, a\rangle}\ a \in A$$

$$\frac{\langle A, a\rangle \quad \langle A, b\rangle}{\langle A, a \cdot b\rangle} \qquad \frac{\langle A, a \cdot b\rangle}{\langle A, a\rangle}$$

$$\frac{\langle A \cup \{a\}, b\rangle}{\langle A, a^b\rangle} \qquad \frac{\langle A, a^b\rangle \quad \langle A, a\rangle}{\langle A, b\rangle}$$

□

2.1.1.3 그 집합은 이렇게 만든다

그런데, 위의 정의는 명확하지만 한가지가 의아하다. 규칙들이 정의하는 집합이 무엇인지 정의하고 있지만, 그 집합을 만드는 방법을 알려주고 있지 않다. (The definition is non-constructive.) 짜장면은 무엇이다라고 정의해 놓았지만,

짜장면을 만드는 방법은 가르치고 있지 않다! 만드는 방법을 알려주는 정의는 다음과 같다.

규칙들의 집합 Φ는 함수 ϕ를 정의한다. ϕ는 집합을 받아서 그 집합을 가지고 Φ 규칙들을 이용해서 만들어 지는 모든 원소들을 내 놓는다:

$$\phi(Y) = \{x \mid \frac{X}{x} \in \Phi,\ X \subseteq Y\}$$

Φ 규칙들이 정의하는 집합은 함수 ϕ에 의해서 닫혀있는 집합(ϕ가 더 이상 새로운 것을 만들어 내지 못하는 집합, 즉 $\phi(X) \subseteq X$인 집합 X)중에서 최소의 집합

$$\bigcap\{X|\phi(X) \subseteq X\} \tag{2.1}$$

이고, 이것은 함수 ϕ의 *최소의 고정점least fixed point*가 된다. (왜?)

함수 ϕ의 최소의 고정점인 이 집합은 빈 집합에서 부터 출발해서(ϕ^0라고 하자) ϕ를 한번 적용하고 나온 집합 ϕ^1, 두번 적용하고 나온 집합 ϕ^2등을 모두 모으면 만들어 진다. 즉,

$$\begin{array}{rcl}\phi^0 & = & \emptyset \\ \phi^n & = & \phi(\phi^{n-1}) \quad n \in \mathbb{N}\end{array}$$

라고하고

$$\phi^0 \cup \phi^1 \cup \phi^2 \cup \cdots = \bigcup_{i \in \mathbb{N}} \phi^i$$

런 식으로 만들어 지는 것이 Φ 규칙들이 정의하는 집합이 되는 것이다. 이렇게 만들어진 집합은 식 2.1과 일치한다. (왜?)

Example 7 자연수의 집합은 다음 두개의 규칙들로 정의된다:

$$n \ \rightarrow\ 0 \mid n+1$$

이 규칙들이 지정하는 집합 $\mathbb{N}$ 은 다음과 같이 만들어 진다:

$$\begin{array}{lcl} \phi^0 & = & \emptyset \\ \phi^1 & = & \{0\} \\ \phi^2 & = & \{0,1\} \\ \phi^3 & = & \{0,1,2\} \\ \cdots \end{array}$$

들의 합집합.□

Example 8 리스트의 집합도 다음 두개의 규칙들로 정의된다:

$$\ell \rightarrow \texttt{nil} \mid \circ\!-\!\ell$$

이 규칙들에 정의하는 "리스트"라는 집합은 다음과 같이 만들어 진다:

$$\begin{array}{lcl} \phi^0 & = & \emptyset \\ \phi^1 & = & \{\texttt{nil}\} \\ \phi^2 & = & \{\texttt{nil}, \circ\!-\!\texttt{nil}\} \\ \phi^3 & = & \{\texttt{nil}, \circ\!-\!\texttt{nil}, \circ\!-\!\circ\!-\!\texttt{nil}\} \\ \cdots \end{array}$$

들의 합집합.□

Example 9 *두갈래 나무binary tree*의 집합은 다음 네개의 규칙들로 정의된다:

$$\begin{array}{rcl} t & \rightarrow & \circ \\ & | & \texttt{N}(t, \texttt{nil}) \\ & | & \texttt{N}(\texttt{nil}, t) \\ & | & \texttt{N}(t, t) \end{array}$$

이 규칙들이 정의하는 “두갈래 나무”라는 집합은 다음과 같이 만들어 진다:

$$\begin{array}{rcl} \phi^0 & = & \emptyset \\ \phi^1 & = & \{\circ\} \\ \phi^2 & = & \{\circ, \mathtt{N}(\circ, \mathtt{nil}), \mathtt{N}(\mathtt{nil}, \circ), \mathtt{N}(\circ, \circ)\} \\ \cdots \end{array}$$

들의 합집합.□

Example 10 영문 소문자 알파벳으로 만들어 지는 스트링의 집합은 다음의 규칙들로 정의된다:

$$\begin{array}{rcll} \alpha & \to & \epsilon & \\ & | & x\alpha & (x \in \{\mathtt{a}, \cdots, \mathtt{z}\}) \end{array}$$

이 규칙들이 정의하는 집합은

$$\begin{array}{rcl} \phi^0 & = & \emptyset \\ \phi^1 & = & \{\epsilon\} \\ \phi^2 & = & \{\epsilon, \mathtt{a}\epsilon, \cdots, \mathtt{z}\epsilon\} \\ \cdots \end{array}$$

들의 합집합이다. □

Example 11 귀납규칙 Φ에 있는 모든 규칙 $\frac{X}{x}$가 X를 공집합으로 가진 게 없다면, Φ가 정의하는 집합은 공집합이 된다. 만들어 지는 시작이 제공되지 않았기 때문이다: $\phi^0 = \phi^1 = \cdots = \emptyset$.□

2.1.1.4 원소들의 순서

Φ 규칙들로 귀납적으로 만들어지는 집합

$$\bigcup_{i \in \mathbb{N}} \phi^i$$

의 모든 원소들은 ϕ를 i번 적용해서 만들어 진다. 원소들의 순서를 정의하기를, ϕ^i번째에 새롭게 만들어 지는 원소들의 순서를 i라고 하자. 원소들 마다 자기가 몇째번에 만들어 진 것이냐에 따라 순서가 있는 것이다.

i째 번에 만들어지는 원소들 x는 $i-1$째 번에 만들어졌던 원소들 X 덕택에 귀납 규칙 (X, x)에 의해서 만들어 진 것이다. $i-1$째 번의 원소들은 또 $i-2$째 번의 원소들 덕택에 만들어 진 것이고. 이 연쇄반응은 결국에는 바닥에 닿는다: 궁극에는 항상 0째 번에 만들어진 원소들이 씨가 된다.

이렇게 언젠가는 바닥을 드러내는 순서를 "*기초가 튼튼한 순서well-founded order*"라고 한다. 귀납법으로 정의한 집합은 항상 기초가 튼튼한 순서를 가지고 있다. 이 성질이 귀납법 증명의 기술을 가능하게 해 준다.(장 2.1.2)

2.1.1.5 우리가 다루는 귀납규칙

규칙들이 만드는 집합의 원소들은 규칙들을 유한번 적용해서 만들어지는 원소들로 생각한다. 또, 애매하지 않는 규칙들만 사용한다. 일반적으로, 한 집합을 정의하는 귀납 규칙들은 무한히 많을 수 도 있고, 애매한 경우도 있을 수 있다(두개의 다른 규칙 (X, x)와 (X', x)이 같은 원소 x에 대해서 말하는 경우).

2.1.2 증명의 방법

집합 S의 모든 원소들이 어떤 성질 P를 만족하는 지를 증명하려고 한다고 하자. 즉,

$$\forall x \in S.P(x)$$

를 증명하려고 한다. $P(x)$는 x가 P라는 성질을 만족한다는 뜻이다. 그리고 그 집합 S가 귀납적인 규칙들 Φ에 의해서 정의되었다고 하자.

따라서, S의 원소들 사이에는 순서가 있고(0째번 원소, 1째번 원소, $\cdots$), 그 순서는 항상 0째번에 기초하고 있다. i째번 원소들이란, Φ 규칙들을 i번 적용해서 새롭게 만들어 지는 원소들이다.

집합 S는 모든 i째번 원소들을 모두 모은 집합이므로, S의 모든 원소에 대한 증명은 모든 i째 번에 만들어 지는 원소들에 대한 증명을 하면 된다. 0째번

에 만들어 지는 원소는 없다. 따라서 증명할 것도 없다.

- 시작은, 1째번에 만들어 지는 원소들에 대해서 P가 사실임을 증명해야 한다.
- 그리고, 임의의 i 이전 번에 만들어진 원소들에 대해서 P가 사실이라는 가정하에(*귀납가정induction hypothesis*라고 함), i째번에 만들어지는 원소들이 P를 만족하는 지를 증명하자.

그러면 집합 S 의 모든 원소들에 대해서 P를 만족하는 지를 증명한 것이 된다. 왜냐면, 집합 S는 모든 i번째 만들어지는 원소들만으로 구성되므로.

2.1.2.1 귀납규칙들에 대한 것으로

이 증명 기술을 귀납규칙들에 대한 것으로 다시 이야기 하면,

- 우선, 첫째번에 만들어 지는 원소들은 Φ에 있는 규칙들 중에는 공집합을 전제로 가지고있는 규칙들 $(\emptyset, x)$가 만드는 x들이다. 고로, 그러한 x들에 대해서 증명하는 것이 첫째번의 원소들에 대한 증명이 된다.
- 그 다음, 그 이상 째번에 만들어 지는 원소들은 Φ에 있는 규칙들 중에서 공집합이 아닌 전제($X \neq \emptyset$)를 가지고 있는 규칙들 (X, x)이 만드는 x들이다. x가 i째번에 만들어 지는 원소라면 X에 있는 원소들은 그 이전 번에 만들어 진 원소들이다. 따라서 X에 나타나는 원소들에 대해서 사실이라는 가정 하에 x에 대해서 사실인지를 증명한다.

이렇게 위의 두가지를 증명하면, 모든 i째 번에 만들어지는 원소들(Φ가 정의하는 모든 원소들)에 대해서 증명한 것이 된다.

2.1.2.2 일반적으로

두개로 쪼개지는 위와같은 증명을 하나로 이야기 하면, 귀납법 증명은 다음을 증명하면 된다: 모든 $i \in \mathcal{O}$에 대해서,

$$(\forall j < i. P(j\text{째번 원소})) \Rightarrow P(i\text{째번 원소}).$$

한꺼번에 두개가 포섭되는 이유를 따져 보면,

- i가 0일 때, "$j < i$"를 만족하는 j는 없다. 공집합의 모든 원소들에 대해서는 임의의 사실이 성립하므로, $\forall j < i.P(j$째번 원소$)$는 항상 성립. 따라서 위의 식은 true $\Rightarrow P(0$째번 원소$)$와 같다. 0 째번 원소도 없으므로 $P(0$째번 원소$)$는 항상 성립.

  i가 1일 때, "$j < i$"를 만족하는 j는 0이고 0째번 원소는 없으므로 위의 식은 true $\Rightarrow P(1$째번 원소$)$ 즉 $P(1$째번 원소$)$가 된다.

- i가 2 이상인 임의의 "째번"에 대해서는 특별한 것 없이, 위의 식 그대로를 증명하면, 귀납가정을 안고$(\forall j < i.P(j$째번 원소$))$ 증명하는 것이 된다.

Example 12 모든 자연수 n에 대해서, $0+1+2+\cdots+n = n(n+1)/2$인지를 증명하자.

모든 자연수는 귀납규칙 $n \rightarrow 0 \mid n+1$를 유한번 적용해서 만들어지는 원소들의 모임이다.

첫째번에 만들어 지는 원소는 0이다. n이 0일 때 위의 등호가 성립하는 지 증명한다. 그 다음은 i 이전 번에 만들어지는 자연수에 대해서 위의 등호가 성립한다고 가정하고, i째 번에 만들어 지는 자연수에 대해서 사실인지 증명한다. i째 번에 만들어지는 자연수를 $n+1$이라고 하자. 그 이전 번에 만들어진 자연수들 중에는 n이 있다. 그러면, $0+\cdots+n+(n+1) = n(n+1)/2+(n+1) = (n+1)(n+2)/2$ 이므로 위의 등식이 성립한다.

귀납규칙들을 따라 증명하면, 0일 때 사실인지 증명하고, n일 때 사실이라는 가정하에 $n+1$일 때 사실인지 증명하면 된다. □

Example 13 *두갈래가 꽉찬 나무complete binary tree*에 대해서, 말단 노드의 갯수는 내부 노드의 갯수와 같거나 하나 많다는 것을 증명하자. 두갈래가 꽉찬 나무 t를 만드는 귀납규칙은

$$t \rightarrow \circ \mid \mathtt{N}(t,t)$$

이다. 임의의 t에서 ∘(말단 노드)의 갯수 l이 N(내부 노드)의 갯수 n 더하기 1이다.

∘인 경우는, 그렇다. t_1과 t_2의 성질이 그렇다고 하자: $l_1 = n_1 + 1$ 이고 $l_2 = n_2 + 1$. 그러면, $\texttt{N}(t_1, t_2)$의 말단 노드의 수는 $l_1 + l_2$이고, 내부 노드의 수는 $n_1 + n_2 + 1$이다. $l_1 + l_2 = (n_1 + n_2 + 1) + 1$이 성립한다. □

2.2 형식 논리*formal logic*

형식 논리는 프로그래밍 언어이다. 그 언어로 짜여진 프로그램은 참이거나 거짓을 계산한다. 형식 논리에 대한 이야기는 프로그래밍 언어에 대해서 이야기할 때 만나게 될 여러 개념들을 익숙하게 해 준다.

선언논리*propositional logic*를 중심으로 알아보자.

2.2.1 모양과 뜻

선언논리에서 생각하는 논리식 f는 다음의 귀납법칙으로 만들어 지는 집합의 원소이다. 즉, 논리식 f는 다음의 방법으로 만들어 진다.

$$
\begin{array}{rcl}
f & \to & T \mid F \\
 & \mid & \neg f \\
 & \mid & f \land f \\
 & \mid & f \lor f \\
 & \mid & f \Rightarrow f
\end{array}
$$

위의 귀납법칙이 정의하는 집합의 원소들, 그것들이 선언논리라는 언어로 짜여진 프로그램들이다. 선언논리식은 위의 방법대로 만들어 지는 것만이 제대로 생긴 선언논리식인 것이다. 무한히 많은 선언논리식들은 모두 위의 일곱가지 방법을 반복 적용해서 만들어 진다.

그럼, 선언논리식의 의미는 무엇인가? 무한히 많은 선언논리식들 하나하나 마다 그 뜻이 무었인지 유한하게 정의할 방법은 있는가? 이 경우에도 귀납

법을 이용해서 의미하는 바를 정의할 수 있다. 임의의 논리식은 i째 번에 만들어 진다. i째 번에 만들어 지는 논리식은 그 이전 번에 만들어진 논리식을 가지고 만들어 진다. 따라서, i째 번 이전에 만들어진 논리식의 의미를 가지고, i째 번에 만들어 진 논리식의 의미를 정의하는 방법이면 되겠다.(다른 더 좋은 방법이 있을까?)

논리식 f의 의미를 $[\![f]\!]$라고 표기하자.

- 첫째 번에 만들어 지는 논리식의 의미를 정의하자. T는 참을 뜻하고 F는 거짓을 뜻한다.

$$\begin{array}{rcl} [\![T]\!] & = & \text{true} \\ [\![F]\!] & = & \text{false} \end{array}$$

- i째번 논리식 $\neg f$의 의미는 그 이전에 만들어진 f의 의미를 가지고 정의된다. 그 의미의 부정이 된다.

$$[\![\neg f]\!] = \text{not}[\![f]\!]$$

- 마찬가지로, 이전에 만들어진 f_1과 f_2의 의미를 가지고, 다른 식들의 의미가 결정된다. $f_1 \land f_2$의 의미는 그 둘의 논리곱이고, $f_1 \lor f_2$의 의미는 그 둘의 논리합이고, $f_1 \Rightarrow f_2$의 의미는 그 둘의 논리승이다.

$$\begin{array}{rcl} [\![f_1 \land f_2]\!] & = & [\![f_1]\!] \text{ andalso } [\![f_2]\!] \\ [\![f_1 \lor f_2]\!] & = & [\![f_1]\!] \text{ orelse } [\![f_2]\!] \\ [\![f_1 \Rightarrow f_2]\!] & = & [\![f_1]\!] \text{ implies } [\![f_2]\!] \end{array}$$

이렇게 어느 논리식의 의미가 그 논리식을 구성하는 하부 논리식의 의미로 정의되면 임의의 논리식의 의미가 정의되는 셈이다. 임의의 논리식은 기초가 되는(1째번) 논리식에서 부터 차곡 차곡 만들어진 것이기 때문이다.

Example 14 논리식 $(T \land (T \lor F)) \Rightarrow F$의 의미는, 정의에 따라 차곡 차곡 써

보면

$$\begin{aligned}
[\![(T \land (T \lor F)) \Rightarrow F]\!] &= [\![T \land (T \lor F)]\!] \text{ implies } [\![F]\!] \\
&= ([\![T]\!] \text{ andalso } [\![T \lor F]\!]) \text{ implies false} \\
&= (\text{true andalso } ([\![T]\!] \text{ orelse } [\![F]\!])) \text{ implies false} \\
&= (\text{true andalso (true orelse false)) implies false} \\
&= \text{false}
\end{aligned}$$

이다. □

2.2.2 추론규칙

선언 논리식의 의미가 정의되었다. 그 의미는 참이거나 거짓이다.

논리식이 참인지를 판별하는 방법은 무엇일까? 물론, 논리식의 의미를 정의한대로 따라가다 보면 결과를 알 수 있다: 참 혹은 거짓. 의미의 결과를 보면 논리식이 참인지 아닌지를 판별하면 된다. 이것은 프로그램을 돌려보고 그 프로그램이 참을 결과로 내는 지를 판별하는 것과 같다.

혹시, 프로그램을 돌리지 않고 알아내는 방법은 없을까? 논리식의 의미를 계산하지 않고, 참인지 거짓인지를 알 수 있는 방법은 없을까? 논리식의 모양만을 보면서 참인지를 판별할 수는 없을까? 이러한 판별규칙이 가능하지 않을까?

그런 규칙을 *증명규칙proof rule* 혹은 *추론규칙inference rule*이라고 한다.

이 규칙들도 어떤 집합을 만들어내는 귀납적인 규칙들로 정의된다. 그 규칙들이 만들어 내는 집합은 우리가 원하는 논리식들(참인 논리식들)로 구성되는 집합이다.

예를 들어 다음과 같은 *증명규칙proof rules*/*추론규칙inference rules*들을 생각하

자. 귀납 규칙들을 분수꼴로 표현한 것이다.

$$\frac{}{(\Gamma, T)} \qquad \frac{}{(\Gamma, f)}\ f \in \Gamma \qquad \frac{(\Gamma, F)}{(\Gamma, f)} \qquad \frac{(\Gamma, \neg\neg f)}{(\Gamma, f)}$$

$$\frac{(\Gamma, f_1) \quad (\Gamma, f_2)}{(\Gamma, f_1 \wedge f_2)} \qquad \frac{(\Gamma, f_1 \wedge f_2)}{(\Gamma, f_1)}$$

$$\frac{(\Gamma, f_1)}{(\Gamma, f_1 \vee f_2)} \qquad \frac{(\Gamma, f_1 \vee f_2) \quad (\Gamma \cup \{f_1\}, f_3) \quad (\Gamma \cup \{f_2\}, f_3)}{(\Gamma, f_3)}$$

$$\frac{(\Gamma \cup \{f_1\}, f_2)}{(\Gamma, f_1 \Rightarrow f_2)} \qquad \frac{(\Gamma, f_1 \Rightarrow f_2) \quad (\Gamma, f_1)}{(\Gamma, f_2)}$$

$$\frac{(\Gamma \cup \{f\}, F)}{(\Gamma, \neg f)} \qquad \frac{(\Gamma, f) \quad (\Gamma, \neg f)}{(\Gamma, F)}$$

(Γ, f)쌍들의 집합을 만드는 규칙들이다. 특히, "Γ에 있는 모든 논리식들이 참이면 f는 참"인 경우에 (Γ, f)를 만들어 내는 규칙들이다.

대개 형식논리에서는, "(Γ, f)"를 "$\Gamma \vdash f$"로 표기한다. 규칙마다 번호를 붙이고 다시 쓰면:

$$\frac{}{\Gamma \vdash T}\ (r1) \qquad \frac{}{\Gamma \vdash f}\ f \in \Gamma \quad (r2) \qquad \frac{\Gamma \vdash F}{\Gamma \vdash f}\ (r3) \qquad \frac{\Gamma \vdash \neg\neg f}{\Gamma \vdash f}\ (r4)$$

$$\frac{\Gamma \vdash f_1 \quad \Gamma \vdash f_2}{\Gamma \vdash f_1 \wedge f_2}\ (r5) \qquad \frac{\Gamma \vdash f_1 \wedge f_2}{\Gamma \vdash f_1}\ (r6)$$

$$\frac{\Gamma \vdash f_1}{\Gamma \vdash f_1 \vee f_2}\ (r7) \qquad \frac{\Gamma \vdash f_1 \vee f_2 \quad \Gamma \cup \{f_1\} \vdash f_3 \quad \Gamma \cup \{f_2\} \vdash f_3}{\Gamma \vdash f_3}\ (r8)$$

$$\frac{\Gamma \cup \{f_1\} \vdash f_2}{\Gamma \vdash f_1 \Rightarrow f_2}\ (r9) \qquad \frac{\Gamma \vdash f_1 \Rightarrow f_2 \quad \Gamma \vdash f_1}{\Gamma \vdash f_2}\ (r10)$$

$$\frac{\Gamma \cup \{f\} \vdash F}{\Gamma \vdash \neg f}\ (r11) \qquad \frac{\Gamma \vdash f \quad \Gamma \vdash \neg f}{\Gamma \vdash F}\ (r12)$$

이 증명규칙들을 $\Gamma \vdash f$들의 집합을 만드는 귀납규칙으로 볼 뿐 아니라, $\Gamma \vdash f$의 증명을 만드는 귀납규칙으로도 볼 수 있다. 그래서 "증명규칙"이라고 부른다.

예를 들어, 증명규칙

$$\frac{\Gamma \vdash f_1 \quad \Gamma \vdash f_2}{\Gamma \vdash f_1 \wedge f_2}$$

이 귀납적으로 이야기하는 것이 두가지다. $\Gamma \vdash f_1$와 $\Gamma \vdash f_2$가 집합에 있다면, $\Gamma \vdash f_1 \wedge f_2$도 집합의 원소여야 한다는 것이 하나. 그리고, $\Gamma \vdash f_1 \wedge f_2$의 증명을 만드는 (귀납적인) 방법으로, $\Gamma \vdash f_1$와 $\Gamma \vdash f_2$의 증명이 있다면 그 증명들을 분자에 매달고 분모에는 $\Gamma \vdash f_1 \wedge f_2$를 놓은 것이 $\Gamma \vdash f_1 \wedge f_2$의 증명을 만드는 방법이라는 것이다.

이렇게 만들어 지는 증명은 하나의 나무 구조가 된다. 이 나무구조의 뿌리 노드는 증명의 최종 결론이다. 나무의 갈래구조는 사용한 증명규칙들이 만들어 낸다. 규칙의 식들이 각각 노드가 되는 데, 분모식 노드에서 분자식 노드들로 가지치는 구조가 된다. 그 분자식들은 다시 다른 규칙의 분모가 되서 그 규칙의 분자식들로 다시 가지치고. 그 가지들의 최종 잎새는 증명규칙중에서 분자가 없는 규칙들의 분모식으로 끝맺게 된다.

예를 들어, $\{p \to \neg p\} \vdash \neg p$의 증명을 위의 증명규칙으로 만들면 아래의 나무 구조가 된다:

$$\dfrac{\dfrac{\dfrac{}{\{p \to \neg p, p\} \vdash p}\ (r2) \qquad \dfrac{\dfrac{}{\{p \to \neg p, p\} \vdash p \to \neg p}\ (r2) \quad \dfrac{}{\{p \to \neg p, p\} \vdash p}\ (r2)}{\{p \to \neg p, p\} \vdash \neg p}\ (r10)}{\{p \to \neg p, p\} \vdash F}\ (r12)}{\{p \to \neg p\} \vdash \neg p}\ (r11)$$

2.2.3 안전한 혹은 완전한

위의 규칙들이 만들어 내는 $\emptyset \vdash f$의 f는 모두 참인가? 반대로, 참인 식들은 모두 위의 증명규칙들을 통해서 $\emptyset \vdash f$ 꼴로 만들어 지는가?

첫 질문에 예 라고 답할 수 있으면 우리의 규칙은 *안전sound*하다고, 믿을 만하다고 한다.

둘째 질문에 예 라고 답할 수 있으면 우리의 규칙은 *완전complete*하다고, 빠뜨림이 없다고 한다.

증명규칙들이 안전하기도 하고 완전하기도 하다면, 참인 식들만 빠짐없이 만들어 내는 규칙이 된다.

안전하냐 완전하냐는 기계적인 증명규칙들의 성질에 대한 것이다. 그 기계적인 규칙들의 성질은 오직 우리가 생각하는 의미에 준해서 결정될 수 있다. 증명규칙이라는 기계가 만들어 내는 식들의 의미를 참조해야만 안전한지 완전한지를 결정할 수 있다. 신라면을 찍어내는 기계는 맹목적일 뿐이다. 그 기계가 좋은 이유는 항상 인기 있는 맛좋은 신라면을 만들어 낸다는 그 기계 바깥의 의미를 참조할 때에 비로소 결정된다.

프로그램에서도 그 겉모양(문법)과 속내용(의미)은 대게 다른 두개의 세계로 정의되고, 프로그램을 관찰하고 분석하는 모든 과정은 기계적으로 정의된다. 컴퓨터로 자동화하기 위해. 그 기계적인 과정들의 성질들(과연 맞는지, 무엇을 하는 것인지) 등에 대한 논의는 항상 그 과정의 의미나 결과물들의 의미를 참조하면서 확인된다.

3 장

모양과 뜻

프로그래밍 언어의 겉모양은 *문법구조syntax*를 말한다. 프로그래밍 언어의 속뜻은 *의미구조semantics*를 말한다.

그 두가지를 정의하지 못하면 그 언어를 정의한 것이 아니다. 그 두가지를 알지 못하면 그 언어를 사용할 수 없다. 제대로 생긴 프로그램을 어떻게 만들고 그 프로그램의 의미가 무엇인지에 대한 정의가 없이는 언어를 구현할 수 도 없고 사용할 수 도 없다. 더군다나 그 정의들은 애매하지 않고 혼동이 없어야 한다.

3.1 *문법구조syntax*

*문법구조syntax*는 프로그래밍 언어로 프로그램을 구성하는 방법이다. 제대로 생긴 프로그램들의 집합을 만드는 방법이다. 귀납적인 규칙으로 정의된다.

이 귀납적인 규칙들로 만들어지는 프로그램은 나무구조를 갖춘 이차원의 모습이다. 왜 나무구조인가?

Example 15 정수식을 만드는 귀납규칙을 생각해 보자.

$$\begin{array}{rcll} E & \to & n & (n \in \mathbb{Z}) \\ & | & E + E & \\ & | & - E & \end{array}$$

첫번째 규칙은 임의의 정수는 정수식이라고 한다. 하나의 정수만 말단으로 가지고 있는 나무가 되겠다.

두번째 규칙은 두개의 정수식을 가지고 +를 이용해서 정수식을 만들 수 있다고 한다. 이것이 나무 두개를 양쪽으로 매달고 있고 뿌리에는 "+"를 가지고 있는 나무가 되겠다.

마지막 규칙은 하나의 정수식을 가지고 −를 이용해서 정수식을 만드는 것이다. 이것도 나무 하나를 매달고 있고 뿌리에는 "-"를 가지고 있는 나무가 된다. □

Example 16 간단한 명령형 언어를 생각해 보자. 이 언어로 짤 수 있는 프로그램 C는 명령문(command)으로서 다음의 방법으로 만들어 진다:

$$
\begin{array}{rcl}
C & \rightarrow & \texttt{skip} \\
 & | & x \texttt{ := } E \\
 & | & \texttt{if } E \texttt{ then } C \texttt{ else } C \\
 & | & C \texttt{ ; } C
\end{array}
$$

E는 위의 예에서 정의한 정수식이라고 하자.

위의 방법들로 만들 수 있는 명령문들은 모두 나무 구조물 들이다.

"skip"만 있는 명령문은 그것만 말단으로 가지고 있는 나무가 되겠다.

두번째 규칙은 정수식 나무를 오른편에, 하나의 변수를 왼편에, 루트 노드는 ":="임을 표시하고 있는 나무가 된다.

세번째 규칙은 세개의 나무(정수식 나무와 두개의 명령어 나무)를 하부 나무로 가지고 있고 루트 노드에는 "if"문 임을 표시하고 있는 나무가 된다.

마지막 규칙은 두개의 명령어 나무를 하부 나무로 가지고 있고 루트 노드에는 순차적인 명령문(";")임을 표시하고 있는 나무가 된다. □

프로그램을 만들 때 필요한 최소한의 정보를 가지고 있는 가장 간단한 문법을 써 보자. Example 15와 Example 16에서 보여준 규칙들에는 규칙을 읽는 사람을 돕기 위해서 필요 이상의 정보가 있기도 하다.

다음이 필요한 정보만 가지고 있는 더욱 날씬한 규칙일 것이다:

$$
\begin{array}{rcl}
C & \rightarrow & \& \\
  & | & = x\ E \\
  & | & ?\ E\ C\ C \\
  & | & ;\ C\ C \\
\\
E & \rightarrow & n \qquad\qquad (n \in \mathbb{Z}) \\
  & | & +\ E\ E \\
  & | & -\ E
\end{array}
$$

긴 심볼을 쓸 필요도 없고, 심볼이 중간에 끼일 필요도 없고, 사이 사이에 “then”이나 “else”같은 장식이 있을 필요도 없다. 오직 각 규칙마다 다른 간단한 심볼을 쓰면 그만이다.

아니면 아예 그러한 심볼이 필요없어도 된다. 우리가 각 규칙마다 구분할 수 만 있다면:

$$
\begin{array}{rcl}
C & \rightarrow & \& \\
  & | & x\ E \\
  & | & E\ C\ C \\
  & | & C\ C \\
\\
E & \rightarrow & n \qquad\qquad (n \in \mathbb{Z}) \\
  & | & E\ E \\
  & | & -\ E
\end{array}
$$

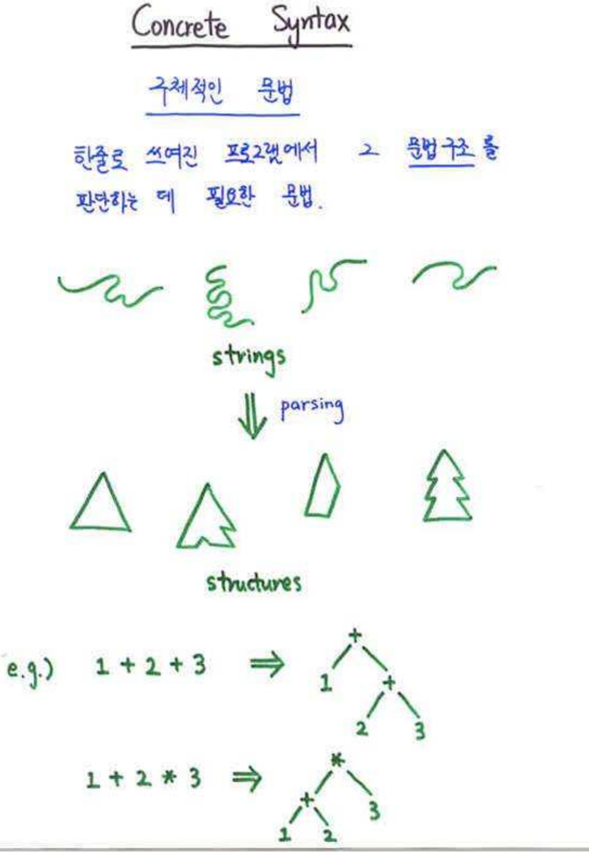
하지만, 이렇게 까지 프로그램 만드는 방법을 최대한 요약해서 보여줄 필요는 없다. 문법 규칙들을 보고 무엇을 만드는 방법인지를 힌트해 주는 규칙들을 사용하게된다.

3.1.1 요약된 혹은 구체적인

지금까지 보아온 문법규칙들을 *"요약된 문법구조"abstract syntax*라고도 하는데, *요약된 문법구조abstract syntax*와 *구체적인 문법구조concrete syntax*의 차이는 프로그램을 만들 때 사용하는 규칙이냐, 읽을 때 사용하는 규칙이냐, 에 달려있다.

프로그램을 만들 때 사용하는 문법은 지금까지 보아온 귀납적인 방법이면 충분하다. 만든 결과는 나무구조를 가진 이차원의 구조물이다.

반면에 프로그램을 읽을 때의 문법은 더 복잡해 진다. 왜냐하면, 대게 만들어진 프로그램을 표현할 때는 나무를 그리지 않고 일차원적인 글자들의 나열이 되고 이러한 글자나 소리의 일차원적인 실을 받아서, 쓰거나 말한 사람의 머리에 그려져 있었을 2차원의 나무구조를 복원시키는 규칙들이 되기 때문이다.



*구체적인 문법concrete syntax*은 얼마나 구체적일까? 나무구조가 아니라 일

차원의 실로 표현되는 다음의 정수식 프로그램을 생각해 보자:

```
-1+2
```

위의 것은 다음 두개의 나무구조중 하나를 일차원으로 펼친것이다:

$$\langle\langle -1\rangle + 2\rangle \qquad -\langle 1 + 2\rangle$$

어느것인가? 프로그램 실 “-1+2”를 둘 중의 어느 프로그램으로 복구시킬 것인가? 요약된 문법

$$\begin{array}{rcll} E & \to & n & (n \in \mathbb{Z}) \\ & | & E \texttt{+} E & \\ & | & \texttt{-} E & \end{array}$$

으로는 둘 중에 어느 구조로 복구시켜야 할 지 알 수 없다. 두가지 구조를 모두 구축할 수 있다.

정수식의 문법이 다음과 같다면 어떨까?

$$\begin{array}{rcll} E & \to & n & (n \in \mathbb{Z}) \\ & | & E \texttt{+} E & \\ & | & \texttt{-} F & \\ F & \to & n & \\ & | & \texttt{(}\ E\ \texttt{)} & \end{array}$$

위의 규칙이 말하는 것은, 음의 부호가 앞에 붙은 정수식인 경우는, 두가지 밖에 없다: 말단 자연수이거나, 괄호가 붙은 정수식이거나. 따라서, 프로그램 실 “-1+2”의 경우는 $\langle\langle -1\rangle + 2\rangle$의 구조밖에는 없다.

*구체적인 문법구조concrete syntax*는 혼동없이 2차원의 프로그램 구조물을 복구하는 데 사용되는데, 이 문법은 *프로그램 복원parsing* 혹은 프로그램의 *문법검증parsing*이라는 과정의 설계도가 된다. 그 과정은 우리가 프로그램을 쓰면(문자들의 1차원 실로) 컴퓨터가 그 구조물을 자동으로 복원하는 과정이다.

우리는 구체적인 문법을 더 이상 다루지않는다. "프로그램"이라고 하면, 요약된 문법 규칙들로 만들어진 이차원의 나무구조를 뜻한다. 비록 이곳에 적는 프로그램들이 그 나무구조를 직접 그리지는 않았더라도, 그 2차원의 구조물이 무엇인지 혼동되지 않도록 적절히 괄호를 쓰거나 할 것이다.

3.2 *의미구조semantics*

*의미구조semantics*는 프로그램이 뜻하는 바를 정의한다.

프로그램이 뜻 하는 바는 무엇인가? "1+2"라는 프로그램은 무엇을 뜻하는가? 결과값 3을 뜻하는가, 1과 2을 더해서 결과를 계산한다, 는 과정을 뜻하는가? 3을 뜻한다고 정의하는 스타일, 뜻하는 바 궁극을 수학적인 구조물의 원소로 정의해 가는 스타일을 *"궁극을 드러내는 의미구조"denotational semantics*라고 한다. 반면에 프로그램의 계산 과정을 정의함으로써 프로그램의 의미를 정의해 가는 스타일을 *"과정을 드러내는 의미구조"operational semantics*라고 한다.

다양한 스타일의 의미구조 정의법은 모두 충분히 엄밀하다. 그리고 각 스타일마다 프로그램 의미의 성질을 증명하는 다양한 기술이 개발되어 있다.

그럼 왜 여러가지 스타일을 살펴볼 필요가 있을까? 그것은 다양한 의미구조 정의 방식과 증명기술들이 모두 종합적으로 사용되는 것이 흔하기 때문이다. 어떤때는 궁극의 스타일이 적절하고 어떤때는 과정을 드러내는 스타일이 적절하다. 각 경우마다 우리가 드러내고 싶은 디테일의 수준 만큼만 표현해 주는 의미구조 방식을 취사선택하게 된다.

3.2.1 과정을 드러내는

*과정을 드러내는 의미구조operational semantics*는 프로그램이 실행되는 과정을 정의한다.

프로그램 실행의 과정을 어느 수준 표현해야 할까? 어느 정도로 자세한 수준이 필요할까? 계산과정에서 삼성의 칩들 사이에 일어나는 전기신호들의 흐름을 일일이 표현해야 할까? 아니면 1이 1을 계산하고 2가 2를 계산하고 +가

$1+2=3$이라는 등식을 적용하는 것으로 표현해야 할까? 아니면 실행되는 과정을, 프로그램과 그 계산결과를 결론으로 하는 논리적인 증명의 과정이라고 표현해야 할까?

의미구조를 정의하는 목표에 맞추어 그 디테일의 정도를 정하게된다. 대게는 상위의 수준, 가상의 기계에서 실행되는 모습이거나, 기계적인 논리 시스템의 증명으로 정의하게 된다.

과정을 드러내는 의미구조도 *궁극을 드러내는 스타일denotational semantics*처럼 충분히 엄밀하다. 다른 것이 있다면,

- 조립식(compositional)이 아닐 수 있다: 프로그램의 의미가 그 프로그램의 부품들의 의미들로만 구성되는 것은 아니다. 그렇게 되면 좋지만, 아니어도 상관없다.

- 하지만 귀납적(inductive)이다: 어느 프로그램의 의미를 구성하는 부품들은 귀납적으로 정의된다. 이 귀납이 프로그램의 구조만을 따라 되돌기도 하지만(이렇게 되면 조립식이 된다), 프로그램 이외의 것(의미를 드러내는 데 사용되는 장치들)을 따라 되돌기도 한다.

3.2.1.1 큰 보폭으로

프로그램의 실행이 진행되는 과정을 *큰 보폭으로 그려보자big-step semantics*. 논리 시스템의 증명 규칙들로 표현된다. 증명하고자 하는 바 는, 명령문 C와 정수식 E에 대해서 각각

$$M \vdash C \Rightarrow M' \quad \text{와} \quad M \vdash E \Rightarrow v$$

이다.

$M \vdash C \Rightarrow M'$는 "명령문 C가 메모리 M의 상태에서 실행이 되고 결과의 메모리는 M'이다"로 읽으면 된다. $M \vdash E \Rightarrow v$는 "정수식 e는 메모리 M의 상태에서 정수값 v를 계산한다"로 읽고.

모든 규칙들을 쓰면 이렇게 된다:

$$\frac{}{M \vdash \texttt{skip} \Rightarrow M}$$

$$\frac{M \vdash E \Rightarrow v}{M \vdash x \texttt{ := } E \Rightarrow M\{x \mapsto v\}}$$

$$\frac{M \vdash C_1 \Rightarrow M_1 \quad M_1 \vdash C_2 \Rightarrow M_2}{M \vdash C_1 \texttt{ ; } C_2 \Rightarrow M_2}$$

$$\frac{M \vdash E \Rightarrow 0 \quad M \vdash C_2 \Rightarrow M'}{M \vdash \texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2 \Rightarrow M'}$$

$$\frac{M \vdash E \Rightarrow v \quad M \vdash C_1 \Rightarrow M'}{M \vdash \texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2 \Rightarrow M'} \; v \neq 0$$

$$\frac{M \vdash E \Rightarrow 0}{M \vdash \texttt{while } E \texttt{ do } C \Rightarrow M}$$

$$\frac{M \vdash E \Rightarrow v \quad M \vdash C \Rightarrow M_1 \quad M_1 \vdash \texttt{while } E \texttt{ do } C \Rightarrow M_2}{M \vdash \texttt{while } E \texttt{ do } C \Rightarrow M_2} \; v \neq 0$$

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

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

$$\frac{M \vdash E_1 \Rightarrow v_1 \quad M \vdash E_2 \Rightarrow v_2}{M \vdash E_1 \texttt{ + } E_2 \Rightarrow v_1 + v_2}$$

$$\frac{M \vdash E \Rightarrow v}{M \vdash \texttt{-} E \Rightarrow -v}$$

위의 규칙들은 논리시스템의 증명규칙이라고 이해해도 된다. 명령 프로그램 C의 의미는 임의의 메모리 M과 M'에 대해서 $M \vdash C \Rightarrow M'$를 증명할 수 있으면 그 의미가 되겠다. 대게는 M이 비어있는 메모리일때 C를 실행시키는 과정을 나타내고 싶으므로, $\emptyset \vdash C \Rightarrow M'$의 증명이 가능하면 C의 의미가 된다. 그것이 증명 불가능하면 C의 의미는 없는 것이다.

Example 17 x `:=` `1` `;` y `:=` x `+` `1` 의 의미는 $\emptyset$메모리에 대해서 다음과 같은 증

명으로 표현된다. 위의 명령문을 C라고 하자.

$$\dfrac{\dfrac{\emptyset \vdash 1 \Rightarrow 1}{\emptyset \vdash x \texttt{ := } 1 \Rightarrow \{x \mapsto 1\}} \qquad \dfrac{\dfrac{\{x \mapsto 1\} \vdash x \Rightarrow 1 \qquad \{x \mapsto 1\} \vdash 1 \Rightarrow 1}{\{x \mapsto 1\} \vdash x \texttt{ + } 1 \Rightarrow 2}}{\{x \mapsto 1\} \vdash y \texttt{ := } x \texttt{ + } 1 \Rightarrow \{x \mapsto 1, y \mapsto 2\}}}{\emptyset \vdash C \Rightarrow \{x \mapsto 1, y \mapsto 2\}}$$

□

이렇게 의미구조를 정의하는 방법을 *"자연스런"natural semantics*, *"구조적인"structural operational semantics*, 혹은 *"관계형"relational semantics* 의미구조라고도 불린다.

- "자연스럽"다고 하는 이유는 아마도 "추론 규칙"꼴로 구성되어 있고, *"자연스런 추론규칙"natural deduction rules* 이라고 불리는 추론 규칙이 있어서 그런 이름을 붙였을 것이다.

- "구조적"이라고 하는 이유는 두가지 정도다.

  지금 돌아보면 당연한 모습처럼 보이지만, 이러한 스타일로 계산과정을 드러내는 의미구조 방식은 당시의 흔한 방식보다는 더욱 짜임새가 있었기 때문이다.

  당시의 흔한 방식은 가상의 기계를 정의하고 프로그램이 그 기계에서 어떻게 실행되는 지를 정의한다. 이러다 보니, 기계의 실행과정중에 프로그램이 부자연스럽게 조각나면서 기계 상태를 표현하는 데 동원되기도 하고, 프로그램의 의미를 과도하게 낮은 수준의 실행과정으로 세세하게 표현하게 된다. (3.2.1.4절에서 자세히 다룸).

  이러한 옛 방식을 "구현을 통해서 정의하기"(definition by implementation), " 어떻게 구현하는 지 보임으로서 무엇인지 정의하기"라고 하는데, 이것보다는 확연히 "구조적"이지 않은가. 프로그램의 구조마다 추론규칙이 정의되고, 계산과정은 그 추론규칙들이 레고블락이 되어 만들어내는 하나의 증명이 되는 것이다.

"구조적"이라고 부르는 다른 이유는 의미규칙들이 한 집합(프로그램과 의미장치들간의 관계쌍들의 집합)을 귀납적으로 정의하는 방식이고, 그 귀납이 프로그램이나 의미장치들의 구조를 따라 흐르기 때문이다.

- "관계형"이라고도 하는 이유는 위의 추론규칙들이 관계된 짝들의 집합을 정의하는 귀납적인 방법으로도 바라볼 수 있기 때문이다.

  관계된 짝들이란 $M \vdash C \Rightarrow M'$ 인 $\langle M, C, M' \rangle$와 $M \vdash E \Rightarrow v$인 $\langle M, E, v \rangle$들이다. 위의 규칙들에 의해서 그러한 관계를 맺을 수 있는 M, C, M'과 M, E, v들이 모아진다.

3.2.1.2 작은 보폭으로

프로그램의 실행을 *작은 보폭으로 정의해 보자small-step semantics.*

$$\frac{}{(M, \texttt{skip}) \to (M, \text{done})}$$

$$\frac{(M, E) \to (M, E')}{(M, x \texttt{ := } E) \to (M, x \texttt{ := } E')}$$

$$\frac{}{(M, x \texttt{ := } v) \to (M\{x \mapsto v\}, \text{done})}$$

$$\frac{(M, E) \to (M, E')}{(M, \texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2) \to (M, \texttt{if } E' \texttt{ then } C_1 \texttt{ else } C_2)}$$

$$\frac{}{(M, \texttt{if } 0 \texttt{ then } C_1 \texttt{ else } C_2) \to (M, C_1)}$$

$$\frac{}{(M, \texttt{if } n \texttt{ then } C_1 \texttt{ else } C_2) \to (M, C_1)} \; n \neq 0$$

$$\frac{(M, C_1) \to (M', C_1')}{(M, C_1 \texttt{ ; } C_2) \to (M', C_1' \texttt{ ; } C_2)}$$

$$\frac{}{(M, \texttt{while } E \texttt{ do } C) \to (M, \texttt{if } E \texttt{ then } C \texttt{ ; while } E \texttt{ do } C \texttt{ else skip})}$$

$$\frac{}{(M, \text{done} \texttt{ ; } C_2) \to (M, C_2)}$$

$$\frac{(M, E_1) \to (M, E_1')}{(M, E_1 + E_2) \to (M, E_1' + E_2)}$$

$$\frac{(M, E_2) \to (M, E_2')}{(M, v_1 + E_2) \to (M, v_1 + E_2')}$$

$$\frac{}{(M, v_1 + v_2) \to (M, v_1 + v_2)}$$

이러한 스타일을 *변이과정 의미구조transition semantics*라고도 한다.

이 방식에서 재미있는 것은 문법적인(겉모양을 구성하는) 물건들과 의미적인(속뜻을 표현하는) 물건들이 한 데 섞이고 있다는 것이다.

이것이 문제될 것은 없다. 것 모양을 구성하는 것들과 속 뜻을 구성하는 것들이 반드시 다른 세계에서 멀찌감치 떨어져 있어야 한다는 것은, Tarski라는 수학자가 시작한 전통일 뿐이다.

프로그램 이외의 의미장치 없이 프로그램만을 가지고도 *변이과정 의미구조transition semantics*를 정의할 수도 있다. 예를들어, 메모리를 뜻하는 M이라는 프로그램 이외의 의미장치 없이.

예를 들어, 간단한 정수식 프로그램들의 의미는 프로그램을 다시쓰는 과정으로 정의된다: $1 + 2 + 3$ 다시쓰면 $3 + 3$ 다시쓰면 6. 프로그램 이외에 다른 장치가 사용되지 않는다.

3.2.1.3 문맥구조를 통해서

프로그램 실행을 프로그램을 다시 쓰는 과정으로 정의해 보자. 예를 들어, $1 + 2 + 3$ 다시쓰면 $3 + 3$ 다시쓰면 6. 이렇게 프로그램을 다시 써 가는 *과정이 프로그램의 실행이라고 볼 수 있다transition semantics*.

두가지를 정의해야 한다: 프로그램의 어느 부분을 다시 써야(계산해야) 하는가? 다시 써야할 부분을 무엇으로 다시 써야 하는가?

프로그램의 어느 부분을 다시 써야(계산해야) 하는가? 이 질문에 대한 답은 *"실행문맥"evaluation context*에 의해서 정의된다. 실행문맥의 정의에 따라서 현재의 프로그램을 구성하다보면, 다시 써야 할 부분(계산해야 할 속 부분)이 결정된다. 재미있는 것은 그 정의가 문법적으로 가능하다는 것이다.

다음의 정의를 보자. 실행문맥을 가지고 있는 프로그램 K를 정의한다. K안에는 $[]$가 딱 하나 있다. 그 곳이 프로그램에서 다시 써야 할 부분, 먼저 실행되야 할 부분이 된다. 그러한 부분을 품은 프로그램을 강조하기 위해 "$K[]$"라고 쓰고, 그 빈칸에 들어있는 (다시 써야할) 프로그램 부분 C 까지 드러내어 "$K[C]$"라고 표현한다.

지금까지 예로 사용되어 온 언어에서, 다시 써야 할 부분 $[]$을 내포한 실행문맥이 문법적으로 다음과 같이 정의된다:

$$
\begin{array}{rcl}
K & \to & [] \\
 & | & x \texttt{ := } K \\
 & | & K \texttt{ ; } C \\
 & | & \text{done} \texttt{ ; } K \\
 & | & \texttt{if } K \texttt{ then } C \texttt{ else } C \\
 & | & \texttt{while } K \texttt{ do } C \\
 & | & K \texttt{ + } E \\
 & | & v \texttt{ + } K \\
 & | & \texttt{- } K
\end{array}
$$

즉, 지정문(assignment command)에서는 다음에 실행되야 할 부분 K는 오른쪽식에 있고

$$x \texttt{ := } K$$

순서문에서(sequential command) 뒤의 명령문이 다음에 실행되야 할 부분 이라면, 앞의 명령문은 시행이 이미 끝났을 때 만이고

$$\text{done} \texttt{ ; } K$$

덧셈식에서 오른쪽 식이 다음에 실행되야 할 부분이라면, 왼쪽 식의 결과는 나와있어야 한다

$$v \texttt{ + } K.$$

다시 써야할 부분을 무엇으로 다시 써야 하는가? 이제 이 질문에 대한 답은 *다시쓰기 규칙rewriting rule*에 의해서 정의된다. 우선, 다시 쓸 곳은 다시 쓰면 되고

$$\frac{(M, C) \to (M', C')}{(M, K[C]) \to (M', K[C'])}$$

$$\frac{(M, E) \to (M, E')}{(M, K[E]) \to (M, K[E'])}$$

속에서 어떻게 다시 쓰여지는 가 하면:

$$\begin{array}{rcll}
(M, x \texttt{ := } v) & \to & (M\{x \mapsto v\}, \text{done}) & \\
(M, \text{done} \;;\; \text{done}) & \to & (M, \text{done}) & \\
(M, \texttt{if } 0 \texttt{ then } C_1 \texttt{ else } C_2) & \to & (M, C_1) & \\
(M, \texttt{if } v \texttt{ then } C_1 \texttt{ else } C_2) & \to & (M, C_2) & (v \neq 0) \\
(M, \texttt{while } 0_E \texttt{ do } C) & \to & (M, \text{done}) & \\
(M, \texttt{while } v_E \texttt{ do } C) & \to & (M, C \;;\; \texttt{while } E \texttt{ do } C) & (v \neq 0) \\
(M, v_1 \texttt{ + } v_2) & \to & (M, v) & (v = v_1 + v_2) \\
(M, \texttt{-}\, v) & \to & (M, -v) & \\
(M, x) & \to & (M, M(x)) &
\end{array}$$

이다.

Example 18 x `:=` `1` ; y `:=` x `+` `1` 의 의미를 문맥구조를 통해서 알아보면,

$$\frac{(\emptyset, x \texttt{ := } \texttt{1}) \to (\{x \mapsto \texttt{1}\}, \text{done})}{(\emptyset, [x \texttt{ := } \texttt{1}] \;;\; y \texttt{ := } x \texttt{ + } \texttt{1}) \to (\{x \mapsto \texttt{1}\}, \text{done} \;;\; y \texttt{ := } x \texttt{ + } \texttt{1})}$$

다음은,

$$\frac{(\{x \mapsto \texttt{1}\}, x) \to (\{x \mapsto \texttt{1}\}, \texttt{1})}{(\{x \mapsto \texttt{1}\}, \text{done} \;;\; y \texttt{ := } [x] \texttt{ + } \texttt{1}) \to (\{x \mapsto \texttt{1}\}, \text{done} \;;\; y \texttt{ := } \texttt{1} \texttt{ + } \texttt{1})}$$

다음은,

$$\frac{(\{x \mapsto 1\}, 1 + 1) \to (\{x \mapsto 1\}, 2)}{(\{x \mapsto 1\}, \text{done} \ ;\ y := [1 + 1]) \to (\{x \mapsto 1\}, \text{done} \ ;\ y := 2)}$$

다음은,

$$\frac{(\{x \mapsto 1\}, y := 2) \to (\{x \mapsto 1, y \mapsto 2\}, \text{done})}{(\{x \mapsto 1\}, \text{done} \ ;\ [y := 2]) \to (\{x \mapsto 1, y \mapsto 2\}, \text{done} \ ;\ \text{done})}$$

다음은

$$(\{x \mapsto 1, y \mapsto 2\}, \text{done} \ ;\ \text{done}) \to (\{x \mapsto 1, y \mapsto 2\}, \text{done}).$$

□

3.2.1.4 가상의 기계를 통해서

어떤 *가상의 기계virtual machine*가 정의되어 있고 프로그램의 의미는 그 프로그램이 그 기계에서 실행되는 과정으로 정의된다. 기계에서 실행되는 과정은 기계상태가 매 스텝마다 변화되는 과정이 된다.

예를 들면, 정수식의 의미가 한 기계의 실행과정으로 다음과 같이 정의된다. 변수가 없는 정수식만을 생각해 보자:

$$E \ \to \ n \mid E + E \mid - E$$

정의하는 기계는 소위 말하는 “스택머신”이다. 그 기계는 스택 S와 명령어들 C로 구성되어 있다:

$$\langle S, C \rangle$$

스택은 정수들로 차곡차곡 쌓여있다:

$$\begin{array}{rcll} S & \rightarrow & \epsilon & (\text{빈 스택}) \\ & | & n.S & (n \in \mathbb{Z}) \end{array}$$

명령어들은 정수식이나 그 조각들이 쌓여있다:

$$\begin{array}{rcll} C & \rightarrow & \epsilon & (\text{빈 명령어}) \\ & | & E.C & \\ & | & \texttt{+}.C & \\ & | & \texttt{-}.C & \end{array}$$

기계 작동과정의 한 스텝은 다음과 같이 정의된다:

$$\begin{array}{rcll} \langle S, n.C \rangle & \rightarrow & \langle n.S, C \rangle & \\ \langle S, E_1 \texttt{ + } E_2.C \rangle & \rightarrow & \langle S, E_1.E_2.\texttt{+}.C \rangle & \\ \langle n_2.n_1.S, \texttt{+}.C \rangle & \rightarrow & \langle n.S, C \rangle & (n = n_1 + n_2) \\ \langle n.S, \texttt{-}.C \rangle & \rightarrow & \langle -n.S, C \rangle & \end{array}$$

정수식 E의 의미는 $\langle \epsilon, E \rangle \rightarrow \cdots$ 가 된다.

또다른 스타일로는, 명령어들을 정수식과는 다른, 기계 고유의 명령어들로 정의할 수 도 있다:

$$\begin{array}{rcll} C & \rightarrow & \epsilon & (\text{빈 명령어}) \\ & | & \texttt{push}\, n.C & (n \in \mathbb{Z}) \\ & | & \texttt{pop}.C & \\ & | & \texttt{add}.C & \\ & | & \texttt{rev}.C & \end{array}$$

그리고, 기계 작동과정의 한 스텝도 다음과 같이 정의된다:

$$\begin{array}{rcll} \langle S, \texttt{push}\, n.C\rangle & \rightarrow & \langle n.S, C\rangle & \\ \langle n.S, \texttt{pop}.C\rangle & \rightarrow & \langle S, C\rangle & \\ \langle n_1.n_2.S, \texttt{add}.C\rangle & \rightarrow & \langle n.S, C\rangle & (n = n_1 + n_2) \\ \langle n.S, \texttt{rev}.C\rangle & \rightarrow & \langle -n.S, C\rangle & \end{array}$$

C에 있는 첫 명령어를 수행하면서 기계상태가 변하는 과정이다.

그리고, 이제 정수식들이 스택머신의 어떤 명령어들로 변환되는 지를 정의하면 된다:

$$\begin{array}{rcl} [\![n]\!] & = & \texttt{push}\, n \\ [\![E_1 \texttt{+} E_2]\!] & = & [\![E_1]\!].[\![E_2]\!].\texttt{add} \\ [\![\texttt{-}\, E]\!] & = & [\![E]\!].\texttt{rev} \end{array}$$

어떻게 정의하던간에, 이러한 기계의 과정이 매우 임의적이라고 생각될 수 있다. 가상의 기계를 어떻게 디자인 하는가? 그 기계의 디테일은 어느 레벨에서 정해야 하는가? 그에 대한 답은 프로그램의 의미를 정의하는 목적, 그에 따라 결정될 것이다.

대게 가상기계를 사용해서 프로그램의 의미를 정의하는 것은 프로그래밍 언어를 구현하는 단계에서 사용된다: 프로그래밍 언어의 *번역기**compiler*나 *실행기**interpreter*를 구현할 때. 프로그래밍 언어의 성질을 궁리하거나 그 언어로 짜여진 프로그램들의 관심있는 성질을 분석할 때에 사용하는 의미구조로 사용되는 예는 드믈다. 가상의 기계는 대게가 이와같은 분석에서는 필요하지 않은 디테일(기계의 볼트 넛트들)을 드러내기 때문이다.

3.2.2 궁극을 드러내는

*궁극을 드러내는 의미구조denotational semantics*는 프로그램의 의미를 전통적인 수학의 세계에서 정의한다. 프로그램이 뜻 하는 바를 수학의 물건으로 정의하는 것이다.

영어 “denotational”을 “궁극을 드러내는”이라고 번역한 이유는? “지칭하

는 바를 드러내는 의미구조"라고 직역할 수 있지만, 그러면 그 고유 특징을 드러내지 못하기 때문이다. "denotational semantics"는 프로그램의 의미를 결정하는 한 방법을 뜻하는 고유명사이다. 그 방법의 특징은, 수학의 세계에서 의미하는 바 그 궁극을 드러내는 것이다.

특히, 프로그램 부품들의 의미들이 전체 프로그램의 의미를 구성한다. 이러한 이유로 *조립식 의미구조compositional semantics*라고 불리기도 한다. 조립식 의미구조가 좋은 이유는 명백하다. 프로그램의 의미가 쉽게 결정된다. 프로그램 생김새만 잘 뜯어서 그 부품들을 파악하면, 그 부품들의 의미를 가지고 전체 프로그램의 의미가 결정된다.

예를 들어보자. 아래와 같은 *명령형 언어imperative language*를 생각해보자. 우리가 늘 알고 있는 명령형 언어라고 보면 된다.

$$
\begin{array}{rcll}
C & \to & \texttt{skip} & \\
 & | & x \texttt{ := } E & \\
 & | & \texttt{if } E \texttt{ then } C \texttt{ else } C & \\
 & | & C \texttt{ ; } C & \\
E & \to & n & (n \in \mathbb{Z}) \\
 & | & x & \\
 & | & E \texttt{ + } E & \\
 & | & \texttt{-}\, E &
\end{array}
$$

프로그램은 명령문이 되고, 명령문은 기계의 메모리에 정수값들을 저장시키면서 일을 진행한다. 프로그램의 변수는 메모리의 주소를 뜻하기도 하고 (지정문 "x := E"의 왼편에 사용되는 경우) 그 주소에 저장된 정수값을 뜻하기도 한다(계산식 E에서 사용되는 경우).

이렇게 간단한 의미구조를 수학적으로 모델하기는 어렵지 않다. 우선, 명령어 C의 의미는 메모리 상태를 변화시키는 함수로 정의한다: 메모리를 받아서 메모리를 내어놓는. 메모리도 또 다른 함수로 정의한다: 프로그램 변수를 받아서 그 변수가 가지는 정수값을 내어놓는. 프로그램 변수는 프로그램에서

사용하는 변수 이름이다.

그리고, 이런 물건들이 어느 집합에 소속되는 지를 정의하자. 그러한 집합을 *의미공간semantic domain*이라고 한다. 아래 $Memory$, $Value$, Var, $Memory \to Memory$, $Memory \to Value$등이 의미공간이 되겠다:

$$\begin{array}{rcl}
M \in Memory & = & Var \to Value \\
z \in Value & = & \mathbb{Z} \\
x \in Var & = & ProgramVariable \\
\text{명령문 } C\text{의 의미 } [\![C]\!] & \in & Memory \to Memory \\
\text{계산식 } E\text{의 의미 } [\![E]\!] & \in & Memory \to \mathbb{Z}
\end{array}$$

이제, 프로그램의 의미는 프로그램의 각각의 경우에 대해서 위에 마련한 의미공간의 원소들을 가지고 다음과 같이 정의된다:

$$\begin{array}{rcl}
[\![\texttt{skip}]\!]\, M & = & M \\
[\![x \texttt{ := } E]\!]\, M & = & M\{x \mapsto [\![E]\!]\, M\} \\
[\![\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2]\!]\, M & = & \mathit{if}\ [\![E]\!]M \neq 0\ \mathit{then}\ [\![C]\!]M\ \mathit{else}\ [\![C_2]\!]M \\
[\![C_1 \texttt{ ; } C_2]\!]\, M & = & [\![C_2]\!]\,([\![C_1]\!]\, M) \\
[\![n]\!]\, M & = & n \\
[\![E_1 \texttt{ + } E_2]\!]\, M & = & ([\![E_1]\!]\, M) + ([\![E_2]\!]\, M) \\
[\![\texttt{-}\, E]\!]\, M & = & -([\![E]\!]\, M)
\end{array}$$

조립식인 것이 보이는가? 보기 좋다. 모든 프로그램의 의미는 그 부품들의 의미만을 가지고 구성되어진다. 예로, if-문의 의미는 그 부품들의 의미들로 정의되어 있지 않은가:

$$[\![\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2]\!] = \cdots [\![E]\!] \cdots [\![C_1]\!] \cdots [\![C_2]\!].$$

3.2.2.1 조립식일 수 있는 이유: *의미공간 이론domain theory*

그런데, 이렇게 부품들의 의미로 전체의 의미를 구성하는 상식적인 방식이 일

사천리 가능하지는 않다. 명령형 언어에서 반복문이 가능하다고 하면, 반복문의 의미를 반복문을 구성하는 부품들의 의미만으로 정의하는 것이 간단치 않게된다.

조립식으로 의미를 결정하기가 난처한 경우를 살펴보자. 명령문중에 반복문으로 while문을 쓸 수 있다고 하자.

$$C \;\rightarrow\; \cdots \mid \texttt{while}\ E\ \texttt{do}\ C$$

while문은 조건식 E의 값이 0이면 아무것도 안하고 끝나지만, 조건식의 값이 0이 아닐 때는 명령문 C를 실행하고 난 후 다시 같은 while문을 반복하게 된다.

따라서 아래와 같이 while문의 의미를 정의해 보려고 할 수 있겠다:

$$[\![\texttt{while}\ E\ \texttt{do}\ C]\!]M = \mathit{if}\ [\![E]\!]M \neq 0\ \mathit{then}\ [\![\texttt{while}\ E\ \texttt{do}\ C]\!]([\![C]\!]M)\ \mathit{else}\ M$$

조립식인가? 그렇지 않다. $[\![\texttt{while}\ E\ \texttt{do}\ C]\!]$가 $[\![E]\!]$와 $[\![C]\!]$만 가지고 정의되지 않고 자기자신 $[\![\texttt{while}\ E\ \texttt{do}\ C]\!]$도 사용되고 있다.

따라서 위의것은 $[\![\texttt{while}\ E\ \texttt{do}\ C]\!]$의 정의가 아니다. 이것은 단순히 $[\![\texttt{while}\ E\ \texttt{do}\ C]\!]$에 대한 방정식일 뿐이다. 그 방정식의 해가 바로 $[\![\texttt{while}\ E\ \texttt{do}\ C]\!]$의 정의일 것이다.

그렇다면 그 해는 무엇일까? 해는 과연 있을까? 항상 있을까? 있다면 유일하게 있을까? 이에 대한 모든 답은 모두 예, 라고 할 수 있다. 이유는 의미방정식에서 사용하는 모든 물건들(원소들과 연산자들)이 소속된 *의미공간semantic domain*이 좀 특별하기 때문이다.

그러한 의미공간에 대한 이론이 *의미공간 이론domain theory*인데, 이 이론은 다음을 확인해 준다:

> 모든 컴퓨터 프로그램(모든 계산 가능한 함수들)의 의미(의미방정식의 해)는 의미공간 이론에서 규정하는 성질의 집합안에서 유일하게 존재하고 그것은 이러이러하다.

의미공간 이론이 규명해 낸 그러한 집합은 *CPOcomplete partial order set*라는

성질을 만족하는 집합이다. 그래서, 모든 프로그램의 의미방정식들에 쓰이는 것들은 모두 어떤 CPO의 원소들이고, 특히 방정식에서 사용하는 연산자들은 모두 CPO에서 CPO로 가는 *연속함수continuous function*로 정의된다. 그러면, 의미방정식의 해는 유일하게 항상 어떤 CPO안에 존재하게 된다.

프로그램 C의 의미 $[\![C]\!]$에 대한 의미방정식은 항상 다음과 같고

$$\begin{array}{rcll} [\![C]\!] & = & \mathcal{F}([\![C]\!]) & \\ \text{여기서 } [\![C]\!] & \in & D & \text{(어떤 CPO)} \\ \text{그리고 } \mathcal{F} & \in & D \rightarrow D & (D\text{에서 } D\text{로가는 연속함수들의 CPO)} \end{array}$$

($[\![$`while` E `do` $C]\!]$의 의미방정식에 해당하는 $\mathcal{F}$는 무엇?) 이 방정식의 해는 항상 연속함수 $\mathcal{F}$의 *최소고정점least fixpoint*으로 정의된다. 연속함수의 최소고정점은 CPO의 성질과 연속함수의 조건때문에 항상 유일하게 존재한다. 이 내용은 다음절에서 다루기로하자.

아뭏튼, 이렇게 *최소 고정점least fixed point*이라는 수학적 깃법을 통해서 궁극의 의미가 조립식으로 가능해 지는 이유로해서, 궁극적인 의미구조를 *고정점 의미구조fixpoint semantics*라고 불리기도 한다.

3.2.2.2 CPO, 연속함수, 최소고정점

프로그램의 수학적인(궁극적인) 의미는 CPO(*complete partial order*)라는 공간의 원소로 정의된다. 어느 집합이 CPO가 되려면, 그 집합의 원소들 간에 어떤 순서가 있고(임의의 두 원소간에 순서가 있을 필요는 없다, 따라서 *"partial" order*), 모든 원소보다 아래에 있는 밑바닥 원소(주로 $\perp$으로 표현한다)가 항상 있고, 그 순서를 가지고 일렬로 줄을 세울 수 있는 원소들(*chain*)이 있다면 그 줄에 있는 모든 원소들 보다 위에 있으면서 가장 작은 원소(LUB, *least upper bound*)를 항상 가지고 있는 집합이다.

CPO D_1에서 CPO D_2로 가는 함수 $f : D_1 \rightarrow D_2$가 연속 함수란, *체인chain*을 체인으로 보내면서, D_1의 *체인chain*의 LUB를 보전해 주는 함수이다. 체인의 LUB을 취하고 함수를 적용한 결과가 함수를 그 체인의 원소들에 각각 취하고

그 결과를 LUB한 것과 일치하는 함수이다:

$$\forall chain\ X \subseteq D_1.f(\bigsqcup X) = \bigsqcup_{x\in X} f(x).$$

CPO에서 CPO로 가는 연속함수 f는 항상 최소 고정점 $fix f$이 유일하게 있고, 그것은

$$\bot \sqcup f(\bot) \sqcup f(f(\bot)) \sqcup \cdots = \bigsqcup_{i\in\mathbb{N}} f^i(\bot)$$

이다.(왜?)

CPO성질을 만족하는 집합들은 매우 다양하게 만들 수 있다. 집합을 가지고 CPO를 만들 수 있고, 만들어 논 CPO들을 가지고 조합해서 새로운 구조의 CPO를 만들 수 있다. 집합에 밑바닥 원소를 하나 추가하고 올려붙인($x \sqsubseteq y$ iff $x = y \vee x = \bot$) 집합은 CPO이다. CPO와 CPO의 *데카르트 곱Cartesian product*도 CPO이다. CPO와 CPO의 *출신을 기억하는 합separated sum*도 CPO이다. CPO에서 CPO로 가는 연속함수들의 집합도 CPO이다. (이렇게 CPO들을 가지고 구축한 CPO들의 원소 사이의 순서는 부품 CPO들의 순서를 가지고 정의되는데, 그 순서를 어떻게 정의해야 결과가 CPO가 될까?)

Example 19 다음과 같이 정수의 집합 $\mathbb{Z}$에서 올려붙인 집합 $\mathbb{Z}_\bot = \mathbb{Z} \cup \{\bot\}$을 생각하자. $\mathbb{Z}$ 원소들 사이의 순서는 없고, 순서는 오직 $\bot$과 $\mathbb{Z}$ 사이에만 존재한다: $\forall x \in \mathbb{Z}.\bot \sqsubseteq x$. 모든 체인은 유한한 길이를 가지고 있으면 맨 꼭대기의 원소가 그 체인의 LUB이 된다. 따라서 $\mathbb{Z}_\bot$은 CPO이다. □

Example 20 CPO D_1과 D_2의 *데카르트 곱Cartesian product*

$$D_1 \times D_2 = \{\langle x, y\rangle \mid x \in D_1, y \in D_2\}$$

은 CPO가 된다. 이 곱집합의 원소들 사이의 순서를 *조립식component-wise*으로 했을 때이다:

$$\langle x, y\rangle \sqsubseteq \langle x', y'\rangle \text{ iff } x \sqsubseteq_{D_1} x' \ \wedge\ y \sqsubseteq_{D_2} y'.$$

이때, 최소의 원소는 $\langle \bot_{D_1}, \bot_{D_2} \rangle$가 된다. □

Example 21 CPO D_1과 D_2의 합

$$D_1 + D_2 = \{\langle x, 1\rangle \mid x \in D_1\} \cup \{\langle x, 2\rangle \mid x \in D_2\} \cup \{\bot\}$$

은 CPO가 된다. 이 합집합의 원소들 사이의 순서는, $\bot$이 가장 작고, 그외에는 고향이 같은 원소들끼리 그 고향에서의 순서로 정했을 때이다:

$$\begin{aligned} \langle x, 1\rangle \sqsubseteq \langle x', 1\rangle \quad &\text{iff} \quad x \sqsubseteq_{D_1} x' \\ \langle x, 2\rangle \sqsubseteq \langle x', 2\rangle \quad &\text{iff} \quad x \sqsubseteq_{D_2} x' \end{aligned}$$

□

여기서 잠깐, 인자가 x인 연속함수를 표현하는 방법을 이야기 하고 가자. 연속함수는 *람다계산법Lambda Calculus*에서 사용하는 함수를 표현하는 방법을 빌려서, "$\lambda x.$함수 몸통" 로 표현하도록 하자. 예를들어, 1을 더하는 연속함수는 "$\lambda x.x + 1$"로 표현한다.

Example 22 CPO D_1에서 D_2로 가는 모든 연속함수의 집합 $D_1 \rightarrow D_2$ 은 CPO가 된다. 연속함수들 사이의 순서를 *조립식point-wise*으로 정의했을 때이다:

$$f \sqsubseteq g \text{ iff } \forall x \in D_1. f(x) \sqsubseteq_{D_2} g(x).$$

가장작은 원소는 $\lambda x.\bot_{D_2}$가 된다. □

의미방정식에서 사용하는 모든 물건들은 CPO의 원소이다. 연산자들은 모두 CPO에서 CPO로 가는 연속함수들이고, 연산자가 아닌 물건들도 모두 CPO의 원소들이다. 따라서 모든 의미방정식의 해는 항상 어떤 연속함수 $\mathcal{F}$의 고정점임을 나타내고 있고:

$$X = \mathcal{F}(X)$$

위의 방정식의 해는 $\mathcal{F}$의 최소 고정점 $fix\mathcal{F}$로 정의할 수 있다:

$$fix\mathcal{F} = \sqcup_{i\in\mathbb{N}}\mathcal{F}^i(\bot)$$

자 이제, CPO와 연속함수의 최소 고정점이라는 장치를 이용해서 while-문의 의미가 어떻게 조립식이 되는 지 보자.

우선, 우리가 직관적으로 구성한 의미공간들은 모두 CPO가 되야 한다:

$$\begin{array}{rcll} M \in \mathit{Memory} & = & \mathit{Var} \to \mathit{Value} & \text{연속함수 CPO} \\ z \in \mathit{Value} & = & \mathbb{Z}_\bot & \text{올려붙인 CPO} \\ x \in \mathit{Var} & = & \mathit{ProgramVariable}_\bot & \text{올려붙인 CPO} \\ \text{명령문 } C\text{의 의미, 연속함수 } [\![C]\!] & \in & \mathit{Memory} \to \mathit{Memory} & \text{연속함수 CPO} \\ \text{계산식 } E\text{의 의미, 연속함수 } [\![E]\!] & \in & \mathit{Memory} \to \mathbb{Z}_\bot & \text{연속함수 CPO} \end{array}$$

그 위에서 while-문의 의미방정식은

$$[\![\texttt{while}\ E\ \texttt{do}\ C]\!]M = \mathit{if}\ [\![E]\!]M \neq 0\ \mathit{then}\ [\![\texttt{while}\ E\ \texttt{do}\ C]\!]([\![C]\!]M)\ \mathit{else}\ M$$

이다. 메모리를 받아서 메모리를 내어놓는 연속함수의 표현($\lambda M.\cdots$)으로 다시 쓰면,

$$[\![\texttt{while}\ E\ \texttt{do}\ C]\!] = \lambda M.\mathit{if}\ [\![E]\!]M \neq 0\ \mathit{then}\ [\![\texttt{while}\ E\ \texttt{do}\ C]\!]([\![C]\!]M)\ \mathit{else}\ M.$$

위의 방정식을

$$X = \mathcal{F}(X)$$

의 모양으로 보면, X는 $[\![\texttt{while}\ E\ \texttt{do}\ C]\!]$에 해당하고, 연속함수 $\mathcal{F}$는

$$\begin{array}{l} \lambda X.(\lambda M.\mathit{if}\ [\![E]\!]M \neq 0\ \mathit{then}\ X([\![C]\!]M)\ \mathit{else}\ M) \\ \in\ (\mathit{Memory} \to \mathit{Memroy}) \to (\mathit{Memory} \to \mathit{Memroy}) \end{array}$$

에 해당하게 되므로, while-문의 의미 $[\![\texttt{while}\ E\ \texttt{do}\ C]\!]$는 위 함수의 최소 고정점으로 정의된다:

$$\begin{array}{rcl}[\![\texttt{while}\ E\ \texttt{do}\ C]\!] & = & \mathit{fix}\mathcal{F} \in \mathit{Memory} \to \mathit{Memory} \\ & = & \mathit{fix}(\lambda X.(\lambda M.\mathit{if}\ [\![E]\!]M \neq 0\ \mathit{then}\ X([\![C]\!]M)\ \mathit{else}\ M))\end{array}$$

조립식인 것이 보이는가? $[\![\texttt{while}\ E\ \texttt{do}\ C]\!]$는 $[\![E]\!]$와 $[\![C]\!]$를 가지고 조립되어있다.

4 장

기계 중심의 언어

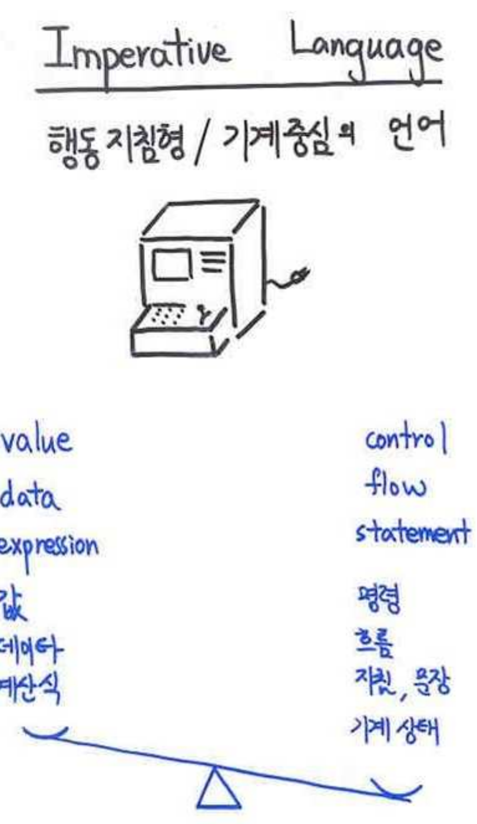
5리에 걸친 안개속. 어떻게 프로그래밍 언어를 디자인해야 하는 지, 무엇이 문제일지가 드러나지 않은 상태. 그렇게 디자인 한 프로그래밍 언어. 그 모습을 따라가 보자. 상식선에서 디자인해 간 언어의 모습.

옆에 컴퓨터가 있다. 사용해야 한다. 컴퓨터에게 시킬 일을 편하게 정의할 수 있는 방법을 고안하자. 이렇게 C 언어를 만들었던 과정은 당시로서는 매우 상식적인 과정을 밟는다.

그러나 되돌아보면 아쉬움이 많다. 현재의 프로그래밍 언어 기술에 기대어 바라볼때 C로 대표되는 옛 시절의 솜씨, 이 과정을 되돌아 볼 것이다.

왜 이 과정을 반복할까? 우선 그 과정에서도 지금까지 살아남은 중요한 개념들이 있다. 살펴볼 것이다. 또 다른 의도는, 과거가 어설펐듯이 현재의 바람직한 프로그래밍 언어 기술이라는 것도 미래에는 어설픈 모습으로 보일 수 있다는 경계.

4.1 주어진 기계



우리에게 주어진 기계는 다음과 같다: 메모리가 있다. 중앙처리장치가 있다. 입출력이 가능하다. 메모리를 읽고, 메모리에 쓸 수 있다. 메모리에서 읽은 것을 화면에 뿌릴 수 있고, 키보드에서 읽은 것을 메모리에 저장할 수 있다. 중앙처리장치*cpu*는 기초적인 기계어 명령들을 처리하는 실행기*interpreter*이다. 실행할 수 있는 기계어 명령의 갯수는 유한하고 고정되어 있다. 그 실행기는 전자회로 디지탈 컴퓨터의 경우 매우 작은 전깃줄들로 구현되어있다.

그 기계는 *폰노이만 머신Von Neuman machine*이다. 기계가 실행할 명령문들이 메모리에 보관될 수 있다. 그 기계는 명령문 하나하나를 메모리에서 읽어와서 실행한다. 메모리에 실려있는 명령문들에 따라 그 기계는 다른 일을 하

는 셈이다.

그 기계는 *보편만능의 기계universal machine*이다. “기계적으로 계산가능한 모든 것”들을 계산해 낼 수 있다. 기계적으로 계산가능한 어떤 일이라도 주어진 명령문들로 구성해 낼 수 있다. 놀랍다. 기계적으로 계산가능한 것은 무한히 많이 있을 수 있다. 하지만 어느 것이라도 유한개의 정해진 기계어 명령들로 조합할 수 있다. 마치 유한한 한국어 단어들만으로 무한히 많은 말을 조합하고 이해할 수 있듯이. 아뭏튼, 그 기계의 *중앙처리장치cpu*는 유한개의 정해진 명령문들만을 알고있다. 그것으로 “보편만능”이 가능한 것이다.

기계가 실행하도록 준비하는 명령문들의 조합이 프로그램이다. 프로그램은 그 기계에 내리는 명령이다. 그 기계는 그 명령대로 충실히 일을 진행한다.

이 기계를 좀 더 편리하게 사용하기 위한 언어를 고안해보자. 그 기계어보다 더 상위의 언어를 고안하자.

4.2 언어 키우기 0: K---

4.2.1 변수: 메모리 주소에 이름 붙이기

프로그램에서는 이름을 사용하고자 한다. 우선, 메모리 주소에 이름을 지을 수 있게 하자. 이름으로 메모리 주소를 대신하도록. 명령문은 여러개가 순서대로 실행되도록 하고싶다. 반복이 있다. 입출력이 있다. 조건에 따라서 실행되는 명령을 구분하고싶다. 이러한 기초적인 조건에 맞추어서 다음과 같은 문법의 언어로 시작해보자.

아래의 정의가 K---언어로 짤 수 있는 프로그램들의 집합을 정의한다. 명령문과 식을 만드는 귀납적인 방법이 정의되있다. 프로그램은 하나의 명령문

이다. 명령문은 여러 명령문들과 식들로 꾸며진다.

$$
\begin{array}{rrcl}
\text{명령문 } \textit{Command} & C & \to & \texttt{skip} \\
& & | & x \texttt{ := } E \\
& & | & C \texttt{ ; } C \\
& & | & \texttt{if } E \texttt{ then } C \texttt{ else } C \\
& & | & \texttt{while } E \texttt{ do } C \\
& & | & \texttt{for } x \texttt{ := } E \texttt{ to } E \texttt{ do } C \\
& & | & \texttt{read } x \\
& & | & \texttt{write } E
\end{array}
$$

$$
\begin{array}{rrcl}
\text{식 } \textit{Expression} & E & \to & n \quad (n \in \mathbb{Z}) \\
& & | & \texttt{true} \mid \texttt{false} \\
& & | & x \\
& & | & E \texttt{ + } E \mid \texttt{-} E \\
& & | & E \texttt{ < } E \mid \texttt{not } E \\
\text{프로그램} & P & \to & C
\end{array}
$$

명령문은 기계의 메모리 상태를 변화시킨다. 식은 기계의 메모리를 참고해서 값을 계산한다.

각 명령문과 식의 의미를 정확히 정의하자. 우선 의미를 정의해갈 때 사용할 *원소semantic object*들이 어느 *집합(의미공간)semantic domain*의 원소들인지를 정하자. 값은 정수거나 참/거짓이다. 따라서 값의 집합은 정수집합이거나 참/거짓 집합니다. 메모리는 주소에서 값으로 가는 테이블일 것이다. 테이블은 수학적으로는 하나의 함수이다. 정의구역은 메모리 주소, 공변역은 값. 특히 정의 구역은 메모리 주소 집합중에서 유한한 집합이 되겠다. 주소는 프로그래머가 사용하는 이름들의 집합으로 정의하자.

그래서 의미정의에 사용할 *의미공간semantic domain*들은 아래와 같다:

$$\begin{array}{rcrcl} n & \in & \mathbb{Z} & & Integer \\ b & \in & \mathbb{B} & & Boolean \\ v & \in & Val & = & \mathbb{Z} + \mathbb{B} \\ M & \in & Mem & = & Addr \xrightarrow{\text{fin}} Val \\ x, y & \in & Addr & = & Id \end{array}$$

정수를 n으로 쓸 것이다. 참이나 거짓은 b로, 값은 v로, 메모리는 M으로, 주소는 x, y, z등으로 쓸 것이다.

Notation 1 $\mathbb{Z} + \mathbb{B}$는 두 집합 $\mathbb{Z}$와 $\mathbb{B}$의 합집합을 뜻한다. $Addr \xrightarrow{\text{fin}} Val$는 함수들의 집합을 뜻한다. 함수들의 이미지는 Val집합에 포함된다. 함수들의 정의구역은 집합 $Addr$의 유한한 부분집합이다. 마크 "fin"(finite)을 화살표 위에 붙인 이유이다.

함수 $f \in A \xrightarrow{\text{fin}} B$를 가지고 $f\{x \mapsto v\}$라고 쓰면 (이때, $x \in A$ 이고 $v \in B$), $A \xrightarrow{\text{fin}} B$에 있는 새로운 함수인데 x에서의 이미지만 v로 정해지고 나머지는 f와 똑같은 함수를 뜻한다.□

프로그램의 의미는 논리 시스템의 증명 규칙들로 정의된다. 다음을 증명하는 규칙들이 될 것이다.

$$M \vdash C \Rightarrow M' \quad \text{와} \quad M \vdash E \Rightarrow v$$

$M \vdash C \Rightarrow M'$는 메모리 M에서 명령 C를 실행하면 메모리 상태가 M'로 변한다, 는 사실을 뜻한다. $M \vdash E \Rightarrow v$는 메모리 M에서 식 E를 실행하면 값이 v가 계산된다, 는 사실을 뜻한다.

아래는 위와 같은 사실들을 증명하는 논리 시스템이다. 주어진 프로그램 C가 있을 때, 이 논리 시스템에서 $M \vdash C \Rightarrow M'$가 증명되는 경우만 프로그램 C는 의미가 있는 것이다. 그러한 증명이 불가능한 C와 E는 의미 없는 명령문이고 식인 것이다.

$\boxed{M \vdash C \Rightarrow M'}$

$$\frac{}{M \vdash \texttt{skip} \Rightarrow M}$$

$$\frac{M \vdash E \Rightarrow v}{M \vdash x \texttt{ := } E \Rightarrow M\{x \mapsto v\}}$$

$$\frac{M \vdash C_1 \Rightarrow M_1 \quad M_1 \vdash C_2 \Rightarrow M_2}{M \vdash C_1 \texttt{ ; } C_2 \Rightarrow M_2}$$

$$\frac{M \vdash E \Rightarrow \mathit{true} \quad M \vdash C_1 \Rightarrow M'}{M \vdash \texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2 \Rightarrow M'}$$

$$\frac{M \vdash E \Rightarrow \mathit{false} \quad M \vdash C_2 \Rightarrow M'}{M \vdash \texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2 \Rightarrow M'}$$

$$\frac{M \vdash E \Rightarrow \mathit{false}}{M \vdash \texttt{while } E \texttt{ do } C \Rightarrow M}$$

$$\frac{M \vdash E \Rightarrow \mathit{true} \quad M \vdash C \Rightarrow M_1 \quad M_1 \vdash \texttt{while } E \texttt{ do } C \Rightarrow M_2}{M \vdash \texttt{while } E \texttt{ do } C \Rightarrow M_2}$$

$$\frac{\begin{array}{c} M \vdash E_1 \Rightarrow n_1 \quad M \vdash E_2 \Rightarrow n_2 \\ M\{x \mapsto n_1 + 0\} \vdash C \Rightarrow M_0 \\ \vdots \\ M\{x \mapsto n_1 + (n_2 - n_1)\} \vdash C \Rightarrow M_{n_2 - n_1} \end{array}}{M \vdash \texttt{for } x \texttt{ := } E_1 \texttt{ to } E_2 \texttt{ do } C \Rightarrow M'} \; n_2 \geq n_1$$

$$\frac{}{M \vdash \texttt{read } x \Rightarrow M\{x \mapsto n\}}$$

$$\frac{M \vdash E \Rightarrow v}{M \vdash \texttt{write } E \Rightarrow M}$$

$\boxed{M \vdash E \Rightarrow v}$

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

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

$$\frac{M \vdash E_1 \Rightarrow n_1 \quad M \vdash E_2 \Rightarrow n_2}{M \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2}$$

$$\frac{M \vdash E \Rightarrow n}{M \vdash \texttt{-} E \Rightarrow -n}$$

$$\frac{M \vdash E_1 \Rightarrow n_1 \quad M \vdash E_2 \Rightarrow n_2}{M \vdash E_1 \texttt{<} E_2 \Rightarrow n_1 < n_2}$$

$$\frac{M \vdash E \Rightarrow b}{M \vdash \texttt{not}\ E \Rightarrow \mathit{not}\ b}$$

이름이 곧 메모리 주소다. 새 메모리 주소를 사용하고자 하면 다른 이름을 사용하면 된다.

입출력문의 의미를 정한 규칙이 특이하다. 입력문은 외부에서 임의의 정수를 받아서 지정한 메모리 주소에 보관하는 것이다. 외부세계에 묻고 입력받는 과정을 드러낼 필요는 없다. 출력도 외부세계에 계산한 값을 보여주는 과정을 드러낼 필요는 없다. 의미를 결정하는 데 필요한 내용들은 아니므로.

4.2.2 이름의 두가지 의미

이름 안에는 무엇이 있나? 이름이 의미하는 것은 무엇인가? 혼동되게도 K---에서는 두가지 역할을 한다. 이름의 의미가 두 가지다.

우선 메모리 주소에 이름을 붙인 것 이므로, 이름지어진 메모리 주소를 뜻했다:

$$\frac{M \vdash E \Rightarrow v}{M \vdash x \texttt{ := } E \Rightarrow M\{x \mapsto v\}}$$

하지만, 그 이름으로 그 이름이 지칭하는 메모리 주소에 있는 값을 뜻하기도 했다:

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

이렇게 이름이 뜻하는 두개의 값을 이름의 "L-value"(left-value)와 "R-value"(right-value)라고 부른다. 이름이 x := E의 왼쪽에 나타났을 때와 오른쪽 식 E안에 나타났을 때 그 의미하는 바가 다르기 때문이다. 왼쪽에 나타났다면 이름은 메모리 주소를 뜻한다. 오른쪽에 나타났을 때는 그 메모리 주소에 보관된 값을 뜻한다.

메모리 주소의 이름이 어떤 때는 그 주소에 보관되있는 값이 되기도 하는 혼동이 싫다면, 두개의 다른 의미는 두가지 다른 문법으로 표현하도록 하면 될 것이다. 예를 들어, 이름의 의미를 하나로 정의하자. 메모리 주소로. 그리고

그 메모리 주소의 값을 뜻할 때는 이름앞에 !표시를 하도록 하는 것이다. 예를 들어, K---의

$$x := x + 1$$

는

$$x := !x + 1$$

로 쓰도록 하는 것이다. 그런데, 이렇게 하면 프로그래머가 조금 불편해하지 않을까? 프로그램 식에 나타나는 모든 이름앞에 !를 붙여야 하므로

$$x + y + 1 \quad \text{대신에} \quad !x + !y + 1.$$

이러한 편리함이 언어가 복잡해 지면서는 프로그램을 이해하기 어렵게 만드는 눈엣가시가 된다. K---를 키워가면서 살펴볼 것이다.

이름을 기계중심의 언어에서는 *변수variable*라고도 부른다. 이름이 의미하는 메모리 주소의 값이 변할 수 있으므로. 프로그램 실행중에 변수가 의미하는 주소에 새로운 값을 써 넣을 수 있으므로.

4.2.3 이름의 유효범위

프로그래머가 메모리 주소에 이름을 짓기 시작하면서 문제가 생긴다. 같은 이름을 다른 용도로도 쓰고 싶다면? 즉, 같은 이름을 다른 메모리 주소를 위해 재사용하고 싶다면? 이것은 자연스러운 요구이다. 항상 다른 이름을 고안해야 한다면 프로그래머의 불편은 클 것이다. 이 세상에 태어나는 모든 사람들의 이름이 모두 달라야 한다면 이름 짓는 것이 얼마나 골치아프겠는가?

위의 K---언어에서는 같은 이름은 같은 메모리 주소를 뜻한다. 다른 메모리 주소를 위해서는 다른 이름을 사용해야 한다. 같은 이름을 다른 메모리 주소의 이름으로 사용할 수가 없다. 프로그램에서 사용하는 메모리 주소가 매우 많다면, 얼마나 많은 다른 이름을 고안해야 할까? 같은 이름을 다른 메모리 주소를 위해서 쓸 수 는 없을까? 이름을 재활용할 수 는 없을까?

이 문제를 해결한 방안은 이미 있다. 수학이나 모든 엄밀한 논술에서 사용하는 방안이다. 이름의 *유효범위scope*를 일정한 범위로만 제한 하는 것이다. 이름들의 *유효범위scope*를 프로그램 전체중에서 일부분이 되도록 하는 것이다. 그러면 유효범위가 겹치지 않는다면, 같은 이름이 다른 메모리 주소를 뜻하는 것으로 재사용될 수 있는 것이다.

그렇다면, 그 *유효범위scope*는 어떻게 표시할까? 수학에서 사용하는 방안을 빌려오자. 프로그램 텍스트의 범위로 이름의 *유효범위scope*가 결정되도록 하자. 이 방식이 훌륭한 이유는, 수학에서 지난 2000년 이상 성공적으로 사용돼왔기 때문이다. 왜? 이름의 유효범위를 쉽게 알아볼 수 있기 때문이다.

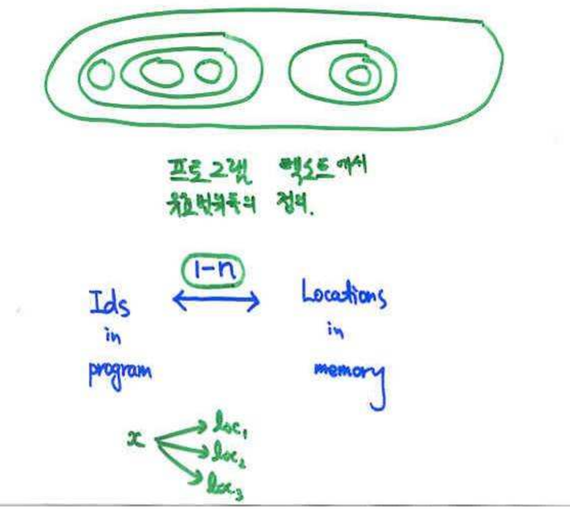
이름의 *유효범위scope*는 다음의 방식으로 정해진다:

$$\begin{array}{rcl} C & \to & \vdots \\ & | & \texttt{let}\ x \texttt{ := } E\ \texttt{in}\ C \end{array}$$

우선 x라는 이름은 새로운 메모리 주소를 뜻한다. 이름을 새롭게 선언하는 셈이다. 그 주소에 저장되는 초기 값은 E의 값이다. 그리고 그 이름이 그 주소를 의미한다는 것은 명령문 C 안에서만 유효하다. 즉, x의 *유효범위scope*는 C 이다. 이름의 *유효범위scope*는 프로그램 텍스트의 일부분인 것이다. 그 범위 이외에서는 x는 다른 메모리 주소를 뜻할 수 있다. 같은 이름이 여러 주소를 뜻하도록 재 사용될 수 있다.

이름이 명령문에 나타날 때, 어느 주소를 뜻하는가? 그 곳을 감싸는 가장 가까운 `let`-명령문이다. 그 이름을 선언한. 당연한 규칙이다. 이름의 실체가 그 이름이 나타나는 프로그램 텍스트에 의해 결정되므로. 프로그램은 나무구조라는 것을 상기하자. 이름이 나타난 곳에서 그 나무구조를 타고 위로 위로 올라가면서 가장 먼저 만나는, 그 이름을 선언한 `let`-명령문. 그곳에서 정한 메모리 주소가 그 이름의 실체가 된다.

그리고 유효범위들은 서로 완전히 포섭되거나 완전히 별개의 경우밖에는 없다. 프로그램의 문법이 그것을 보장한다. 유효범위 두개가 일부분만 겹치는 경우는 없다.



4.2.4 환경

같은 이름이 다른 것을 지칭할 수 있는 프로그램에서, 이름의 실체를 결정해 주는 규칙은 위에서 이야기한대로다. 나타난 이름, 그 곳을 감싸는 가장 먼저 만나는, 그 이름을 선언한 `let`-명령문.

프로그램의 의미를 정확히 정의하는 데는 위와 같은 규칙을 정확히 드러내야 한다. *환경environment*이라는 것을 가지고 의미구조를 정확히 정의할 수 있다. *환경environment*은 이름의 실체를 정의한 함수이다.

$$
\begin{array}{rcll}
\sigma & \in & Env & = \; Id \xrightarrow{\text{fin}} Addr \\
M & \in & Mem & = \; Addr \xrightarrow{\text{fin}} Val \\
\ell & \in & Addr & \quad \text{an index set}
\end{array}
$$

환경은 이름의 실체($\in Addr$, 메모리 주소)를 결정한다. 이름들에서 주소들로 가는 유한 함수이다. 메모리는 주소들에 값들을 대응시키는 유한 함수이다. 주소는 이제 더 이상 프로그램에 나타나는 이름이 될 수 없다. 그것과는 다른 집합이 될 것이다.

환경이 새롭게 *의미구조semantics* 정의에 사용된다. 그래서 프로그램의 의미는 다음 두개를 증명하는 규칙들이 된다:

$$\sigma, M \vdash C \Rightarrow M' \quad \text{와} \quad \sigma, M \vdash E \Rightarrow v$$

예전과는 다르게 *환경environment* σ 가 명령문 C와 식 E의 의미를 정의하는 데 필요하게 되었다. 이름이 C나 E에 나타날 때, 그 이름이 지칭하는 메모리 주소가 어떤 것이 되는 지를 환경 σ가 결정해 준다. 이름은 이제 더이상 곧 메모리 주소가 아니다. 이름이 지칭하는 주소는 현재의 환경(σ)이 결정해 준다.

의미 정의의 판단 "$\sigma, M \vdash C \Rightarrow M'$"는 메모리 M과 환경 σ에서 명령 C를 실행하면 메모리 상태가 M'로 변한다, 는 사실을 뜻한다. $\sigma, M \vdash E \Rightarrow v$는 메모리 M과 환경 σ에서 식 E를 실행하면 값이 v가 계산된다, 는 사실을 뜻한다.

이제는 이름을 사용하고자 하면, 항상 `let`-명령문으로 정의하고 사용해야 한다. 이름은 환경을 통해서 실체가 결정되고, `let`-명령문에 의해서만이 새로운 이름의 실체가 환경(σ)에 덧붙여지기 때문이다:

$$\frac{\sigma, M \vdash E \Rightarrow v \quad \sigma\{x \mapsto \ell\}, M\{\ell \mapsto v\} \vdash C \Rightarrow M'}{\sigma, M \vdash \texttt{let}\ x\ \texttt{:=}\ E\ \texttt{in}\ C \Rightarrow M'}\ \ell \notin \text{Dom}\ M$$

사용하려는 이름의 실체를 현재의 환경(σ)이 모른다면, 프로그램은 의미가 없다. 실행될 수 없다. 이름이 나타나면 그 실체는 항상 현재 환경(σ)을 참조해서 결정된다:

$$\frac{\sigma, M \vdash E \Rightarrow v}{\sigma, M \vdash x\ \texttt{:=}\ E \Rightarrow M\{\sigma(x) \mapsto v\}}$$

$$\frac{}{\sigma, M \vdash x \Rightarrow M(\sigma(x))}$$

$$\frac{}{\sigma, M \vdash \texttt{read}\ x \Rightarrow M\{\sigma(x) \mapsto n\}}$$

같은 이름이 다른 주소를 지칭할 수 있다. 이름의 재활용, 동명이인. 그러나 거꾸로는 아니다. 다른 이름이 같은 주소를 지칭할 수는 없다. 같은 대상을 지칭하는 다른 이름은 가능하지 않다. 별명*alias*은 없다. 위의 언어로 별명*alias*이 만들어지는 경우는 없다.

$\boxed{\sigma, M \vdash C \Rightarrow M'}$

$$\frac{\sigma, M \vdash E \Rightarrow v \quad \sigma\{x \mapsto \ell\}, M\{\ell \mapsto v\} \vdash C \Rightarrow M'}{\sigma, M \vdash \texttt{let}\ x\ \texttt{:=}\ E\ \texttt{in}\ C \Rightarrow M'}\ \ell \notin \mathrm{Dom}\, M$$

$$\overline{\sigma, M \vdash \texttt{skip} \Rightarrow M}$$

$$\frac{\sigma, M \vdash E \Rightarrow v}{\sigma, M \vdash x\ \texttt{:=}\ E \Rightarrow M\{\sigma(x) \mapsto v\}}$$

$$\frac{\sigma, M \vdash C_1 \Rightarrow M_1 \quad \sigma, M_1 \vdash C_2 \Rightarrow M_2}{\sigma, M \vdash C_1\ ;\ C_2 \Rightarrow M_2}$$

$$\frac{\sigma, M \vdash E \Rightarrow \mathit{true} \quad \sigma, M \vdash C_1 \Rightarrow M'}{\sigma, M \vdash \texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \Rightarrow M'}$$

$$\frac{\sigma, M \vdash E \Rightarrow \mathit{false} \quad \sigma, M \vdash C_2 \Rightarrow M'}{\sigma, M \vdash \texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \Rightarrow M'}$$

$$\frac{\sigma, M \vdash E \Rightarrow \mathit{false}}{\sigma, M \vdash \texttt{while}\ E\ \texttt{do}\ C \Rightarrow M}$$

$$\frac{\sigma, M \vdash E \Rightarrow \mathit{true} \quad \sigma, M \vdash C \Rightarrow M_1 \quad \sigma, M_1 \vdash \texttt{while}\ E\ \texttt{do}\ C \Rightarrow M_2}{\sigma, M \vdash \texttt{while}\ E\ \texttt{do}\ C \Rightarrow M_2}$$

$$\frac{\begin{array}{c}\sigma, M \vdash E_1 \Rightarrow n_1 \quad \sigma, M \vdash E_2 \Rightarrow n_2 \\ \sigma, M\{x \mapsto n_1 + 0\} \vdash C \Rightarrow M_0 \\ \vdots \\ \sigma, M\{x \mapsto n_1 + (n_2 - n_1)\} \vdash C \Rightarrow M_{n_2 - n_1}\end{array}}{\sigma, M \vdash \texttt{for}\ x\ \texttt{:=}\ E_1\ \texttt{to}\ E_2\ \texttt{do}\ C \Rightarrow M'}\ n_2 \geq n_1$$

$$\overline{\sigma, M \vdash \texttt{read}\ x \Rightarrow M\{\sigma(x) \mapsto n\}}$$

$$\frac{\sigma, M \vdash E \Rightarrow v}{\sigma, M \vdash \texttt{write}\ E \Rightarrow M}$$

$\boxed{\sigma, M \vdash E \Rightarrow v}$

$$\frac{}{\sigma, M \vdash n \Rightarrow n}$$

$$\frac{}{\sigma, M \vdash x \Rightarrow M(\sigma(x))}$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow n_1 \quad \sigma, M \vdash E_2 \Rightarrow n_2}{\sigma, M \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2}$$

$$\frac{\sigma, M \vdash E \Rightarrow n}{\sigma, M \vdash \texttt{-}\ E \Rightarrow -n}$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow n_1 \quad \sigma, M \vdash E_2 \Rightarrow n_2}{\sigma, M \vdash E_1 \texttt{ < } E_2 \Rightarrow n_1 < n_2}$$

$$\frac{\sigma, M \vdash E \Rightarrow b}{\sigma, M \vdash \texttt{not}\ E \Rightarrow \mathit{not}\ b}$$

4.2.5 자유로운 혹은 묶여있는

프로그램에서 사용되는 이름의 실체를 찾을 수 없다면 그 이름은 *자유로운 이름**free variable*이라고 한다. 자유로운 이름을 가지고 있는 프로그램의 의미는 정의 될 수 없다.

프로그램내에서는 모든 이름들은 `let`-명령문의 선언으로 정의되어 있어야 한다. 이름의 실체가 정의된 정상적인 경우를 *묶여있는 이름**bound variable*이라고 한다.

묶여 있는 이름과 자유로운 이름
Bound v.s. Free Ids

주어진 유효범위에서 이름이 묶여있거나 자유로울 수 있다.

e.g.)
```
let x := 1 in
  let y := 2 in
    y := x+y
  let x := 3 in
    let y := 4 in
      y := y+1
    x := x+1
  x := x+1
```

e.g.)
```
let x := 1 in
  x := x+y
```

전체 프로그램이 아니라 제한된 지역을 기준으로 자유로운 이름과 묶여있는 이름이 이야기되기도 한다. 프로그램 텍스트의 어느 지역만 한정해서 보면서, 그 안에서 사용되는 이름이 자유롭거나 묶여있다고 판단하기도 한다.

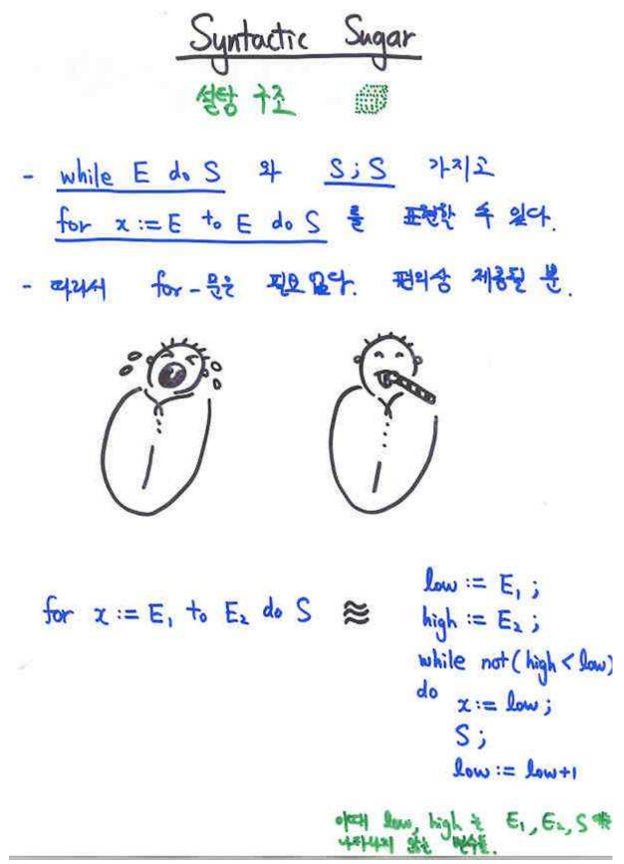
4.2.6 설탕구조

프로그래밍 언어는 꼭 필요한 것만으로 구성되어있지는 않다. 위의 K--- 언어에서도 불필요하지만 사용자의 편의를 위해서 제공해 주는 것이 있다. 예를 들어, for x := E_1 to E_2 do C는 while문을 이용해서 같을 일을 하는 명령어로 바뀔 수 있다:

$$\texttt{for}\ x\ \texttt{:=}\ E_1\ \texttt{to}\ E_2\ \texttt{do}\ C \quad \equiv \quad \begin{array}{l} low\ \texttt{:=}\ E_1;\ high\ \texttt{:=}\ E_2; \\ \texttt{while}\ high \geq low\ \texttt{do}\ (x\ \texttt{:=}\ low;\ C;\ low\ \texttt{:=}\ low + 1) \end{array}$$

단, 위와 같이 for-문을 "녹이는"것이 항상 옳으려면 *low*와 *high*라는 이름들이 E_1, E_2, C에서 사용되어서는 않된다. 특히, 위의 for-문이 프로그램에 포함되 있을 수 있는데, 이 경우에는 *low*와 *high*라는 이름들이 E_1, E_2, C 뿐 아니라 전체 프로그램에서 사용되어서는 않된다.

아뭏튼, 이렇게 편의를 의해서 제공하는 문법 구조를 *설탕구조syntactic sugar*라고 한다. *설탕구조syntactic sugar*는 언어에 진정 새로운 것을 제공하지는 않는다.



프로그래밍 언어에서 *설탕구조syntactic sugar*는 프로그래머의 편의를 위해서 제공될 뿐이다. 프로그래밍 언어 자체를 연구하는 사람들은 그런 모든 설탕을 없앤 최소의 코어만을 가지고 연구한다. 프로그래밍 언어를 처리하는 *실행기interpreter*나 *번역기compiler*도 실행이나 번역전에 프로그램에 뿌려져있는 모든 설탕을 코어만으로 다시 쓰는 작업을 진행한다. 그리하여 실행이나 번역은 작은 코아에 대해서만 정의해 놓으면 되도록.

4.3 언어 키우기 I: K--

4.3.1 프로시져: 명령문에 이름 붙이기

메모리 주소뿐 아니라, 명령문에도 이름을 붙일 수 있도록 하자. 그래서 그 명령문을 반복해서 쓰지 않고도 이름 만으로 그 명령문을 수행할 수 있도록.

조금 더 나아가, 이름붙은 명령문들을 일반화해서 인자에 의해서 조금씩 다른 일을 할 수 있도록 하자. 예를 들어, 변수 x의 값을 읽어서 1을 더한 값을 다시 쓰는 명령문 보다는, 변수 x의 값을 임의의 크기만큼 증가시키는 명령문으로 일반화하자. 그 명령문에 인자가 있어서 인자를 통해서 증가시키고 싶은 양을 전달할 수 있도록. 이렇게 인자를 받는 명령문에 이름을 붙인 것을 프로시져*procedure*라고 한다.

이름을 지을 때는 항상 그 유효범위를 정하도록 하므로, 기존의 `let`-명령문을 이용해서 프로시져를 정의할 수 있도록 하자:

$$\begin{array}{rcl} C & \to & \vdots \\ & | & \texttt{let proc } f(x) \texttt{ = } C \texttt{ in } C \\ & | & f\ (E) \end{array}$$

`let proc` $f(x)$ `=` C `in` C'에서는 두개의 이름이 선언되었다. 프로시져 이름 f와 인자 이름 x. f는 명령문 C의 이름이다. x는 프로시져가 사용될 때의 인자 값을 보관하는 메모리 주소의 이름이다.

인자 이름 x의 *유효범위scope*는 C이다. 프로시져 이름 f의 *유효범위scope*는 C'이다. 재귀호출이 가능한 프로시져를 위해서는 f의 유효범위는 C'뿐 아니라 C까지도 포함되야 한다.

프로시져를 사용하는 명령문 $f\ (E)$은 f로 이름붙은 명령문을 실행시킨다. 그런데, 인자로 E 값을 사용하라는 것이다.

의미를 정확히 정의하자. 우선 *환경environment*이 확장된다. 이름의 실체는

메모리 주소뿐 아니라 프로시져까지 포함된다:

$$\sigma \in Env = Id \xrightarrow{\text{fin}} Addr + Procedure$$

그러나 프로지져의 실체 *Procedure* 가 무엇이 되야 하는지에 따라서 프로시져의 호출에 관한 의미가 전혀 달라질 수 있다.

4.3.2 이름의 유효 범위

4.3.2.1 동적으로 결정되는 유효범위

우선

$$Procedure = Id \times Command$$

로 정의해보자.

Notation 2 두 집합 A와 B가 있을 때, $A \times B$는 두 집합의 원소들의 쌍으로 구성된 집합을 뜻한다:

$$A \times B = \{\langle a, b\rangle \mid a \in A, b \in B\}.$$

□

메모리는 변함없다:

$$\begin{array}{rcll} M & \in & Mem & = Addr \xrightarrow{\text{fin}} Val \\ \ell & \in & Addr & \quad \text{an index set} \end{array}$$

의미 규칙은:

$$\frac{\sigma\{x \mapsto \langle x, C_1\rangle\}, M \vdash C \Rightarrow M'}{\sigma, M \vdash \texttt{let proc}\ f(x) = C_1\ \texttt{in}\ C \Rightarrow M'}$$

$$\frac{\sigma(f) = \langle x, C\rangle \quad \ell \notin \text{Dom}\,\sigma \quad \sigma, M \vdash E \Rightarrow v \quad \sigma\{x \mapsto \ell\}, M\{\ell \mapsto v\} \vdash C \Rightarrow M'}{\sigma, M \vdash f\ (E) \Rightarrow M'}$$

- 프로시져가 선언되면 프로시져 이름의 실체는 해당 명령문과 인자 이름이 된다.

- 프로시져 호출이 일어나면 인자에 대해서는 let-명령문 같은 일이 벌어진다. 새로운 메모리 주소를 프로시져 인자 이름으로 명명하고 그 주소에 인자로 전달할 값을 저장하고, 프로시져 이름이 지칭하는 명령문을 실행한다.

  프로시져 이름이 지칭하는 명령문을 프로시져 몸통이라고 부른다.

위의 의미 정의를 보면, 프로시져 호출은 명령문 텍스트가 움직이는 것과 같은 효과를 가진다. 프로시져 몸통 명령문이 프로시져 호출이 일어난 곳으로 끼여들어오는.

이렇게 명령문이 프로그램안에서 움직여 다닌다면, 프로시져의 몸통 명령문안에 있는 이름들의 실체는 어떻게 결정되야 할까? 이름들의 실체가 프로그램 텍스트의 위치로 결정되야 하는데, 텍스트가 실행중에 옮겨다니는 형국이 되버렸으니.

예를 들어 다음의 프로그램을 생각하자.

```
let x := 0 in
  let proc inc(n) = x := x+n in
    let x := 1 in (inc(1); write x)
```

프로지셔 호출 inc(1)의 실행중에 몸통 명령문 x := x+n에 나타나는 x의 실체는 어느것인가? 첫번째로 선언된 x인가? 두번째로 선언된 x인가?

위의 의미 정의에 따르면, 두번째로 선언된 x이다. 왜냐하면 프로시져가 호출되어 몸통 명령문이 실행될 때의 환경은 프로시져가 호출될 때의 환경을 참조하기 때문이다.

프로시져 몸통 명령문안에 있는 *자유 변수free variabl*들의 실체가 그 프로시져가 어디에서 호출되냐에 따라서 결정된다. 몸통 명령문의 위치가 아니라 그 프로시져의 호출 위치에 따라.

이렇게 이름의 실체가 실행중에 결정되는 것을 동적으로 *유효범위가 정해진다*dynamic scoping라고 한다.

4.3.2.2 정적으로 결정되는 유효 범위

이름의 실체가 실행전에 결정되야하지 않을까? *정적 유효범위*static scoping를 유지하고 싶다.

프로시져 몸통 명령문안에 있는 *자유 변수*free variabl들의 실체가 몸통 명령문의 위치에 의해서 미리 결정되야 하지 않을까? 그 프로시져가 어디에서 호출되냐에 따라서 실행중에 다이나믹하게 결정되는 것 보다는.

그래야 프로그램을 이해하기 쉬워진다. 프로그램을 이해하기 간단하고 쉬운 방식이다. 그리고 이것이 수학에서 인류가 2000년 이상 사용한 규칙이기도 하다.

이것을 위해서는 프로시져의 의미가 인자 이름과 몸통 명령문의 쌍으로는 부족하다. 그 프로시져가 정의되는 위치에서의 환경까지 가지고 있어야 한다:

$$Procedure = Id \times Command \times Env$$

이렇게 되면 프로시져 정의와 호출의 의미가 아래와 같이 정의된다:

$$\frac{\sigma\{x \mapsto \langle x, C_1, \sigma\rangle\}, M \vdash C \Rightarrow M'}{\sigma, M \vdash \texttt{let proc } f(x) \texttt{ = } C_1 \texttt{ in } C \Rightarrow M'}$$

$$\frac{\begin{array}{c}\sigma(f) = \langle x, C, \sigma'\rangle \quad \ell \notin \mathrm{Dom}\,\sigma' \\ \sigma, M \vdash E \Rightarrow v \quad \sigma'\{x \mapsto \ell\}, M\{\ell \mapsto v\} \vdash C \Rightarrow M'\end{array}}{\sigma, M \vdash f\ \texttt{(}E\texttt{)} \Rightarrow M'}$$

4.3.3 프로시져 호출 방법

지금까지는 *값전달 호출*call-by-value이다:

$$\frac{\begin{array}{c}\sigma(f) = \langle x, C, \sigma'\rangle \quad \ell \notin \mathrm{Dom}\,\sigma' \\ \sigma, M \vdash E \Rightarrow v \quad \sigma'\{x \mapsto \ell\}, M\{\ell \mapsto v\} \vdash C \Rightarrow M'\end{array}}{\sigma, M \vdash f\ \texttt{(}E\texttt{)} \Rightarrow M'}$$

호출될 때 인자식 E의 값 v가 프로시져의 *인자 이름formal parameter*에 전달되고 프로시져 몸통 명령어가 실행된다.

그런데, 이름이 뜻하는 메모리 주소를 전달해 줄 방법이 있었으면 한다. 그러기 위해서는 우선, 프로시져 호출문의 생김새부터 구분을 해 주자. *값전달 호출call-by-value*인 경우와 *주소전달 호출call-by-reference*인 경우를:

$$
\begin{array}{rcl}
C & \to & \vdots \\
 & | & \texttt{let proc}\ f(x) = C\ \texttt{in}\ C \\
 & | & f\ (E) \\
 & | & f\ \texttt{<}x\texttt{>}
\end{array}
$$

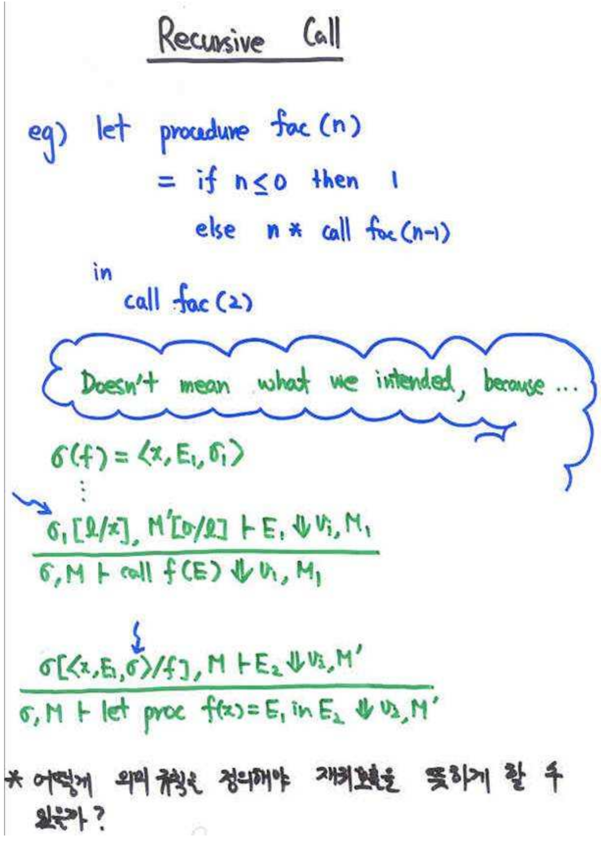
*주소전달 호출call-by-reference*의 정의는 다음과 같다:

$$
\frac{\sigma(f) = \langle x, C, \sigma' \rangle \quad \sigma'\{x \mapsto \sigma(y)\}, M \vdash C \Rightarrow M'}{\sigma, M \vdash f\ \texttt{<}y\texttt{>} \Rightarrow M'}
$$

이렇게 되면, 다른 이름이 같은 메모리 주소를 지칭할 수 있게된다. 같은 대상을 지칭하는 다른 두개의 이름들, *별명alias*이 생기게된다.

별명이 많아지면 프로그램을 이해하기가 어려워지지 않을까? 특히, 별명들이 프로시져 호출에 의해서 실행중에 만들어진다. 프로시져 f의 *인자 이름formal parameter*이 x라고 하면, $f\ \texttt{<}y\texttt{>}$에 의해서는 x와 y가 별명이 되었다가, 또 다른 호출 $f\ \texttt{<}z\texttt{>}$에 의해서는 x와 z가 별명이 된다. 이런 다이나미즘, 혼동스럽다. 이런 복잡한 상황을 지원하는 언어는 바람직한가?

4.3.4 재귀 호출



의미정의를 보면, 재귀 호출은 불가능했다:

$$\frac{\sigma(f) = \langle x, C, \sigma' \rangle \quad \ell \notin \mathrm{Dom}\,\sigma' \qquad \sigma, M \vdash E \Rightarrow v \quad \sigma'\{x \mapsto \ell\}, M\{\ell \mapsto v\} \vdash C \Rightarrow M'}{\sigma, M \vdash f\ (E) \Rightarrow M'}$$

몸통 명령문안에 자신의 프로시져 이름 f가 나타나면 자신임을 *환경*environment σ'에서 알려줘야 한다. 그러나 σ'는 프로시져가 정의되는 위치를 감싸는 환경이었다.

재귀호출을 지원하는 의미정의는 아래와 같다:

$$\frac{\sigma(f) = \langle x, C, \sigma' \rangle \quad \ell \notin \mathrm{Dom}\,\sigma' \qquad \sigma, M \vdash E \Rightarrow v \quad \sigma'\{x \mapsto \ell\}\{f \mapsto \langle x, C, \sigma' \rangle\}, M\{\ell \mapsto v\} \vdash C \Rightarrow M'}{\sigma, M \vdash f\ (E) \Rightarrow M'}$$

4.3.5 명령문과 식의 통합

프로시져 호출은 명령문이었다. 메모리에 반응을 일으키는. 프로시져 호출의 결과로 값이 계산되게 하고 싶으면? 메모리를 변화시킨 후에 최종적으로 어느 값을 결과로 내 놓는 식이 있으면 어떤가? C에서는 `return` E라는 명령문이 이 역할을 한다.

그런데 생각해 보자. 명령문과 식, 두 가지가 과연 다른 것들인가? 메모리에 값을 쓸 수 있는 것과 메모리를 읽기만 하는 것. 결과 값이 없는 것과 결과 값이 있는 것. 하나로 합칠 수 있다. 식의 실행결과는 항상 어떤 값이 된다. 그리고 식은 메모리 변화를 일으킬 수 도 있다.

$$
\begin{array}{lcll}
\text{식 } \mathit{Expression} & E & \to & \texttt{skip} \\
 & & \mid & x \texttt{ := } E \mid E \texttt{ ; } E \\
 & & \mid & \texttt{if } E \texttt{ then } E \texttt{ else } E \\
 & & \mid & \texttt{while } E \texttt{ do } E \\
 & & \mid & \texttt{for } x \texttt{ := } E \texttt{ to } E \texttt{ do } E \\
 & & \mid & \texttt{read } x \mid \texttt{write } E \\
 & & \mid & \texttt{let } x \texttt{ := } E \texttt{ in } E \\
 & & \mid & \texttt{let proc } f(x) \texttt{ = } E \texttt{ in } E \\
 & & \mid & f\ (E) \mid f \texttt{<}x\texttt{>} \\
 & & \mid & n \quad (n \in \mathbb{Z}) \\
 & & \mid & \texttt{true} \mid \texttt{false} \\
 & & \mid & x \\
 & & \mid & E \texttt{ + } E \mid \texttt{-} E \\
 & & \mid & E \texttt{ < } E \mid \texttt{not } E \\
\text{프로그램} & P & \to & E
\end{array}
$$

그래서 프로그래머는 더욱 간단한 언어를 가지고 더욱 자유롭게 프로그램하게 될 것이다. 간단한 이유는 모든 것은 식이므로. 자유스러운 이유는 식과 명령문이 제한없이 섞일 수 있으므로. 명령문과 식, 두가지 다른 타입의 프로그램

부품을 운용할 필요가 없다.

의미 정의는

$$\sigma, M \vdash E \Rightarrow v, M'$$

을 증명하는 규칙들로 통일된다. 명령문들은 의미없는 빈 값 "·"을 계산한다고 하자:

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

실은, 프로그램에서 새로운 타입의 값을 다룰 수 있게 하려면, 그 언어는 그 집합의 값을 만들고 사용하는 문법을 제공해 주게 된다. 새로운 값 ·을 만드는 문법은? 이미 있는 skip 명령문이 그 값을 만드는 식이라고 하자. 덩달아서, 이전의 모든 명령문들이 그 값을 만드는 식이라고 한 것이다. 그리고 ·를 사용하는 문법은 없다. 불행하게도 ·은 만들어지기만 하고 사용할 데는 없는 무의미한 값이다.

$\boxed{\sigma, M \vdash E \Rightarrow v, M'}$

$$\frac{\sigma, M \vdash E_1 \Rightarrow v \quad \sigma\{x \mapsto \ell\}, M\{\ell \mapsto v\} \vdash E_2 \Rightarrow v', M'}{\sigma, M \vdash \texttt{let } x \texttt{ := } E_1 \texttt{ in } E_2 \Rightarrow v', M'} \ \ell \notin \mathrm{Dom}\, M$$

$$\frac{\sigma\{x \mapsto \langle x, E_1, \sigma\rangle\}, M \vdash C \Rightarrow v, M'}{\sigma, M \vdash \texttt{let proc } f\texttt{(}x\texttt{)} \texttt{ = } E_1 \texttt{ in } E \Rightarrow v, M'}$$

$$\frac{}{\sigma, M \vdash \texttt{skip} \Rightarrow \cdot, M}$$

$$\frac{\sigma, M \vdash E \Rightarrow v, M'}{\sigma, M \vdash x \texttt{ := } E \Rightarrow v, M'\{\sigma(x) \mapsto v\}}$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow v_1, M_1 \quad \sigma, M_1 \vdash E_2 \Rightarrow v_2, M_2}{\sigma, M \vdash E_1 \texttt{ ; } E_2 \Rightarrow v_2, M_2}$$

$$\frac{\sigma, M \vdash E \Rightarrow \mathit{true}, M' \quad \sigma, M' \vdash E_1 \Rightarrow v, M''}{\sigma, M \vdash \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \Rightarrow v, M''}$$

$$\frac{\sigma, M \vdash E \Rightarrow \mathit{false}, M' \quad \sigma, M' \vdash E_2 \Rightarrow v, M''}{\sigma, M \vdash \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \Rightarrow v, M''}$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow \mathit{false}, M'}{\sigma, M \vdash \texttt{while } E_1 \texttt{ do } E_2 \Rightarrow \cdot, M'}$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow \mathit{true}, M' \quad \sigma, M \vdash E_2 \Rightarrow v, M_1 \quad \sigma, M_1 \vdash \texttt{while } E_1 \texttt{ do } E_2 \Rightarrow v, M_2}{\sigma, M \vdash \texttt{while } E_1 \texttt{ do } E_2 \Rightarrow v, M_2}$$

$$\frac{\begin{array}{c}\sigma, M \vdash E_1 \Rightarrow n_1, M' \quad \sigma, M \vdash E_2 \Rightarrow n_2, M'' \\ \sigma, M''\{x \mapsto n_1 + 0\} \vdash C \Rightarrow v_0, M_0 \\ \vdots \\ \sigma, M_{n_2-n_1-1}\{x \mapsto n_1 + (n_2 - n_1)\} \vdash C \Rightarrow v_{n_2-n_1}, M_{n_2-n_1}\end{array}}{\sigma, M \vdash \texttt{for } x \texttt{ := } E_1 \texttt{ to } E_2 \texttt{ do } C \Rightarrow \cdot, M_{n_2-n_1}} \; n_2 \geq n_1$$

$$\frac{\begin{array}{c}\sigma(f) = \langle x, E', \sigma' \rangle \quad \ell \notin \mathrm{Dom}\, \sigma' \\ \sigma, M \vdash E \Rightarrow v, M' \quad \sigma'\{x \mapsto \ell\}\{f \mapsto \langle x, E', \sigma' \rangle\}, M'\{\ell \mapsto v\} \vdash E' \Rightarrow v', M''\end{array}}{\sigma, M \vdash f \texttt{ (}E\texttt{)} \Rightarrow v', M''}$$

$$\frac{}{\sigma, M \vdash \texttt{read } x \Rightarrow n, M\{\sigma(x) \mapsto n\}}$$

$$\frac{\sigma, M \vdash E \Rightarrow v, M'}{\sigma, M \vdash \texttt{write } E \Rightarrow v, M'}$$

$$\frac{}{\sigma, M \vdash n \Rightarrow n, M}$$

$$\frac{}{\sigma, M \vdash x \Rightarrow M(\sigma(x)), M}$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow n_1, M_1 \quad \sigma, M_1 \vdash E_2 \Rightarrow n_2, M_2}{\sigma, M \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2, M_2}$$

$$\frac{\sigma, M \vdash E \Rightarrow n, M'}{\sigma, M \vdash \texttt{-}\, E \Rightarrow -n, M'}$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow n_1, M_1 \quad \sigma, M \vdash E_2 \Rightarrow n_2, M_2}{\sigma, M \vdash E_1 \texttt{ < } E_2 \Rightarrow n_1 < n_2, M_2}$$

$$\frac{\sigma, M \vdash E \Rightarrow b, M'}{\sigma, M \vdash \texttt{not } E \Rightarrow \mathit{not}\, b, M'}$$

4.3.6 메모리 관리

K-- 프로그램의 실행에 대해서 생각해 보자. 컴퓨터는 K-- 프로그램을 자동으로 실행해 준다. 프로그램을 구성하는 식의 의미 정의에 따라 컴퓨터는 어김없이 실행할 뿐이다.

이 때 프로그램이 소모하는 자원은 컴퓨터의 시간과 메모리이다. 컴퓨터의 시간은 무한하지만 메모리는 유한하다. 프로그램이 한 없이 컴퓨터 메모리를 소모할 수 는 없다.

메모리가 어디에서 소모되는 지 보자. 프로그램 실행중에 메모리는 `let`-식과 프로시져 호출식의 실행을 위해 하나씩 소모된다:

$$\frac{\sigma, M \vdash E_1 \Rightarrow v \quad \sigma\{x \mapsto \ell\}, M\{\ell \mapsto v\} \vdash E_2 \Rightarrow v', M'}{\sigma, M \vdash \texttt{let}\ x\ \texttt{:=}\ E_1\ \texttt{in}\ E_2 \Rightarrow v', M'}\ \ell \notin \mathrm{Dom}\, M$$

$$\frac{\begin{array}{c}\sigma(f) = \langle x, E', \sigma' \rangle \quad \ell \notin \mathrm{Dom}\, \sigma' \\ \sigma, M \vdash E \Rightarrow v, M' \quad \sigma'\{x \mapsto \ell\}\{f \mapsto \langle x, E', \sigma' \rangle\}, M'\{\ell \mapsto v\} \vdash E' \Rightarrow v', M''\end{array}}{\sigma, M \vdash f\ (E) \Rightarrow v', M''}$$

`let`-식에서 선언된 이름과 프로시져의 인자 이름을 위해서 매번 새로운 메모리가 소모된다. 그리고, 소모되기만 한다. 소모되기만 한다. 소모되기만 한다.

`let`-식과 프로시져 호출이 많은 횟수 반복되게되면 언젠가는 컴퓨터가 메모리 부족으로 프로그램 실행을 못하게된다. 메모리는 유한하기 때문이다.

따라서 메모리는 재활용되야 한다. 어느 메모리가 재활용되야 하나? 프로그램 실행중에 사용한 메모리 중에서 앞으로 더 이상 사용되지 않을 메모리다.

그런 메모리를 어떻게 알아낼 수 있나? 앞으로 더 이상 사용안 될 메모리를.

K-- 프로그램에서는 쉽다. 메모리 주소의 사용기간이 곧 그 주소의 이름의 유효범위이기 때문이다. 메모리 주소는 하나의 이름만 붙고 그 이름을 통해야만 그 메모리 주소를 사용할 수 있기 때문이다. 따라서 이름의 유효범위가 끝나면 그 이름이 지칭하는 메모리는 더 이상 사용될 방법이 없다.

따라서 K-- 프로그램에서는 이름의 유효범위가 끝나는 곳에서 그 이름이 붙은 메모리 주소를 재활용하면 된다. 그 곳이 어디인가? `let`-식의 마지막이

나 프로시져 몸통(인자 이름의 유효범위)의 마지막이다. 그 곳이 선언한 이름들의 유효범위가 끝나는 곳이다.

그러나, 언어가 좀더 확장 되면, 이렇게 간단한 *메모리 재활용garbage collection*이 간단치 않게된다. 메모리 주소가 프로그램 식의 결과 값으로 프로그램의 여기저기를 흘러다니게 되면, 그 메모리 주소의 사용 기간은 그 주소에 붙은 이름의 유효범위와 일치하지 않게 된다. 이런 언어에서의 *메모리 재활용garbage collection* 문제는 4.4.3절에서 다룬다.

4.4 언어 키우기 II: K-

K--는 그럴듯하다. 이름 붙일 수 있는 것은 메모리 주소와 프로그램 코드. 이름의 *유효범위scope*가 있고, *재귀호출recursive call*이 가능하고, *값전달 호출call-by-value*과 *주소전달 호출call-by-reference*이 가능하고. `while`-식과 `for`-식으로 반복이 가능하고.

4.4.1 레코드: 구조가 있는 데이타, 혹은 메모리 뭉치

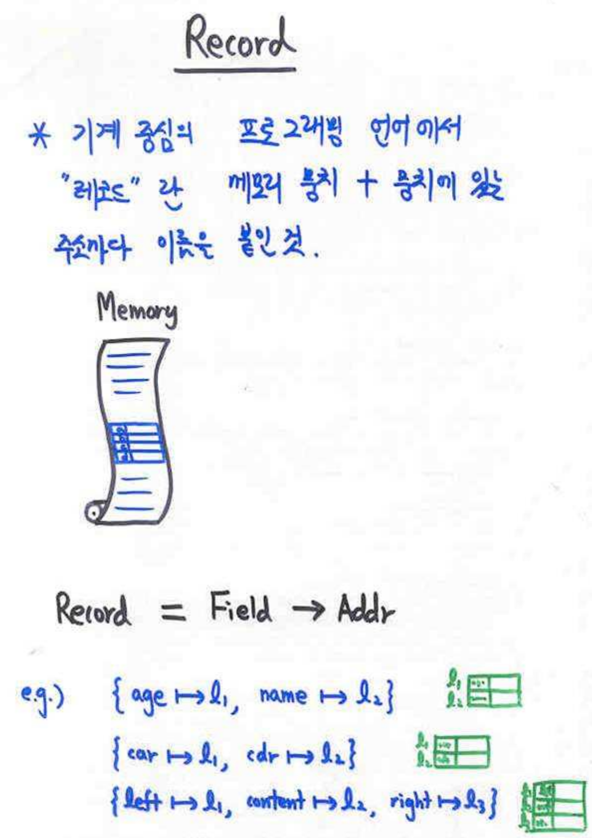
그런데, *리스트list*나 *나무구조tree*를 만들고 사용하는 프로그램을 짤 수 있는가? 자동차를 만들고 사용하는 프로그램을 짤 수 있는가? 집합을 만들고 사용하는 프로그램을 짤 수 있겠는가?

할 수 는 있다. 그런 구조물들을 정수나 참/거짓으로 표현하는 방안만 정한다면. 그러나, 손쉽게 다룰 수 있도록 프로그래밍 언어에서 지원이 되면 좋겠다. *기초적인 값primitive value*(정수나 참/거짓) 이외의, *구조가 있는 값compound value*를 프로그램에서 손쉽게 다룰 수 있도록.

기초적인 값를 보관하는 데는 메모리 주소 하나로 충분했다. 구조가 있는 값은 여러개의 메모리 주소가 필요하다. 왜냐하면 "구조"란 관계를 가지는 것이고 관계는 여러개들을 끌어들여 표현할 수 밖에 없기 때문이다.

구조가 있는 값을 다루는 데는 메모리 뭉치를 하나의 단위로 이름짓고 프로그램할 수 있으면 편리할 것이다. 메모리 뭉치는 메모리 주소가 하나 이상

모여있는 것이다. 그런 것들을 *레코드record*라고 하자. C에서는 `struct`라고한다. 그리고 그 뭉치안에 있는 각각의 메모리 주소들도 이름이 있다. 그러한 이름들을 레코드 필드라고 부르자. 참고로, *배열array*도 메모리 뭉치다, 각각의 메모리 주소들의 이름이 자연수인.



이제 레코드가 프로그램에서 하나의 값이 되어 자유롭게 쓸 수 있도록 할 것이다. 우선 레코드라는 값을 다음과 같이 정의하자:

$$r \in Record = Id \xrightarrow{\text{fin}} Addr$$

한 레코드란 메모리 뭉치들이고 뭉치안의 각 메모리 주소가 이름이 붙은 것이다. 하나의 유한 함수(테이블)로 생각할 수 있다:

$$\{x_1 \mapsto \ell_1, \cdots, x_n \mapsto \ell_n\}.$$

이제 값들은 레코드까지 포함하게되고, 메모리는 레코드를 보관할 수 있게 된다:

$$\begin{array}{rcrcl} v & \in & Val & = & \mathbb{Z} + \mathbb{B} + \{\cdot\} + Record \\ M & \in & Mem & = & Addr \xrightarrow{\text{fin}} Val \end{array}$$

프로그래밍 언어에서 레코드를 만들고 사용하는 문법이 아래와 같이 제공된다:

$$\begin{array}{rcl} E & \rightarrow & \vdots \\ & | & \texttt{\{\}} \\ & | & \texttt{\{}x \texttt{ := } E, y \texttt{ := } E\texttt{\}} \\ & | & E.x \\ & | & E.x \texttt{ := } E \end{array}$$

레코드 필드의 수(메모리 뭉치의 메모리 수)가 0이거나 2개인 것만 생각해도 충분하다.

의미 정의는 다음과 같다:

$$\frac{}{\sigma, M \vdash \texttt{\{\}} \Rightarrow \cdot, M}$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow v_1, M_1 \quad \sigma, M_1 \vdash E_2 \Rightarrow v_2, M_2}{\sigma, M \vdash \texttt{\{}x \texttt{ := } E_1, y \texttt{ := } E_2\texttt{\}} \Rightarrow \{x \mapsto \ell_1, y \mapsto \ell_2\}, M_2\{\ell_1 \mapsto v_1\}\{\ell_2 \mapsto v_2\}} \ \{\ell_1, \ell_2\} \not\subseteq \mathrm{Dom}\, M_2$$

$$\frac{\sigma, M \vdash E \Rightarrow r, M'}{\sigma, M \vdash E.x \Rightarrow M'(r(x)), M'}$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow r, M_1 \quad \sigma, M_1 \vdash E_2 \Rightarrow v, M_2}{\sigma, M \vdash E_1.x \texttt{ := } E_2 \Rightarrow v, M_2\{r(x) \mapsto v\}}$$

레코드에 이름을 붙이고고, 레코드의 필드가 아래와 같이 사용된다.

```
let
  item := {id := 200012, age := 20}
in
  item.id + item.age
```

이제는 나무가 프로그램하기 쉽다. 나무의 각 노드가 레코드가 된다. 필드는 `left`, `v`, `right`이다.

```
let
   tree := {left:={}, v:=0, right:={}}
in
   tree.right := {left:={}, v:=2, right:=3};
   shake(tree)
```

레코드가 메모리 뭉치를 뜻하면서, 그리고 레코드가 값이 되면서, 메모리 주소의 사용기간을 간단히 알 수 없게된다:

```
  ...
f( let
    tree := {left:={}, v:=0, right:={}}
   in
    tree
  )
  ...
```

위의 경우 `tree`의 유효범위는 `let`-식에 한정되지만, `let`-식의 결과로 만들어지는 레코드(메모리 뭉치)들은 프로시져 `f`로 전달되어 사용되게 된다. *메모리 재활용garbage collection*이 어려워 진다(4.4.3절).

4.4.2 포인터: 메모리 주소를 데이타로

지금까지 프로그램에서 값으로 운용되는 것들은 정수, 참/거짓, ·, 그리고 레코드였다:

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

그것들은 자유자제로 사용된다. 변수에 저장되기도 하고, 프로시져에 전달되기도 하고, 프로시져의 결과값이 될 수 도 있고, 또 레코드에 저장되기도 한다. 프로그램 식의 계산 결과로 만들어 지는 것들이었다.

다른 각도에서 이야기 하면, 값들로 자유롭게 운용될 수 있는 것들이란, 반드시 이름이 붙어있을 필요가 없이 계산되는 것들 이기도하다. 정수마다 이름을 붙여서 써야 한다면 얼마나 귀찮은 일인가? 레코드마다 이름이 있어야 한다면? 정수를 더할 때 마다 그 결과에 이름을 붙여야만 그 결과를 사용할 수 있다면? 불편할 것이다. 불평은 정당할 것이다.

그런데, 지금까지 항상 이름을 붙여야만 사용할 수 있는 값들이 있었다. 두 가지였다. 메모리 주소와 프로그램 식(프로시저)이였다.

메모리 주소를 이름 없이도 쓸 수 있게하자. (프로시져에 마저에도 이름을 붙일 필요가 없이 자유자재의 값이 되는 방안은 5장에서 자연스럽게 등장한다.) 즉, 이제는 값들 중에 메모리 주소가 값이 되게 해보자:

$$Val = \mathbb{Z} + \mathbb{B} + \{\cdot\} + Record + Addr$$

프로그램에서 메모리 주소를 값으로 다룰 수 있게 하려면, 그 언어는 메모리 주소 값을 만들고 사용하는 문법을 제공해 줘야 한다. 아래와 같다:

$$\begin{array}{lcl} E & \to & \vdots \\ & | & \texttt{malloc}\, E \\ & | & \texttt{\&}x \ \mid\ \texttt{\&}E.x \\ & | & \texttt{*}E \\ & | & \texttt{*}E\ \texttt{:=}\ E \end{array}$$

`malloc` E와 &-식은 메모리 주소 값을 만드는 식이다. `malloc` E는 새로운 메모리 주소를 할당받아서 그 시작주소를 값으로 하는 식이다. &-식은 메모리 주소의 이름으로 부터 그 주소를 가져오는 식이다. `*`E와 `*`E `:=` E는 메모리 주소에 저장된 값을 가져오거나 그 주소에 값을 저장하는 식이다.

Exercise 1 레코드는 메모리 주소 뭉치이고, 레코드는 값이므로, 레코드를 이용해서 메모리 주소가 값으로 여겨지는 프로그램을 할 수 는 있다. 어떻게?

의미 정의는 다음과 같다.

$$\frac{\sigma, M \vdash E \Rightarrow n, M'}{\sigma, M \vdash \texttt{malloc}\, E \Rightarrow \ell, M'}\ n > 0, \{\ell, \ell+1, \cdots, \ell+n-1\} \not\subseteq \mathrm{Dom}\, M'$$

$$\frac{}{\sigma, M \vdash \texttt{\&}x \Rightarrow \sigma(x), M}$$

$$\frac{\sigma, M \vdash E \Rightarrow r, M'}{\sigma, M \vdash \texttt{\&}E.x \Rightarrow r(x), M'}$$

$$\frac{\sigma, M \vdash E \Rightarrow \ell, M'}{\sigma, M \vdash *E \Rightarrow M'(\ell), M'}$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow \ell, M_1 \quad \sigma, M_1 \vdash E_2 \Rightarrow v, M_2}{\sigma, M \vdash *E_1 \texttt{ := } E_2 \Rightarrow v, M_2\{\ell \mapsto v\}}$$

$*E$는 그 자체가 식인 경우와 메모리 지정문의 왼쪽에 있을 경우, 의미가 다르다. 변수 x가 지정문의 왼쪽이냐 오른쪽이냐에 따라 의미가 달라 졌듯이. $*E$ 가 지정문의 왼쪽에 있는 경우는 E가 뜻하는 메모리 주소값을 뜻한다. 항상 메모리 주소여야 한다. $*E$가 지정문의 오른쪽에 있는 경우는 E가 뜻하는 메모리 주소에 보관되어 있는 값을 뜻한다. (C언어가 이렇다.)

그래서 아래 프로그램을 실행하면 x가 가르키는 주소의 다음번 주소에는 3이 보관된다.

```
let
  x := malloc(2)
in
  *x := 1;
  *(x+1) := *x + 2
```

4.4.3 메모리 관리

이제 K- 프로그램을 실행할 때 컴퓨터의 유한한 메모리를 어떻게 재활용할 수 있는 지를 생각해보자.

이제 프로그램 실행중에 메모리를 소모하는 경우는 let-식과 프로시저 호출식 뿐 아니라 레코드 식과 메모리 할당 식들이 되었다. 그리고 메모리 주소

가 값으로 사용될 수 있기 때문에, 그 주소들의 사용기간이 어떻게 되는 지 알 방법이 쉽지가 않다.

예전의 K-- 언어의 경우는 *메모리 재활용*garbage collection이 쉬웠다. 메모리 주소의 사용기간이 그 주소에 붙은 이름의 유효범위와 일치했기 때문이다. K-- 프로그램을 실행하면서 이름의 유효범위가 끝나는 시점에서 그 이름이 붙은 메모리를 컴퓨터가 재활용하면 되었다.

K-에서는 쉽지가 않다. 어떻게 해야 할까?

4.4.3.1 수동 메모리 재활용

해결을 미루는 쉬운 방법이 있다. 메모리 재활용을 프로그래머에게 맡기는 것이다. 프로그래머의 책임으로. 그래서 프로그래밍 언어에 메모리 재활용 명령문을 제공하는 것이다 (C가 이렇다):

$$\begin{array}{rcl} E & \to & \vdots \\ & | & \texttt{free}\, E \end{array}$$

$\texttt{free}\, E$ 식의 의미를 정의해보자. 식 E는 메모리 주소를 계산한다. 그 주소는 반드시 할당받은 메모리 주소이어야 한다. 메모리 뭉치를 재활용하려면 뭉치에 포함된 모든 주소를 재활용해야 한다.

엄밀하게 정의하면 다음과 같다:

$$\frac{\sigma, M \vdash E \Rightarrow \ell, M'}{\sigma, M \vdash \texttt{free}\, E \Rightarrow \cdot, M'}\ \ell \in \mathrm{Dom}\, M$$

위의 정의에서 이상한 것이 있다. 메모리 재활용 효과가 표현되지 않았다. 메모리는 그대로다. 그래도 괜찮은가?

의미 정의에서 메모리 재활용 효과가 표현되지 않아도 충분하다. 메모리 재활용은 프로그래밍 언어의 의미 바깥의 일이다. 메모리 재활용은 유한한 메모리를 가진 컴퓨터가 K- 프로그램을 실행하게 되서 생기는 문제다. 의미 정의에 명시된 메모리 모델대로(의미 정의대로) 구현하기 위해서 해결해야 하는

문제인 것이다.

의미 정의에는 메모리 재활용에 대해서 명시할 필요는 없다. 의미 정의의 목적은 프로그램의 의미를 우리가 필요로하는 디테일의 정도에서 혼동이 없게 정의하는 것이다. 마치 요리책과 같다. 짜장면의 요리법을 명시하고 있다. 그러나 필요이상의 디테일까지는 명시하지 않는다. 돼지고기를 구하기 위해서 돈은 어떻게 벌 수 있는지, 센 불을 구하려면 어디로 가야 하는 지, 등은 명시하지 않는다.

K-의 의미 정의로 부터 메모리 관리에 대해 알 수 있는 사실들은:

- 새 메모리가 필요할 때 (let-식, 프로시져 호출, 레코드 식, malloc-식) 새로운 메모리는 항상 거기에 있다. 항상 필요한 메모리는 있어야 한다.

- 메모리 주소에 저장된 데이타가 없을 때, 그 주소를 접근하는 식은 의미가 없다. 예를 들어

  ```
  let x := malloc(1) in *x
  ```

  는 의미가 없다. *x 식의 의미를 정의할 수 없기 때문이다. 메모리 할당이 일어났어도 식 x가 계산하는 메모리 주소는 어떤 값도 아직 보관하고 있지는 않다. 의미정의에 따르면, 그 메모리 주소는 아직 메모리의 정의 구역에 없다. 따라서 식 *x의 의미인 그 주소의 메모리 이미지는 정의되지 않는다.

- 마지막으로, 할당된 메모리들은 서로에게서 서로에게 도달할 수 있는 방법이 없다. 할당받은 주소로 부터 몇 발자국을 움직여서 별도로 할당받은 다른 메모리에 도달할 수 있는 방법이 없다.

  이게 C와 다른 점이기도 하다. C 에서는 기준 주소에서 임의의 보폭으로 떨어진 메모리 주소를 접근하는 것이 가능하기 때문이다.

아뭏튼, 메모리 재활용을 프로그래머에게 미루는 것은 바람직한가? 어쩔 수 없는 선택인듯 하다. 언어가 이미 복잡해 질 대로 복잡해졌고, 메모리 재활용을 자동으로 할 수 있는 방안이 오리무중이므로.

그러나, 프로그래머가 신경쓰는 것은 많아졌다. 프로그램의 논리적인 흐름만이 아니다. 유한한 메모리를 가진 컴퓨터가 자신의 프로그램을 실행하게 된다는 조건을 항상 생각해야 한다. 그래서 적절히 재활용 명령문을 프로그램의 적절한 위치에 넣어줘야 한다. 소비한 각각의 메모리 주소마다.

좋은가? 프로그래머가 자신의 프로그램이 소모하는 메모리의 사용기간을 제일 잘 알 수 있는가? 그런가?

C 프로그래밍의 경험이 지난 20년 넘게 쌓이면서, 그 답은 "아니다"이다. 프로그래머가 가장 많이 하는 실수가 *메모리 재활용garbage collection* 오류이다. 메모리 관리를 프로그래머에게 맡기면서 C 프로그램의 가장 골치아픈 오류가 메모리 재활용이 되었다.

그 오류는 두 가지이다. 재활용 지점을 너무 늦게 잡는 오류와 너무 빨리 잡는 오류이다. 재활용이 너무 늦게 되면 자칫 컴퓨터의 메모리 자원이 고갈될 수 있다. 프로그램은 계속 메모리를 소모할 터이고 메모리 재활용이 제 때 되지 않는다면 필요한 메모리가 고갈 될 것이다. 재활용이 너무 빨리 되면 사용중인 메모리가 용도변경이 되버리는 것이다.

재활용이 너무 늦게 되는 오류를 *메모리 누수memory leak* 오류라고 한다. 너무 빨리 되는 오류를 *메모리 도난dangling pointer* 오류라고 한다.

메모리 관리를 프로그래머가 책임지도록하자 의 문제점

1. 메모리 출혈 memory leak

- 사용이 끝난 메모리를 너무 오래 잡고있다.
- 재활용이 너무 늦으면

memory
!
limit
time

2. 미아가 된 메모리 dangling pointer

- 사용이 끝나기 전에 너무 일찍 재활용 시키면

x
Y

```
free y ;
  :
x.next   /* was gone! */
```

4.4.3.2 자동 메모리 재활용

메모리 재활용을 프로그래머가 제대로 하기에는 너무 어렵다. 프로그래머에게 맡기면서 소프트웨어의 심각한 오류가 쉽게 나타났다. *메모리 누수memory leak*와 *메모리 도난dangling pointer*이다.

자동으로 메모리 재활용이 되었으면 한다. 프로그램 실행은 계속 메모리를 소모한다. 메모리 소모량이 어느 수위에 오르면, 실행을 잠시 멈춘다. 그리고 메모리를 재활용한다. 이제 컴퓨터의 메모리 자원이 풍부해졌다. 멈춘 프로그램의 실행을 재개한다. 가능할까? 이 과정을 또 다른 소프트웨어로 자동화하는 것이 가능할까?

프로그램을 멈추고, 그 프로그램이 사용했지만 앞으론 사용하지 않을 메모리를 찾아서 재활용해야 한다. 이것이 가능할까? 미래를 여행하는 타임머신을 만들 수 있을까?

그런 타임 머신(프로그램)은 만들 수 없다. 디지탈 컴퓨터의 한계이다. 좀 더 정확하게 이야기 하면, 프로그램의 미래에 사용되지 않을 메모리를 정확하게 골라내서 모두 재활용하는 프로그램은 만들 수 없다.

뭐가 어려운 걸까? "정확하게 모두"가 어렵다. 앞으로 사용않될 메모리를 모두 빠뜨림없이 재활용해야 하고, 그렇지 않은 것은 고스란히 남기고. 이게 불가능 하다.

그런 프로그램이 불가능하다는 것을 어떻게 확신하는가? 그런 프로그램이 있다면, 그 프로그램을 이용해서, *불가능**undecidable*하다고 증명된 *멈춰요 문제Halting Problem*를 푸는 프로그램을 짤 수 있기 때문이다. 불가능하다고 증명된 것을 가능하게 만들 수 가 있는가? 그럴 수는 없다.

*멈춰요 문제Halting Problem*를 푸는 프로그램이 불가능하다는 증명은 간단하다. 다음과 같다: 그러한 프로그램이 존재한다고 하자. 그러면 모순이다. 그러한 프로그램은 $H(p)$라고 하자. 프로그램 p를 받아서 p가 유한시간에 끝나는 프로그램이면 true, 그렇지 않으면 false를 낸다. 그 프로그램 H를 가지고 다음과 같은 모순된 함수 `f`를 정의할 수 있게된다:

```
function f() = if H(f) then (while true skip)
               else skip
```

`f()`는 끝나는가? 끝난다면 안끝나야 한다. 안끝난다면 끝나야 한다. 모순이다.

정확한 메모리 재활용이 불가능한 이유는, 미래에 사용않될 메모리를 정확하게 빠뜨리지않고 골라내는 프로그램 G가 있다면, *멈춰요 문제Halting Problem*를 해결하는 프로그램 $H(p)$를 다음과 같이 작성할 수 있게된다: 입력된 프로그램 p를 다음과 같이

```
let x := malloc() in p; x
```

고친다. `x`는 p에서 사용되지 않는 이름이라고 하자. 이렇게 고쳐진 플로그램을 실행시킨 후, `let`-몸통식의 p가 시작되기 직전에 G를 실행시킨다. 그 결과

에 x가 가지고 있는 메모리 주소가 포함되었다면, p는 끝나는 프로그램이고, 그렇지 않다면 p는 끝나지 않는 프로그램이다.

그러나 희망은 있다. 모두 정확히 재활용 해주는 프로그램은 불가능하지만, 조금이라도 재활용해 주는 프로그램은 만들 수 있다. 몇개는 빠뜨릴 수 있지만, 대부분은 재활용해 주는 프로그램을 만들 수 있다. *자동 메모리 재활용기garbage collector*의 원리는 이렇다:

- 원칙: 프로그램의 진행을 멈추고 나서, 지금까지 할당된 메모리중에서 미래에 사용할 수 있는 메모리를 제외하고 나머지는 모두 재활용해야 한다.

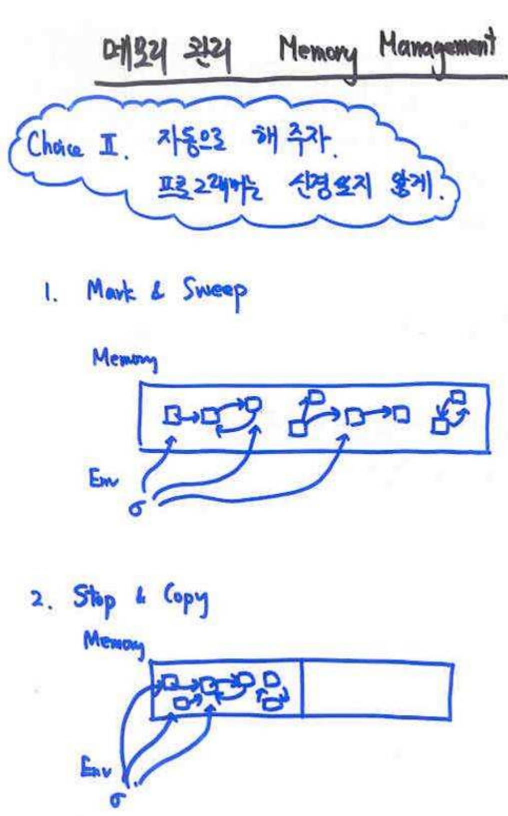
- 사실: 재활용을 완전하게(빠뜨리지 않고) 할 수 있는 방법은 없다. 있다면 그런 재활용기를 이용해서 *멈춰요 문제Halting Problem*를 풀 수 있기 때문이다.

- 양보: 그러나 재활용을 안전하게는(빠뜨리는 것은 있으나) 할 수 있는 방법은 있다. 뭔고하니,

⋆ 앞으로 실행할 식인 E만 있다고 하자. 식 E의 현재 *환경environment*으로부터 현재의 메모리를 통해 다다를 수 있는 모든 주소들은 앞으로 E를 실행중에 다시 사용될 수 있다. 이것들만 빼고 재활용하자. 즉, 그렇게 해서 다다를 수 없는 메모리 주소들은, 과거에 할당되어서 사용되었으나 앞으로의 E를 실행하는데는 사용되지 않을 것이 분명하므로, 재활용해도 된다.

모든 현대 언어들의 *메모리 재활용기garbage collector*는 위의 방식으로 구현되어 있다. (Java, ML, Scheme, Haskell, C#, Prolog, etc.)

메모리 재활용기 구현 알고리즘은 대표적으로 두가지가 있다:



하나의 딜레마가 있다. 메모리 재활용기들이 실행되는 데에도 메모리가 필요하다. 메모리 부족으로 재활용을 하는 것인데, 그러려면 메모리가 필요하다. 프로그램 실행중에 메모리에 만들어진 구조물들(*그래프graph*)을 모두 따라가면서 앞으로 사용될 수 있는 메모리 주소들을 모으게 된다. 그런데, 그래프를 누*비고 다니는 알고리즘depth-first-, breath-first-traversal algorithms*들은 메모리를 소모한다. 그래프의 가장 긴 경로 만큼의 스택*stack*이 필요하다.

메모리가 없어서 메모리 재활용기를 돌리는 것인데, 메모리가 필요하다니. 메모리를 소모하지 않는 *그래프를 누비는 알고리즘graph traversal algorithms*이 있어야 한다. 가능할까? 스택없이 가능한 알고리즘이 있다.

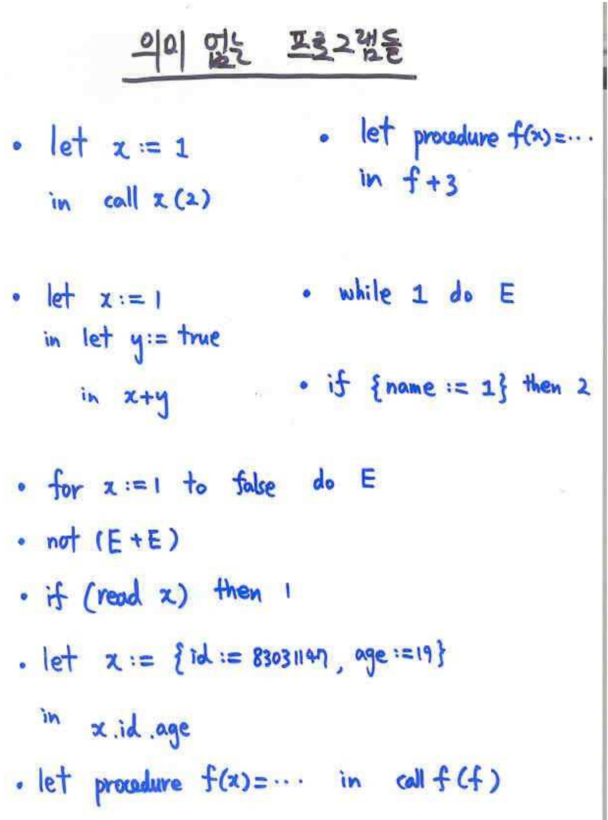
4.5 언어 키우기 III: K

4.5.1 실행되서는 않될 프로그램들

K-는 그럴듯 한 언어이지만, 현재 모든 프로그래밍 언어에 공통인 문제를 가지고 있다. 제대로 돌 수 없는 프로그램을 쉽게 만들 수 있다는 것이다. 프로그램이 문법은 틀린 곳이 없지만 오류로 실행될 수 없는 프로그램들.

프로그램 만드는 방법을 충실히 따라서 제대로 생긴 프로그램을 만들기는 쉽다. 그런데, 그 중에는 실행될 수 없는 프로그램들이 많다. 실행을 더 이상 진행할 수 없는 프로그램들이 많다.

레코드의 필드 이름들이 항상 제대로 접근되나. 함수 호출의 결과가 분별력있게 사용되는가. 함수의 인자값이 그 함수가 바라는 값들이 흘러드는가?



제대로 생긴(문법에 맞는) K-프로그램들 중에서 제대로 실행될 수 있는 프로그램은 일부분에 불과하다. “제대로 실행될 수 있는” 프로그램이란 우리의 의미규칙에 따라서 의미가 정의될 수 있는 프로그램이다.

4.5.2 오류 검증의 문제

주어진 K- 프로그램이 제대로 실행될 지를 미리 확인할 수 있는 기술이 없을까? 미리 검증해야만 한다. 프로그램 소스를 컴퓨터가 실행시키기 전에. 소프트웨어를 팔고 나서 오류를 수정해야할까? 로봇에 내장된 소프트웨어가 실행중에 오류로 무쇠팔을 갑자기 휘두르게 된다면? 소프트웨어가 자체로 상품인 경우는 말할 것도 없고, 모든 전자기계 제품들의 경쟁력중에서 내장된 소프트웨어의 신뢰도가 차지하는 비중은 적어도 제품 디자인 만큼은 될 것이다.

다른 분야는 우리보다 훨씬 앞서있다. 제대로 작동할 지를 미리 검증할 수 없는 기계설계는 없다. 제대로 서있을 지를 미리 검증할 수 없는 건축설계는 없다. 인공물들이 자연세계에서 문제 없이 작동할 지를 미리 엄밀하게 분석하는 수학적 기술들은 잘 발달해 왔다. 뉴튼 역학, 미적분 방정식, 통계 역학등등이 그러한 기술들일 것이다.

소프트웨어라는 인공물에 대해서는 어떠한가? 작성한 소프트웨어가 제대로 실행될 지를 미리 엄밀하게 확인해 주는 기술들은 있는가? 소프트웨어 오류를 소스만을 보고 미리 자동으로 찾아주는 기술들.

이 기술을 향한 연구들이 소프트웨어 기술의 발달과정의 중요 축이었다. 최종 목표는 소프트웨어가 오류없이 작동할 지를 실행전에 자동으로 검증하는 기술을 이룩하는 것이다. 소프트웨어의 오류를 자동으로 찾아주는 기술이다. 오류가 없다면 오류가 없다고 확인해 주는 기술이다. 소프트웨어가 생각대로(기획한대로) 구현되었는지, 그 소프트웨어가 작동하기 전에 엄밀하고 안전하게 자동으로 확인할 수 있는 기술.

이 축을 따라 프로그래밍 언어에서는 어떤 성과를 내었는가? 프로그래밍 언어가 어떻게 디자인되면 이 문제가 어디까지 해결될 수 있을까? 프로그래밍 언어 분야에서는 이 시급한 문제에 어떤 답을 내 놓고 있나? 그 수준은 현재 어느 정도 인가?

4.5.2.1 오류 검증 기술의 발전과정

프로그래밍 언어 분야를 돌이켜보면, 소프트웨어 오류를 자동으로 찾는 기술은 세 박자의 호흡만에 한 발 전진하는 형태였다. 첫 번째 호흡에서는 우리가 확인할 수 있는 오류의 한계를 정확하게 정의하고, 두 번째 호흡에서는 정의한 오류의 존재를 찾는 방법을 고안하며, 세 번째 호흡에서는 그 방법을 컴퓨터가 자동으로 실행할 수 있도록 소프트웨어로 만들어 낸다.

이때, 각 호흡마다 애매하고 허술한 구석이 없을 때에만 비로소 한 발짝의 전진이 있게 된다. 오류의 정의는 모호하지 않아야 하고, 고안된 방법은 프로그램이 그렇게 정의된 오류를 품고있다면 반드시 찾아낼 수 있어야 하고, 그 방법을 구현한 소프트웨어는 그 방법 그대로 충실히 구현되어야한다. “반드시”와 “충실히”라는 것은 엄밀한 증명과정을 거친다는 뜻이다.

이 세 박자 호흡을 반복하면서 확인할 수 있는 소프트웨어 오류의 정의는 기술 발전이 진행되면서 점점 정교해진다. 첫 세대에서는 비교적 쉬운 소프트웨어 오류들을 확인해주는 기술이 만들어지고, 두벗째 세대에서는 그보다 정

교한 오류들을 확인해주는 기술이 만들어지고, 세번째 세대에서는 더욱 더 정교한 오류들을 확인해주는 기술이 만들어지고... 이렇게 하면서 궁극적으로는, 소프트웨어가 생각대로 작동할지를 자동으로 확인해주는 소프트웨어 기술을 달성해가는 것이다.

이러한 과정을 밟으며 소프트웨어 기술이 전진한 횟수는 이제 겨우 두 발짝이다.

- **1세대 기술:** 문법 검증 기술.

  1970년대에 달성된 첫 발짝은, 생긴게 잘못된 프로그램을 자동으로 찾아내는 기술이다. 프로그램이 제대로 작동하려면 우선 프로그램의 생긴것이 제대로 되야한다. 이때 오류는 프로그램의 생김새가 틀린것이라고 정의된 경우고, 이러한 오류를 정확히 자동으로 찾아주는 기술이 달성되었다. 정확히 찾아준다는 뜻은, *안전하고sound* *빠뜨림없다는complete* 뜻이다. 멀쩡한 프로그램을 기형이라고 진단하는 경우(빠뜨리는 경우)도 없고, 기형인 프로그램을 멀쩡하다고 하는 경우(믿을 수 없는 경우)도 없다는 뜻이다.

  이 기술을 1세대 오류잡는 기술이라고하자. 이 기술을 문법검증기술(parsing)이라고 하고, 완전히 완성되어 오늘날의 프로그래머가 프로그램을 짜면서 늘상 아무렇지도 않게 사용할 만큼 관심밖으로 사라져 버린 성숙한 기술이다.

  그러나 이 오류잡는 기술이 겨우 1세대인 까닭은, 생긴건 멀쩡한데 생각대로 돌아가지 않는 소프트웨어에는 속수무책인 기술이기 때문이다. 겉모습의 오류 뿐이아닌, 속 내용(논리)의 오류 까지 자동으로 찾아주는 기술이 필요했다.

- **2세대 기술:** 타입 검증 기술.

  1990년대에 빛을 보기 시작한 두번째 기술은, *타입검증type checking*이라는 기술이다. 제 2세대 오류잡는 기술이라고 할 수 있다. 이때 오류의 정의는 겉모양이 틀린 프로그램만이 아니고, 속 내용이 잘못될 수 있는 경우

까지를 포함하게 된다. 여기서 잘못된 속 내용이란, 프로그램이 실행중에 잘못된 값이 잘못된 계산과정에 휩쓸리는 경우로 정의된다.

프로그램의 실행은 일종의 계산인데, 그 계산중에는 분별있는 일들이 일어난다. 더하기에는 숫자만 들어와야 한다던지, 결혼하기라는 계산에는 남자와 여자가 한쌍으로 있어야 한다던지, 수력발전하기에는 우라늄대신에 물이 있어야 한다던지.

이렇게 분별있게 값들이 소통되야 하는데, 그렇지 않은 경우를 타입에 맞지 않다고 한다. 이러한 경우가 발생하면 프로그램의 진행은 생각대로 흐르지못하고 급기야는 갑작스럽게 멈추고 만다. 휘발유가 공급되야 할 전투기의 엔진에 물이 새어들어 비행중에 추락해 버리듯이.

프로그램에서 타입에 맞지 않는 계산이 실행중에 발생할 지를 미리 검증하는 방법이 바로 타입검증이다. 이 성과는 프로그래밍 언어를 연구하는 분야에서 달성되었는데, 프로그램의 안전한 타입검증은 그 프로그래밍 언어가 제대로 디자인된 경우에만 가능하다. (대표적인 언어가 ML, OCaml, Haskell등이다.)

안전한 타입검증을 갖춘 프로그래밍 언어기술은 아직은 문법검증 기술만큼 모든 프로그래머들이 늘상 사용하는 기술로 널리 퍼지지는 않았다. 그러나 그런 타입 검증 기술은 앞으로 모든 소프트웨어 개발자가 당연한 기술이라고 생각하면서 사용하게 될 날이 멀지 않다.

4.5.3 타입 시스템: 소개

타입 시스템은 말이 되는 프로그램을 판단하는 한 방법이다. 말이 되는 프로그램을 판단하는 방법들은 점점 정교해져가고 있는데, 타입 시스템은 프로그램이 말이 되는 지를 판단하는 두번째로 정교한 방법이라고 할 수 있다.

첫번째 방법은 문법규칙이다. 제대로 생긴 프로그램인 지를 확인하는 방법. 문법규칙은 간단한 귀납규칙들이다: 부품이 제대로 생긴 프로그램이기만 하면 그 것들을 가지고 어떻게 어떻게하면 제대로 생긴 프로그램이 된다, 는 식

이다. “간단한” 이유는 조건이 간단해서다. 부품이 제대로 생긴 프로그램이기만 하면 된다. 이렇게 조건이 간단하다보니, 말이 되지 않는 프로그램을 모두 걸러내지 못하는 경우가 많다.

Example 23 프로그램식 E의 문법규칙이 다음과 같다고 하자:

$$
\begin{array}{rcll}
E & \rightarrow & n & \text{정수} \\
 & | & s & \text{스트링} \\
 & | & E + E &
\end{array}
$$

이때 2+‘‘a’’는 문법규칙에 맞는다. 2와 ‘‘a’’는 제대로 생긴 것 들이고, 그것들을 가지고 +식을 만들었으므로. 하지만 말이 되지 않는 프로그램이다. 정수와 스트링은 더할 수 없다. □

대게의 문법규칙들이 요구하는 조건은 너무 간단하다. 말이되는 프로그램을 판단하기에는. 문맥을 생각하지 않는 규칙들이기 때문이다. 고려해야 할 문맥은 이런 것이다: +식은 두개의 부품식들이 모두 정수식이어야 한다는. 이러한 문맥에 대한 조건을 대게의 “문법규칙”에는 표현하지 않는다. 표현하고 정교하게 다듬으면 그것은 우리가 “문법규칙”의 영역에서 “타입 시스템”의 영역으로 응근슬쩍 넘어가게 되는 셈이다.

“타입 시스템”에서는 어떤 문맥을 고려하는가? 프로그램의 겉모습만을 따지는 문맥이 아니라, 한 단계 더 파고 들어서, 텍스트가 가지는 속내용을 문맥으로 고려해야 한다. 타입 시스템은 타입을 문맥으로 고려한다. 예를들어, “x+1”이 말이 되는 프로그램이려면 변수 x는 정수값을 가진 변수여야 한다. 타입 시스템은 x가 정수값을 가지는 상황에서만 말이된다고 판단한다.

타입 시스템은 타입에 맞는 값들이 분별있게 계산되는 프로그램인지를 판단해 준다. 프로그램의 실행은 일종의 계산이고 그 계산중에 값들은 분별있게 흘러다닌다. 더하기 계산에는 숫자들이, 결혼이라는 계산에는 남자와 여자가, 라면끓이기 계산에는 라면과 물과 불이, 수력발전 계산에는 물과 터빈이, 등등. 숫자더하기에 남자와 여자가 들어가면 안되고, 수력발전에 불과 자동차가 들어가면 안된다.

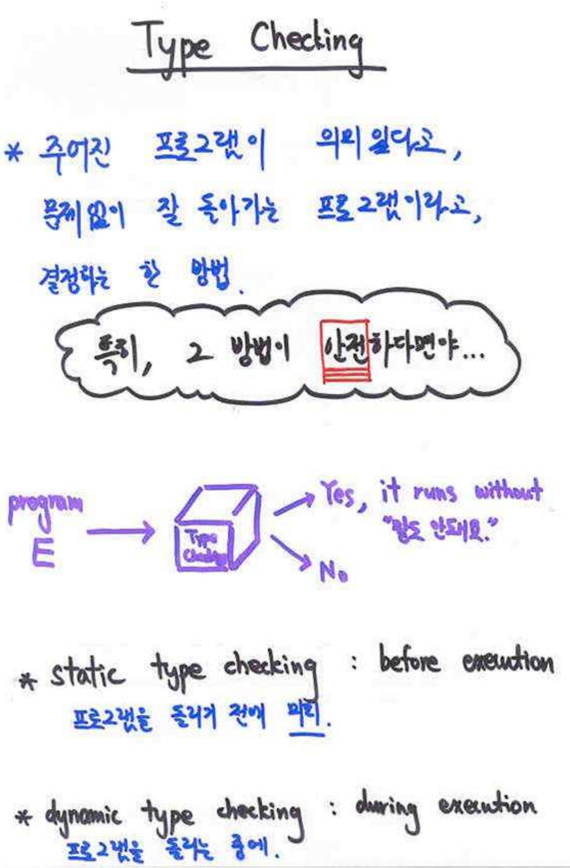
4.5.4 K = K- + 타입 시스템

K- 언어에도 *타입 시스템type system*을 통해서 논리적인 오류를 미리 자동으로 찾을 수 있도록 해보자. *타입 검증type checking*이 가능하도록.

우선 *동적 타입 검증dynamic type checking*을 알아보자. K-의 의미정의대로 실행하는 *실행기interpreter*를 구현했다면, 실행중에 타입을 확인해가는 과정이 필요하게된다. 덧셈 식 E_1 + E_2 의 두 식들은 정수값을 만들어내는 식들인 경우에만 더하기가 진행될 수 있다. 실행결과가 정수인지를 확인한 후에 덧셈이 진행되야 한다. 프로시져 호출 식 $f\ (E)$ 은 f의 실체가 프로시져인 지를 실행중에 확인한 후에나 호출이 일어날 수 있다. 실체가 메모리 주소인 보통의 변수라면 호출이 일어날 수 가 없다. 이렇게 실행중에 값의 종류가 예상대로 흘러다니는 지를 확인하는 것을 *동적 타입 검증dynamic type checking*이라고 한다.

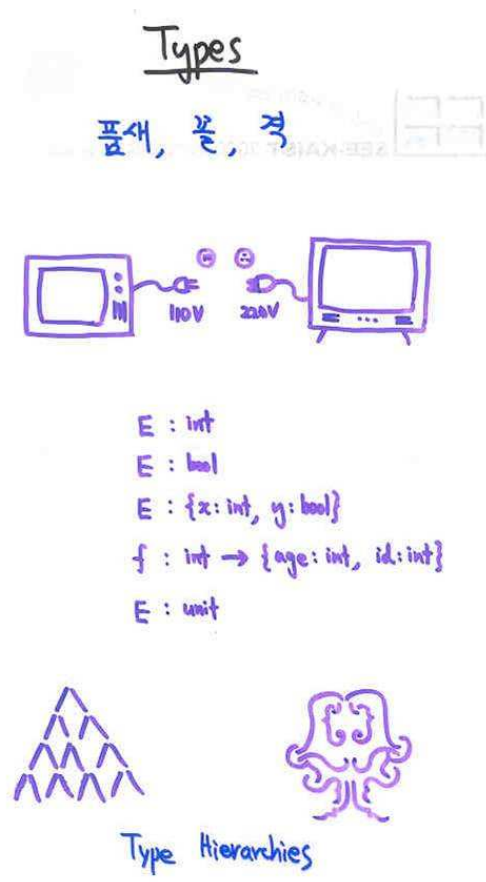
그러나, 위에서 이야기한대로, 우리는 *정적 타입 검증static type checking*이 필요하다. 프로그램 실행전에 타입 검증하기. 실행하다가 문제가 발생하는 것보다는 문제를 미리 찾아내고 무 결점임을 확인한 후에 실행을 시작하는 것이 필요하다. 이런 기술을 *정적 타입 검증static type checking*이라고 한다.

검증기가 프로그램을 분석해서 "실행중에 타입 오류가 없을 것임"이라고 했을 때에만 프로그램을 실행시키고 싶다. 더군다나 그 검증기는 *믿을만sound*하면 좋겠다. 타입 검증 결과 문제 없을 것이라고 했는데 실행중에 문제가 발생하는 경우가 없었으면 좋겠다.



미리 이야기 하지만, 우리 K- 에서는 *믿을만sound*한 타입 검증기를 만들어 내지는 못한다. K-는 *안전한 타입 검증기sound type checker*를 갖춘 언어를 다자인하자는 목표를 처음부터 가지고 디자인 된 언어는 아니다. 현재 대부분의 인기있는 프로그래밍 언어의 모습이기도 하다.

그러한 언어들의 타입 시스템이 어디까지 와 있는 지를 K-를 통해서 살펴보자. 현재 대부분의 프로그래밍 언어의 모습이 어떻게 해서 그 모습이 되었는지.



4.5.4.1 K-에서 계산되는 값들의 종류

위에서 이야기한 K-에서 프로그램 식들이 계산하는 값들의 종류는 5가지가 있다:

$$
\begin{aligned}
\mathit{Val} &= \mathbb{Z} + \mathbb{B} + \{\cdot\} + \mathit{Record} + \mathit{Addr} \\
\mathit{Record} &= \mathit{Id} \xrightarrow{\text{fin}} \mathit{Addr}
\end{aligned}
$$

*기본 값primitive value*의 종류마다 이름을 붙이자. 이것들이 *기본 타입primitive type*들이다. 그리고 레코드는 *복합 값compound value*이고, 그 타입은 레코드 부품들의 타입으로 만들어진 *복합 타입compound type*이 된다. 메모리 주소는 값을 보관하고 있고, 메모리 주소의 타입은 보관하고 있는 값의 타입에 따라 분류한 *복합 타입compound type*이다. 프로시져는 K-에서 값은 아니지만, 그 타입을 가지고 호출식들이 제대로 쓰이는 지 확인할 필요가 있다. 인자 타입과 결과 타

입으로 구성된 복합 타입*compound type*이 된다.

$$
\begin{array}{llcll}
\mathit{Type} & \tau & \rightarrow & \mathit{int} \mid \mathit{bool} \mid \mathit{unit} & \\
 & & \mid & \tau\ \mathit{loc} & \text{location type} \\
 & & \mid & \{\} \mid \{x \rightarrow \tau, y \rightarrow \tau\} & \text{record type} \\
 & & \mid & \tau \rightarrow \tau & \text{procedure type}
\end{array}
$$

4.5.4.2 타입 검증 규칙

K- 타입 시스템은 다음의 판단*typing judgment*

$$\Gamma \vdash E : \tau$$

을 증명하는 논리 규칙들이다. 프로그램 의미 정의 규칙과 동일한 프레임웍이다. 타입 환경*type environment*

$$\Gamma \in \mathit{Id} \xrightarrow{\text{fin}} \mathit{Type}$$

이라고 불리는 Γ는 프로그램에 나타나는 이름들의 타입을 가진 테이블이다. 타입 판단

$$\Gamma \vdash E : \tau$$

은

> 타입환경 Γ에서 식 E가 제대로 실행되고 만일 끝난다면 타입 τ의 값을 계산한다,

는 사실을 뜻한다. 규칙들을 하나 하나 보자. 각 식의 구조에 따라 하나씩 주어져있다.

$\boxed{\Gamma \vdash E : \tau}$

$$\frac{}{\Gamma \vdash \texttt{skip} : \mathit{unit}}$$

$$\frac{\Gamma \vdash E : \tau \quad \Gamma(x) = \tau\ \mathit{loc}}{\Gamma \vdash x \texttt{ := } E : \tau}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash E_1 \texttt{ ; } E_2 : \tau_2}$$

$$\frac{\Gamma \vdash E : bool \quad \Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 : \tau}$$

$$\frac{\Gamma \vdash E_1 : bool \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash \texttt{while } E_1 \texttt{ do } E_2 : unit}$$

$$\frac{\Gamma \vdash E_1 : int \quad \Gamma \vdash E_2 : int \qquad \Gamma\{x \mapsto int\ loc\} \vdash E : \tau}{\Gamma \vdash \texttt{for } x \texttt{ := } E_1 \texttt{ to } E_2 \texttt{ do } E : unit}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma\{x \mapsto \tau_1\} \vdash E_2 : \tau}{\Gamma \vdash \texttt{let } x \texttt{ := } E_1 \texttt{ in } E_2 : \tau}$$

$$\frac{\Gamma\{f \mapsto \tau_1 \to \tau_2\}\{x \mapsto \tau_1\} \vdash E_1 : \tau_2 \quad \Gamma\{f \mapsto \tau_1 \to \tau_2\} \vdash E : \tau}{\Gamma \vdash \texttt{let proc } f\texttt{(}x\texttt{) = } E_1 \texttt{ in } E : \tau}$$

$$\frac{\Gamma(f) = \tau_1 \to \tau_2 \quad \Gamma \vdash E : \tau_1}{\Gamma \vdash f \texttt{ (}E\texttt{)} : \tau_2}$$

$$\frac{\Gamma(x) = int\ loc}{\Gamma \vdash \texttt{read } x : int}$$

$$\frac{\Gamma \vdash E : \tau}{\Gamma \vdash \texttt{write } E : \tau}$$

$$\frac{}{\Gamma \vdash n : int}$$

$$\frac{\Gamma(x)\tau\ loc}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash E_1 : int \quad \Gamma \vdash E_2 : int}{\Gamma \vdash E_1 \texttt{ + } E_2 : int}$$

$$\frac{\Gamma \vdash E : int}{\Gamma \vdash \texttt{- } E : int}$$

$$\frac{\Gamma \vdash E_1 : int \quad \Gamma \vdash E_2 : int}{\Gamma \vdash E_1 \texttt{ < } E_2 : bool}$$

$$\frac{\Gamma \vdash E : bool}{\Gamma \vdash \texttt{not } E : bool}$$

$$\frac{}{\Gamma \vdash \texttt{\{\}} : \{\}}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash \{x \texttt{ := } E_1, y \texttt{ := } E_2\} : \{x \mapsto \tau_1\ loc, y \mapsto \tau_2\ loc\}}$$

$$\frac{\Gamma \vdash E : \tau \quad \tau(x) = \tau'\ loc}{\Gamma \vdash E.x : \tau'}$$

$$\frac{\Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \tau' \quad \tau(x) = \tau'\ loc}{\Gamma \vdash E_1.x \texttt{ := } E_2 : \tau'}$$

$$\frac{\Gamma \vdash E : \tau}{\Gamma \vdash \texttt{malloc}\, E : \tau\ loc}$$

$$\frac{}{\Gamma \vdash \texttt{\&}x : \Gamma(x)}$$

$$\frac{\Gamma \vdash E : \tau}{\Gamma \vdash \texttt{\&}E.x : \tau(x)}$$

$$\frac{\Gamma \vdash E : \tau\ loc}{\Gamma \vdash \texttt{*}E : \tau}$$

$$\frac{\Gamma \vdash E_1 : \tau\ loc \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash \texttt{*}E_1 \texttt{ := } E_2 : \tau}$$

위의 타입 시스템을 곰씹어 보자. 구현을 어떻게 할 지는 다음 절에서 살필것이다. 순수한 논리 시스템으로서 바라보자. 프로그램의 구석구석이 어떤 타입을 갖는다고 판단 될 수 있는지.

위의 타입 시스템은 한 가지 재미 있는 사실이 있다. 프로그래머가 타입에 대해서 프로그램 텍스트안에 써 넣은 것이 전혀 없다.

4.5.5 K- 타입 시스템의 논리적 문제

그럴듯 하지만 아쉬운 면이 있다. 두가지 문제가 있다. *안전sound*하지도 않고 *완전complete*하지도 않다.

4.5.5.1 타입 검증 규칙이 안전하지않다

위의 타입 시스템은 *믿을만 하지 않다unsound.* 위의 타입 시스템이 문제없다고 판단한 프로그램이 실행중에 타입 오류로 멈출 수 있다. 오류 검증이 믿을 수 없는 것이다.

```
* type 리스트 = {int x, 리스트 next}
  let
    리스트 node = {x:=1, next:={}}
  in
    node.next.x
  end

  type-checked!. but cannot run :(

What would you do to make the
type system safe ?
  - what did we miss ?
```

이 문제는 지금까지 “상식의 수준에서” 키워 온 언어인 K-에서는 해결할 수 없다. 언어를 디자인 할 때 부터, 이 문제를 염두에 두고 잘 디자인해 가야 해결할 수 있는 문제이다. 5장에서 이 과정을 밟아 갈 것이다. 어떻게 범용의 언어가 이 문제를 해결한 편리한 타입 시스템을 가질 수 있는지 겪게 될 것이다.

이 타입 시스템이 안전하지 않은 경우는 다음과 같다:

```
* type 리스트 = {int x, 리스트 next}
  let
    리스트 node = {x:=1, next:={}}
  in
    node.next.x
  end
```

type-checked! but cannot run :(

What would you do to make the type system safe?

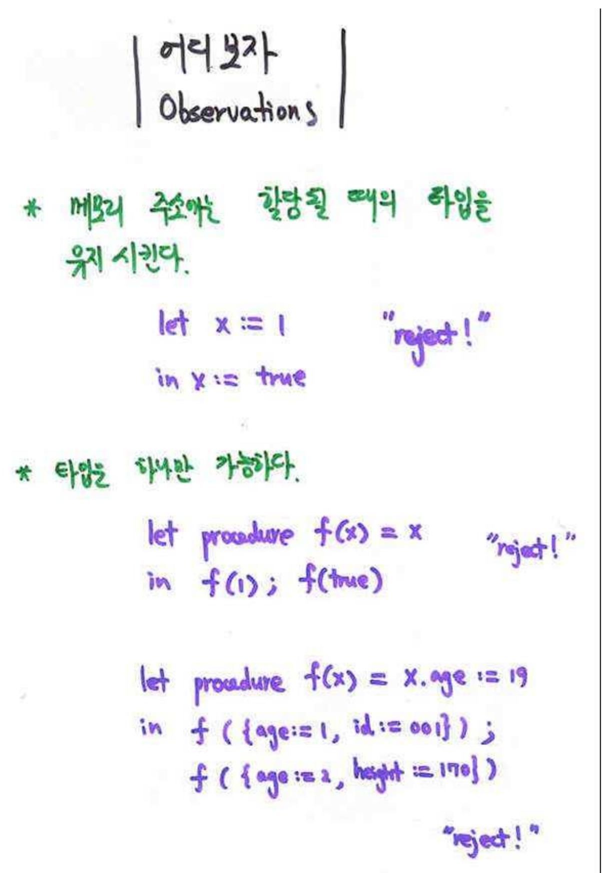
- what did we miss?

타입 시스템을 안전하게 만들어 갈 수는 있다. 그렇게 되면 적어도 세가지로 결론나게 될 것이다. 안전하게 되기 위해서 K-를 주무르게 된다. 가지치고. 그렇게 되면 더이상 K-라는 언어의 모습이 아닐 것이다. 비슷한 이야긴데, 다른 한 결론은, 안전한 타입 시스템을 고안하고 구현할 수 있게되면 그 타입 시스템을 통과하는 K- 프로그램은 매우 소수가 될 것이다. 그러한 K- 프로그램은 K- 언어의 제한된 기능만 가지고 작성된 프로그램들이 될 것이다. 또 다른 결론은, 안전하게 되기 위해서 매우 정교한 타입 시스템을 고안하게 될 것이고, 그 정교함이 극에 달해서 아마도 지금의 기계로는 구현될 수 없는 것이 될 것이다.

다음 5장에서 타입 시스템의 안전성을 염두에 두고 프로그래밍 언어를 디자인해 가면서 달성한 성과를 살펴볼 것이다. 어떻게 실용적인 범용의 언어가 이 문제를 해결한 편리한 타입 시스템을 갖추게 되었는 지 알아보자. 프로그래밍 언어 분야의 빛나는 성과이다.

4.5.5.2 타입 검증 규칙이 완전하지않다

K- 타입 규칙의 두번째 문제는, 제대로 도는 프로그램중에서 위의 타입 시스템을 통과하는 것은 일부분*incomplete*이다. 잘 돌 프로그램을 모두 타입 검증할 수 있는게 아니다. 제대로 타입 오류 없이 잘 돌 프로그램을, 위의 타입 시스템은 타입이 없다고 판단할 수 있다. 타입 시스템이 사용하는 그물이 너무 촘촘한 것이다.



- 메모리 주소의 타입이 변할 수 가 없다. 메모리 주소에 처음으로 보관되는 초기 값의 타입이 그 주소가 저장할 수 있는 값들의 타입으로 고정된다. 변수는 항상 그 초기 값과 같은 타입의 값만 보관할 수 있는 것이다. 그렇지 않은 프로그램은 타입 검증이 실패한다.

  이 문제를 해결할 수 있는 좋은 방안이 뭘까?

- 어떤 프로시저는 인자 값의 타입에 상관없이 실행될 수 있는 프로시저 들이 있다. 이러한 프로시저는 다양한 타입의 인자를 받으면서 아무 문제 실행될 수 있다. 이러한 프로시저는 위의 타입 시스템에서는 타입 검증을 통과할 수 없다.

  이 문제를 해결할 수 있는 좋은 방안이 뭘까?

- 레코드 타입 관련해서도 타입 검증이 너무 까다롭다. 다음의 문제가 있다.

  복합적인 데이타를 표현하는 데 쓰이는 레코드의 타입이 유연하지 않다. 복합적인 데이타는 주로 재귀적으로 정의되는 데이타들이다(예, 리스트나 나무구조). 리스트가 리스트를 품고 있을 것이고, 나무가 나무를 품고 있을 것이다. 이런 구조를 레코드로 구현하다보면, C 에서 처럼, 레코드의 한 필드가 그 재귀를 품는 고리가 된다(예, 리스트 노드의 "`next`" 필드나 나무구조 노드의 "`sub-tree`" 필드). 문제는, 그 필드의 타입이 하나의 레코드 타입으로 정해져서는 않된다. 그 필드는 초기에는 빈 레코드를 가질 것이다. 재귀적으로 품고 있는 것이 없는 것이다. 재귀의 깊이가 0인 레코드이다. 그러다가 실행중에 그 빈 레코드 필드가 다른 레코드를 품게 될 것이다. 재귀 구조를 품게 되는 것이다. 예를들어,

  ```
  lst := {val := 0, next := {}};
  lst.next := {val := 100, next := {}}
  ```

  위의 `lst`의 `next` 필드의 타입은 빈 레코드 타입으로 고정될 수 없다. 초기에 빈 레코드를 가지게 되었지만 곧 이어 그 필드가 다른 타입의 레코드를 가지게 되므로. 그렇다고 두번째 지정문 때문에 레코드 타입 $\{val \mapsto int, next \mapsto \{\}\}$ 으로도 고정될 수 없다. 초기의 빈 레코드와 타입이 맞지 않을 뿐 더러 `lst.next.next`를 통해서 또 다른 레코드가 빈 필드를 바꿔칠 수 있기 때문이다.

  이 문제를 해결할 수 있는 좋은 방안이 뭘까?

타입 시스템의 *안전성**soundness*을 개의치 않는다면, 위의 문제를 해결하는 방법은 쉽게 내어놓을 수 있다. 사실 이렇게 고안된 방법들이 현재의 대부분의 프로그래밍 언어에 적용되었다. 그 흔적을 따라가 볼 것이다.

세번째 문제를 다음과 같이 해결하자. 우선, 프로그래머가 타입을 정의하고 이름을 지을 수 있도록 하자. 그리고 그 이름이 타입 정의에서 재귀적으로 사용될 수 있도록 하자. 그러면 재귀 데이타를 구현하는 레코드에서 고리역할을 하는 필드의 타입에 이름을 사용하면, 그 필드가 품고 있을 재귀의 깊이를 타입에 드러내지 않게 된다.

예를 들어

```
type list = {int val, list next}
```

라고 정의된 `list` 타입은 레코드를 뜻한다, 정수를 가지는 `val` 필드와 재귀적으로 `list`를 품을 수 있는 `next` 필드. 재귀 고리에 보관되는 리스트의 깊이는 표현될 필요가 없다.

프로그램에서 쓰는 타입과 타입 정의는 다음과 같은 문법을 따르도록 하자.

$$
\begin{array}{rcll}
t & \to & \texttt{int} \mid \texttt{bool} \mid \texttt{unit} & \\
& \mid & t\,\texttt{loc} & \\
& \mid & tid & \text{type name} \\
T & \to & \texttt{type}\ tid\ \texttt{=}\ \texttt{\{}t\,x, t\,y\texttt{\}} & \text{type declaration} \\
P & \to & T^*\,E & \text{program}
\end{array}
$$

타입 이름들의 유효범위는 프로그램 전체가 되도록 했다. 타입 이름들이 정의되는 곳은 프로그램의 처음 부분 뿐이다. 타입 이름의 유효범위가 다른 이름들 같이 일 부분으로 제한되게 하는 것도 가능하겠지만.

그리고, 빈 레코드 식은 임의의 타입으로 결정될 수 있다고 타입 규칙을 정의한다.

$$
\overline{\Gamma \vdash \texttt{\{\}} : \{\}}
$$

대신에

$$\overline{\Gamma \vdash \{\} : \tau}$$

로. 빈 레코드를 초기에 가지게 되는 list 레코드의 next필드는 list타입이 되는 것으로 판단되는 것이다.

이렇게 되면, 위에서 살펴본 프로그램

```
lst := {val := 0, next := {}};
lst.next := {val := 100, next := {}}
```

에서 lst의 타입은 *list loc*으로 판단할 수 있고, 두 번째 줄의 지정문도 타입 검증을 통과한다. lst.next도 *list loc*이므로.

위의 해결 책으로 부터 배우는 중요한 아이디어는 이름을 사용한다는 것이다. 타입에 이름을 붙이고 그 이름으로 속 내용을 감춘다. 재귀적인 구조를 구현하는 레코드는 재귀 고리 역할을 하는 필드가 있다. 임의의 깊이로 레코드를 품을 수 있는 그 필드의 타입. 변화무쌍한 재귀의 깊이를 타입에 표현하지 않기 위해서 이름을 사용하는 것이다.

그러나, 위의 해결책이 어설픈 이유는, 타입 시스템이 안전하지 않은 문제가 해결되지 않았기 때문이다. 타입 시스템의 *안전성soundness*을 포기하고 이렇게 고안된 방법들이 현재의 대부분의 프로그래밍 언어에 적용되었다.

5장에서는 제대로 된 해결책을 논의할 것이다. 타입 시스템의 *안전성soundness*을 보장하면서 위의 문제들을 해결해 간 결과들. 프로그래밍 언어의 타입 시스템 연구의 핵심을 보게 될 것이다. 안전하면서 최대한 많이 타입 오류 없이 실행될 프로그램들을 검증해 주는 시스템을 만들고자 한 노력의 결과들. 이 과정을 살펴볼 것이다.

4.5.6 K- 타입 시스템의 구현 문제

한편, 위와 같이 고안한 K- 타입 시스템을 어떻게 구현할 것인가? 구현한 타입 시스템은 K-프로그램을 받아서 예/아니오를 출력한다. 정의한 증명 규칙

에 따라서 그 프로그램의 구석 구석 타입이 제대로 되었는 지 증명해 낼 수 있으면 "예", 아니면 "아니오."

그러한 증명기를 어떻게 구현할 수 있을까? K- 타입 규칙들은 의미 규칙과는 다른 양상이 있다. 타입 검증기 구현은 K- 의미 규칙을 보고 *실행기interpreter*를 구현할 때와 같이 간단하지는 않다.

구현의 문제를 간편하게 넘어가는 방법은, 프로그래머가 타입 검증을 돕는 코멘트를 프로그램 택스트에 포함시키도록 강제하는 것이다. 이와 같은 방식으로 구성된 언어가 C, C++, Pascal, Java등의 언어들이다. 이 과정을 그대로 K-에서 밟아갈 것이다.

K- 타입 시스템의 구현이 간단치 않은 이유를 살펴보자. 우선, 구현 전략을 생각해 보자. 타입 판단

$$\Gamma \vdash E : \tau$$

의 증명을 구현하는 타입검증 함수의 입력은 두 가지가 될것이다: 프로그램 식 E 와 Γ. 주어진 두 개의 입력에 대해서 타입검증 함수는 위의 타입판단에 해당하는 τ가 있는 지를 찾게 된다. 타입환경 Γ는 프로그램에 나타나는 이름들의 타입들에 대해서 알려진(가정하게 된) 내용이다. 이 것을 가지고 프로그램 텍스트 E가 타입 τ를 가질 수 있는 지 타입 규칙에 준해서 알아보는 것이다. 프로그램 전체에 대해서는 시작 Γ는 물론 비어있을 것이다. 타입검증 함수가 프로그램의 부분들에 대해서 재귀적으로 호출되면서 Γ에 정보가 쌓이게 될 것이다.

우리의 타입 시스템에서 구현이 어려운 이유는 다음의 규칙에서 드러난다:

$$\frac{\Gamma\{f \mapsto \tau_1 \to \tau_2\}\{x \mapsto \tau_1\} \vdash E_1 : \tau_2 \quad \Gamma\{f \mapsto \tau_1 \to \tau_2\} \vdash E : \tau}{\Gamma \vdash \texttt{let proc } f(x) = E_1 \texttt{ in } E : \tau}$$

결론(분모)이 가능하려면 조건(분자 부분에 있는) 판단들이 사실이어야 한다. 즉, 프로시져 몸통 E_1이 타입 τ_2를 가진다는 것이 확인되야 한다. 조건 판단들이 사실인지를 확인하는 것은, 다시 타입검증 함수를 재귀 호출하는 것이다.

이 때, 타입검증 함수에 입력으로 줄 타입환경

$$\Gamma\{f \mapsto \tau_1 \to \tau_2\}\{x \mapsto \tau_1\}$$

은 이미 우리가 알아내야 할 τ_1 과 τ_2가 쓰인다. 그 두 타입들이 프로시저의 타입을 구성하기 때문이다. 당황스럽다. 출력으로 알아내야 할 것을 입력으로 넣어줘야 한다? 가능하지 않다.

어떻게 해야, 위와 같이 서로 물고 무는 상황을 타입검증 함수에서 구현할 수 있을까? 프그래머에게 책임을 맡기자. 프로그래머가 타입을 명시하도록 하자. 타입 검증은 그 힌트에 기초해서 하자. 프로시져 정의에서 프로그래머가 인자의 타입과 결과 타입을 명시하도록 하고, 타입 검증은 그것을 가정한 상태에서 프로시져 몸통을 타입 검증한다. 이렇게:

$$\frac{\Gamma\{f \mapsto \tau_1 \to \tau_2\}\{x \mapsto \tau_1\} \vdash E_1 : \tau_2 \quad \Gamma\{f \mapsto \tau_1 \to \tau_2\} \vdash E : \tau}{\Gamma \vdash \texttt{let proc}\ f\texttt{(}\tau_1\ x\texttt{)}\texttt{:}\tau_2 \texttt{=} E_1\ \texttt{in}\ E : \tau}$$

이렇게 되면 구현은 쉽다. K- *실행기interpreter*의 구현과 같이 간편하다. 타입 검증 규칙에 따라서 재귀호출을 부르면 된다. 프로시져 결과의 타입을 구성하는 τ_1과 τ_2는 프로그램 텍스트에서 프로그래머가 명시해 놓은 것을 쓰면 된다.

이제 지금까지 확장되어 온 K- 언어를 살짜 확장해서, 타입을 프로그래머가 프로그램 텍스트에 포함시키도록 하자. 프로시져와 변수 선언에서 그 변수

의 타입을 프로그래머가 명시하도록 한다. K 라는 언어라고 하자.

$$
\begin{array}{rcll}
E & \rightarrow & \vdots & \\
 & | & \texttt{let}\ t\ x\ \texttt{:=}\ E\ \texttt{in}\ E & \\
 & | & \texttt{let proc}\ f\,(t\ x)\,\texttt{:}\,t\ \texttt{=}\ E\ \texttt{in}\ E & \\
t & \rightarrow & \texttt{int}\ |\ \texttt{bool}\ |\ \texttt{unit} & \\
 & | & t\ \texttt{loc} & \\
 & | & tid & \text{type name} \\
T & \rightarrow & \texttt{type}\ tid\ \texttt{=}\ \{t\ x, t\ y\} & \text{type declaration} \\
P & \rightarrow & T^*\ E & \text{program}
\end{array}
$$

이제 다음과 같이 프로그램하고 타입 검증이 통과될 것이다.

```
e.g.)  type bbs = { int name, bool zap}
       let bbs x := { name := 0, zap := true}
       in if x.zap then 1
                   else x.name

e.g.)  type intlist = { int x, intlist next}
       let intlist l := { x:=0, next := {}}
       in
          l.next := { x:=1, next := {}} ;
          l.next.next := {x:=2, next :={}}

e.g.)  type intree = { intree l, int x, intree r}
       let procedure shake ( intree t) : intree
         = if t = {} then t
           else let intree t' := {}
                in t' := call shake (t.l);
                   t.l := call shake(t.r);
                   t.r := t'
       in call shake({ l:={}, x:=1, r:={ l:={}, x:=2, r:={}}
          })
```

아뭏튼 위와 같이 확장된 K 언어의 타입 검증 규칙은 다음과 같다. *타입 환경*type environment은 프로그램에 나타나는 이름들(변수, 프로시져 이름, 레코드

필드)의 타입에 관한 것이었으나, 이제부터는 타입 이름들의 속 내용(타입)에 대한 환경도 필요해 진다.

$$\begin{array}{lrcll}
 & \Gamma & \in & Id \stackrel{\text{fin}}{\to} Type & \text{변수의 타입환경} \\
 & \Delta & \in & TypeId \stackrel{\text{fin}}{\to} Type & \text{타입이름의 타입환경} \\
Type & \tau & \to & int \mid bool \mid unit & \\
 & & \mid & \tau\ loc & \text{location type} \\
 & & \mid & tid \mid \{\} \mid \{x \to \tau, y \to \tau\} & \text{record type} \\
 & & \mid & \tau \to \tau & \text{procedure type} \\
TypeId & tid & & &
\end{array}$$

$\boxed{\Gamma, \Delta \vdash P : \tau}$

$$\frac{\emptyset \vdash T^* : \Delta \quad \emptyset, \Delta \vdash E : \tau}{\emptyset, \emptyset \vdash T^*\ E : \tau}$$

$\boxed{\Delta \vdash T^* : \Delta'}$

$$\frac{\Delta \vdash T_1 : \Delta_1 \quad \Delta_1 \vdash T_2 : \Delta_2}{\Delta \vdash T_1\ T_2 : \Delta_2}$$

$$\frac{}{\Delta \vdash \texttt{type}\ tid\ \texttt{=}\ \texttt{\{}t_1\ x_1, t_2\ x_2\texttt{\}} : \Delta\{tid \mapsto \{x_1 \mapsto t_1\ loc, x_2 \mapsto t_2\ loc\}\}}\ x_1 \neq x_2$$

$\boxed{\Gamma, \Delta \vdash E : \tau}$

$$\frac{\Gamma, \Delta \vdash E : \tau \quad \Delta(tid)\{x \mapsto \tau\ loc\}}{\Gamma, \Delta \vdash \texttt{\{}x\ \texttt{:=}\ E\texttt{\}} : tid}$$

$$\frac{}{\Gamma, \Delta \vdash \texttt{\{\}} : \tau}$$

$$\frac{\Gamma\{f \mapsto \tau_1 \to \tau_2\}\{x \mapsto \tau_1\}, \Delta \vdash E_1 : \tau_2 \quad \Gamma\{f \mapsto \tau_1 \to \tau_2\}, \Delta \vdash E : \tau}{\Gamma, \Delta \vdash \texttt{let proc}\ f\texttt{(}\tau_1\ x\texttt{)}\texttt{:}\tau_2\ \texttt{=}\ E_1\ \texttt{in}\ E : \tau}$$

어디보자 II
Observations

```
* let x := {a:=1, b:=2} in x end

* type 리스트 = {int x, 리스트 next}
  type 트리스 = {int x, 트리스 prev}
  let
      ... {x:=1, next:={}}
          {x:=1, prev:={}}
  ...

* type t1 = {int x, bool y}
  type t2 = {bool y, int x}
  let ... {x:=1, y:=true}
  ...

* type 장군 = {int x, 멍군 y}
  type 멍군 = {bool x, 장군 z}
  let 장군 x:={x:=1, y:={}}
      멍군 x':={x:=true, z:={}}
  in
      x.y := x';
      x'.z := x
  end
```

4.5.7 이름 공간, 같은 타입

프로그래밍 언어의 두가지 이슈를 논의하고 이 장을 마치자. *이름 공간name space*과 *같은 타입type equivalence*의 정의에 대해서.

우선 이름공간에 대해서. K 언어에서는 많은 것에 이름을 지을 수 있다. 메모리 주소(변수), 레코드의 필드, 프로그램 코드(프로시저), 타입. 타입 이름은 프로그램 전체가 *유효범위scope*였고, 변수 이름과 프로시져의 이름은 *유효범위scope*를 한정할 수 있었다.

K에서는 같은 이름을 타입 이름이나 변수 이름으로 사용해도 프로그램 텍스트에서 문제없이 구분할 수 있다. 한 이름이 프로그램 텍스트에서 타입 이름으로 쓰이는지 변수 이름으로 쓰이는지 구분된다. 타입 이름이었으면 그 속내용은 타입이름 환경

$$\Delta \in TypeId \xrightarrow{\text{fin}} Type$$

를 참조하면 되고, 변수 이름이었으면 변수의 타입 환경

$$\Gamma \in Id \xrightarrow{\text{fin}} Type$$

를 참조하면 된다. 이런 경우, 타입 이름과 메모리 주소의 이름은 다른 *이름 공간*name space을 가진다고 한다. *TypeId*와 *Id*로 구분되는. 한편, K 언어에서 변수 이름과 프로시져 이름은 같은 *이름 공간*name space *Id*을 공유한다. 이름이 변수로 쓰였는지 프로시져로 쓰였는지를 구분할 수 있지만. 모두 변수의 타입 환경을 참조한다.

같은 타입이란 무엇인가? 타입에 이름을 붙일 때, 그 내용이 같더라도 다른 이름을 붙일 수 있다:

```
type a = {int x; int y}
type b = {int y; int x}
```

내용은 같지만 이름이 다르면 다른 타입인 것으로 정의할 수 도 있고(*name equivalence*), 내용만 같다면 같은 타입이라고 정의할 수 도 있겠다(*structural equivalence*). K 언어의 타입 시스템은 타입 이름이 같아야 같은 타입인 것으로 정의한 셈이다.

4.6 정리

이 장에서는 상식선에서 프로그래밍 언어를 디자인했던 과정을 따라가 보았다. 현재 대부분의 널리 쓰이는 명령형 프로그래밍 언어들이 만들어진 과정이었기도 하다. 그 대표적인 경우가 C라는 언어이다. 어떻게 프로그래밍 언어를 디자인해야 하는 지, 무엇이 문제일지가 드러나지 않은 상태에서, 주어진 디지탈 컴퓨터를 편하게 사용하기 위한 도구로서 프로그래밍 언어를 디자인했던 과거.

이렇게 언어를 만들었던 과정은 당시로서는 매우 상식적이지만 현재에서 바라보면 아쉬움이 많다. 현재의 프로그래밍 언어 기술에 기대어 보면 그러한

솜씨는 어설프다. 이 과정을 되밟아 오면서 어떤 문제들이 있는 지 살펴보았다.

그리고, 그 과정에서도 지금까지 살아남은 중요한 개념들을 살펴보았다.

- 디지탈 컴퓨터라는 기계, 폰노인만 *기계Von Neuman machine*, *보편만능의 기계universal machine*, 프로그램
- 메모리 주소에 이름 붙이기, 이름의 *유효범위scope*, 이름의 실체를 결정해 주는 *환경environment*, *자유로운 이름free name*과 *묶여있는 이름bound name*, *설탕구조syntactic sugar*
- 명령문에 이름 붙이기, *프로시저procedure* 호출, *재귀 호출recursive call*, 프로시져 호출이 이름의 유효범위에 미치는 영향, *동적/정적으로 결정되는 이름의 유효범위dynamic/static scoping*
- 메모리를 뭉치로 다루기, 데이타로서의 메모리 주소
- 메모리 관리, *메모리 재활용garbage collection*
- 프로그램의 오류를 미리 검증하기, 타입 오류, 타입 시스템, 타입 시스템의 *안전성soundness*과 *완전성completeness*, 타입에 이름 붙이기

5 장

값 중심의 언어

프로그래밍 언어를 제대로 디자인해 보자. 디지탈 컴퓨터와 거리를 두자. 기계를 편리하게 사용하려고 언어를 디자인하는게 아니다. 더 높은 눈 높이를 가지자. 소프트웨어의 문제를 해결해주는 프로그래밍 언어를 디자인해 보자. 작성한 프로그램에 오류가 있는 지를 미리 자동으로 검증할 수 있도록 하고 싶다. 그리고 그 검증이 안전하다는 것을 보장하고싶다.

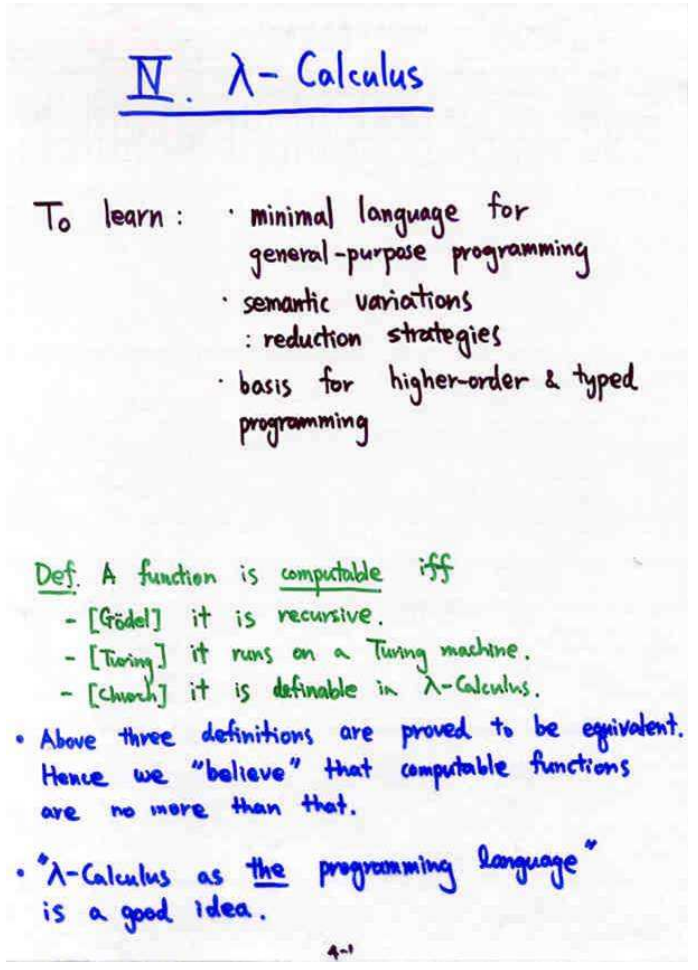
다른 차원의 눈 높이에서 고안된 언어. 그 결과는 월등한 수준에서 기계를 편리하게 사용할 수 있게 해준다. "기계의 편리한 사용", 그 슬로건을 직접 좇아 다니면서는 달성할 수 없었던 결과. 좋은 소식은 과감히 (혹은 무심하게) 다른 차원으로 발길을 돌리면서 만나게 된다.

이렇게 디자인 된 언어는 공허하지 않다. 기계중심의 언어들 만큼 효율적으로 실행되고 있고, 광범위한 상용 소프트웨어 작성에 널리 사용되고 있다. 제 2세대 자동 오류 검증 기술이 프로그래밍 언어에 단단히 갖추어진 덕이다.

5.1 언어의 모델

프로그래밍 언어는 사실, 컴퓨터라는 기계가 세상에 나오기 전부터 이미 있었다. 프로그래밍 언어가 컴퓨터를 돌리기 위해 만들어진 게 아니고, 컴퓨터가 프로그래밍 언어를 돌리기 위해 만들어 졌다고 할 수 있다. 역사적으로 프로

그래밍 언어가 먼저라고 할 수 있기 때문이다.



1930-40년대에 이미 논리학자와 수학자들은 기계적으로 계산가능한 것이 무엇인지를 고민하기 시작했다. 괴델(Gödel)은 *부분 재귀 함수Partial Recursive Function* 꼴로 정의되는 함수들을 기계적으로 계산가능한 것들이라고 정의했다. 처치(Church)는 *람다 계산법Lambda Calculus*으로 계산될 수 있는 것들이라고 정의했고. 튜링(Turing)은 *튜링 기계Turing Machine*가 실행할 수 있는 것들이라고 정의했다.

그런데, 세 가지가 모두 같은 정의였다. 부분 재귀 함수는 람다 계산법으로 계산될 수 있고, 람다 함수는 튜링 기계로 실행할 수 있고, 튜링 기계가 실행하는 함수는 부분 재귀 함수 꼴로 정의할 수 있다.

그러고 나니 사람들은 기계적으로 계산 가능한 함수가 그게 다가 아닐까 믿

고 있다. 세 사람이 독자적으로 정의한 것들이 결국 다르지 않은 것으로 미루어. 이것을 "Turing-Church Thesis"라고 부른다.

이 셋 중에서 *람다 계산법Lambda Calculus*이 언어의 모습을 가지고 있다. 문법*syntax*이 있고 *의미정의semantics*가 있다.

5.1.1 *람다 계산법Lambda Calculus*

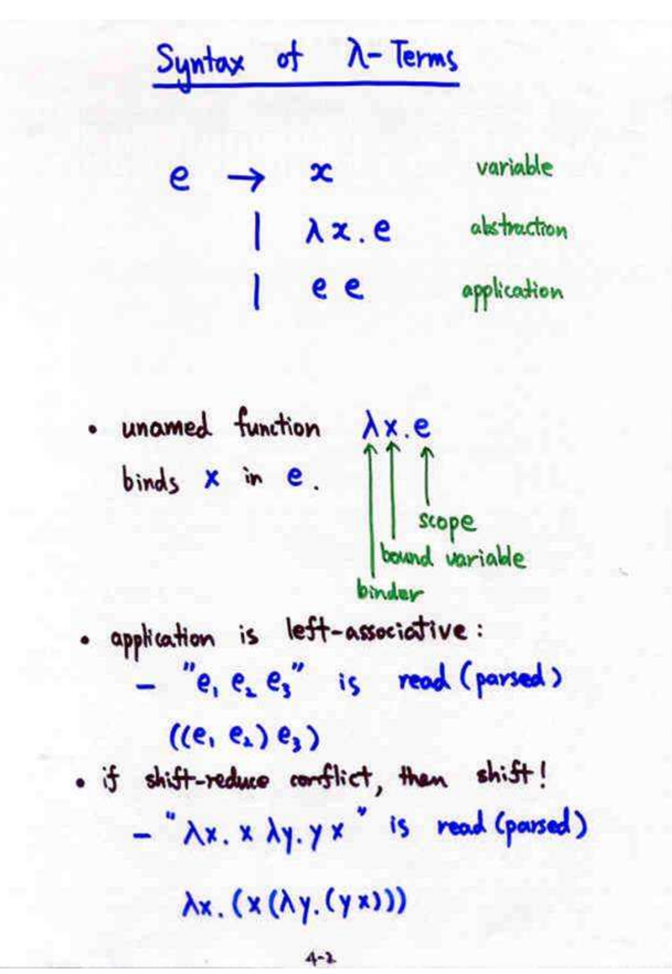
*람다 계산법Lambda Calculus*이라는 언어를 기초로 프로그래밍 언어를 디자인해가면 든든하다. 그 언어의 표현력이 완전하기 때문이다. 기계적으로 계산가능한 모든 것이 (컴퓨터로 돌릴 수 있는 모든 프로그램이) 바로 그 언어로 작성되는 것이라고 여겨지고 있으니. 이렇게 빠뜨림없이 기계적인 계산의 모든 것을 표현할 수 있는 언어를 "Turing-complete 하다"고 한다.

앞으로 *람다 게산법Lambda Calculus*을 언어이름으로 간단히 "람다"라고 부르겠다.

람다의 *문법구조syntax*는 간단하다:

$$
\begin{array}{llcll}
Exp & E & \to & x & \text{variable} \\
 & & \mid & \lambda x.E & \text{abstraction} \\
 & & \mid & E\ E & \text{application}
\end{array}
$$

x는 이름이다. $\lambda x.E$는 함수를 뜻한다. x는 함수의 인자고 E는 함수의 몸통이다. x의 유효범위는 E이다. $\lambda x.E$는 E에 나타나는 x를 "묶어*bind*준다"고 한다. $E_1\ E_2$는 함수를 적용하는 식이다. E_1은 계산하면 함수 $\lambda x.E$가 되야하고 E_2는 그 함수의 인자 x의 실체가 된다.



람다의 *의미구조semantics*도 간단하다. 사용할 표기법을 우선 익히자.

Notation 3 E의 *자유 변수free variable*들 $FV(E)$은 E에서 λ-식으로 묶여있지 않은 변수들이다.

$$\begin{array}{rcl}
FV(x) & = & \{x\} \\
FV(\lambda x.E) & = & FV(E) \setminus \{x\} \\
FV(E_1\ E_2) & = & FV(E_1) \cup FV(E_2)
\end{array}$$

□

Notation 4 *치환substitution*은 변수를 다른 것으로 바꿔주는 연산이다. 치환

S는 변수 x_i를 Y_i로 바꾸는 아이템 $x_i \mapsto Y_i$들의 집합이다.

$$\{x_1 \mapsto Y_1, \cdots, x_n \mapsto Y_n\}$$

모든 x_i들은 달라야 한다.

$Supp(S)$ (support set of S)는 S가 바꾸는 변수들 $\{x \mid S(x) \neq x\}$이다.

변수 $x \in A$를 $Y \in B$로 바꿔주는 치환들의 집합을

$$A \stackrel{\text{fin}}{\hookrightarrow} B$$

로 표현한다.

변수를 람다식으로 바꾸는 치환 $S \in Id \stackrel{\text{fin}}{\hookrightarrow} Exp$에 대해서, $FV(S)$는 바꿔들어갈 식들의 자유변수들의 합집합이다:

$$FV(S) = \bigcup_{1 \le i \le n} FV(E_i)$$

□

Notation 5 치환 S를 람다식 E에 적용하는 것을 $S\,E$로 쓰는데 다음과 같이 정의된다:

$$\begin{array}{rcl} S\,x &=& \begin{cases} E & \text{if } x \mapsto E \in S \\ x & \text{otherwise} \end{cases} \\ S\,(\lambda x.E) &=& \lambda y.(S\,\{x \mapsto y\}\,E) \qquad (\text{new } y) \\ S\,(E_1\,E_2) &=& (S\,E_1)(S\,E_2) \end{array}$$

위에서 "new y"란 나타난 적이 없는 변수 y라는 뜻이다:

$$y \notin \{x\} \cup FV(\lambda x.E) \cup FV(S).$$

□

Notation 6 두 개의 치환 S, T를 나란히 $S\,T$ 라고 쓰면, T로 바꾸고나서 S로

바꾸는 치환을 뜻한다:

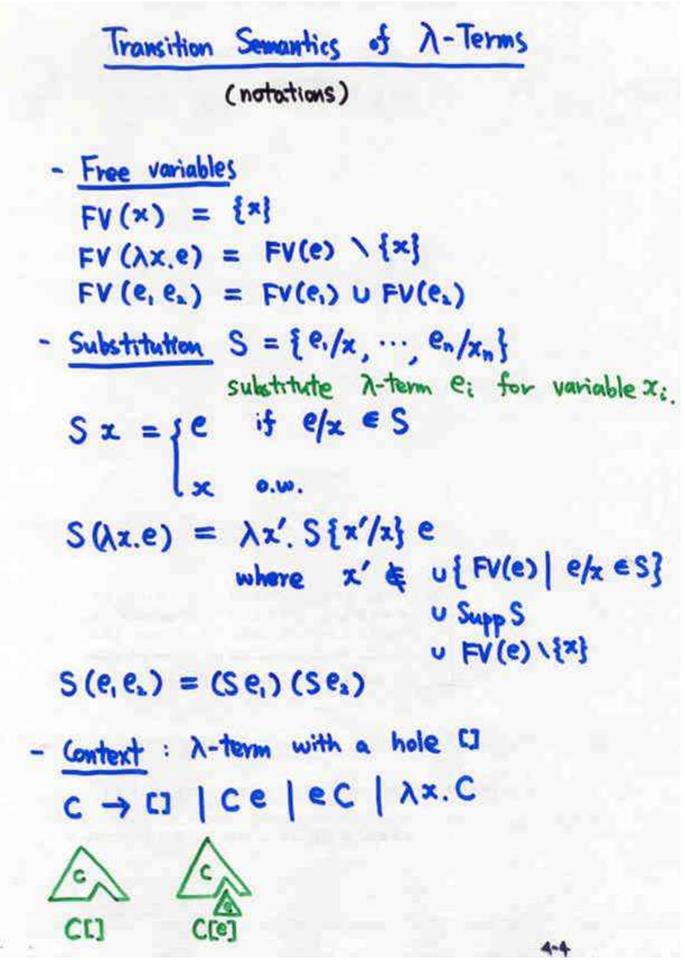
$$S\,T = \{x \mapsto S\,(T(x)) \mid x \in Supp(T)\} \cup \{x \mapsto S(x) \mid x \in Supp(T) \setminus Supp(S)\}$$

□

Notation 7 문맥구조*context* C는 빈칸 []을 딱 하나 품고 있는 람다식이다.

$$\text{Context} \quad C \quad \rightarrow \quad [] \mid C\ E \mid E\ C \mid \lambda x.C$$

빈칸을 품은 람다식을 강조하기 위해 "$C[]$"라고 쓰고, 그 빈칸에 들어있는 (다시 써야할) 람다식 E 까지 드러내어 "$C[E]$"라고 표현한다. 모든 람다식은 "$C[E]$"꼴로 표현되는 데, 그 람다식의 부품으로 E가 있다는 것을 뜻한다. □



Transition Semantics of λ-Terms

(the transition relation → between terms)

- $(\lambda x.e_1)\, e_2 \to_\beta \{e_2/x\}\, e_1$ (β-reduction)
- $\dfrac{e_1 \to e_2}{C[e_1] \to C[e_2]}$
- $\dfrac{x' \notin FV(\lambda x.e)}{\lambda x.e \to_\alpha \lambda x'.\{x'/x\}e}$ (α-conversion)

* the β-reduction is the only way of doing computation. (computing is applying functions!)
* expression $(\lambda x.e_1)\, e_2$, which fires the β-reduction, is called redex. (reduceable expression)
* a term without a redex is called normal. (every computation is done, i.e., a value!)

4-5

다음의 계산 규칙으로 람다식을 바꿔쓰는 것이 계산과정이다. 계산 규칙은 3개다.

$$(\beta\text{-reduction}) \qquad \overline{(\lambda x.E_1)\ E_2 \longrightarrow \{x \mapsto E_2\}E_1}$$

$$(\alpha\text{-reduction}) \qquad \frac{x' \notin FV(\lambda x.E)}{\lambda x.E \longrightarrow \lambda x'.\{x \mapsto x'\}E}$$

$$\frac{E_1 \longrightarrow E_2}{C[E_1] \longrightarrow C[E_2]}$$

α-연산*α-reduction*은 람다 함수의 인자 이름을 바꿔주기만 한다. 의미있는 계산은 아니다.

의미있는 계산 과정은 오직 하나다. β-연산*β-reduction* 규칙이다. 람다식에서 β-연산 가능한 부분을 찾아 다시 쓰고, 다시 그 결과에서 β-연산 가능한 부

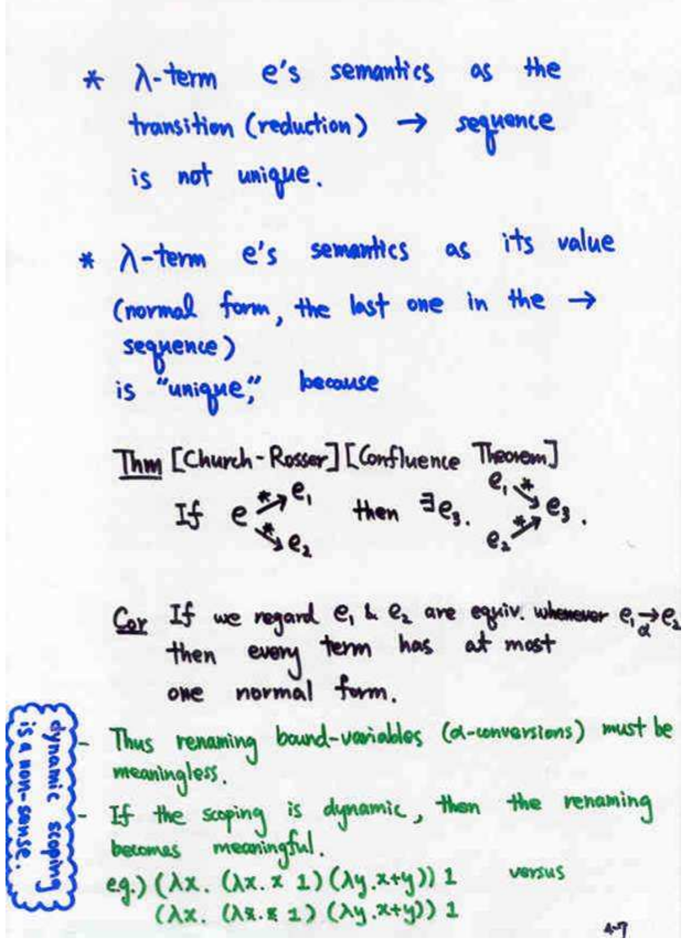
분을 찾아서 다시쓰고. 이 과정이 계산 과정이다. 람다식에서 β-연산 가능한 부분 "$(\lambda x.E)E_1$"을 *레덱스redex(reducible expression)*라고 부른다.

람다 계산 과정은 끝 나지 않고 영원히 계속될 수 도 있다. 끝나는 때는, β-연산 가능한 부분(*레덱스redex*)이 더 이상 없을 때 이다. 그런 람다식을 *정상식normal term*이라고 한다. *정상식normal term*으로 계산을 끝낼 수 있는 람다식을 *정상식normal term*을 가지고 있는 람다식이라고 한다.

위의 3가지 계산 규칙들이 유일한 바꿔쓰기 과정을 정의하지는 않는다. 람다식에서 바꿔쓰기 하는 순서가 여러가지가 있을 수 있다. *레덱스redex*가 여러 곳에 있을 수 있기 때문이다.

어느 *레덱스redex*부터 바꾸느냐에 따라 계산이 달라진다. 계산 순서에 따라, *정상식normal term*을 가진(끝날 수 있는) 람다식이 그 정상식으로 끝나지 못하고 영원히 계산을 계속할 수도 있다.

그러나, 람다식이 계산이 끝난다면, 그 결과는 오직 한가지만 가진다. 계산 순서가 다르게 진행되 왔을 수 도 있겠으나, 끝난다면 모두 하나의 *정상식normal term*으로 끝맺음된다. "Church-Rosser Theorem" 혹은 "Confluence Theorem"이 이것을 보장한다.



*정상식*normal term을 가진 람다식이라면, 어떤 순서로 계산을 해야 항상 그 정상식에 도달 할 수 있을까? 자칫 순서를 함부로하면 정상식에 도달 못하고 영원히 떠돌 수 있다.

*정상 순서*normal-order reduction로 계산하면 된다. *레덱스*redex가 여럿 있을 때, 그 중에 제일 왼쪽 제일 위의 *레덱스*redex를 계산해 가는 것이다. 이 순서로 계산하면 *정상식*normal term이 있는 람다식은 항상 그 정상식에 도달한다.

* λ-term does not always have a normal form.

$(\lambda x. x\, x)\ (\lambda x. x\, x) \to$

$(\lambda x. f\ (x\, x))\ (\lambda x. f\ (x\, x)) \to$

(non-terminating programs!)

* for λ-term with a normal form the reduction → rule does not guarantee to reach the normal form.

Thm [Standardization Theorem]
If a λ-term has a normal form then the normal-order reduction arrives at the normal form.

* Normal-order reduction

$$\frac{e_1 \to e_2}{C[e_1] \to C[e_2]}$$ e_1 is the left-most redex of the outer-most redius of $C[e_1]$.

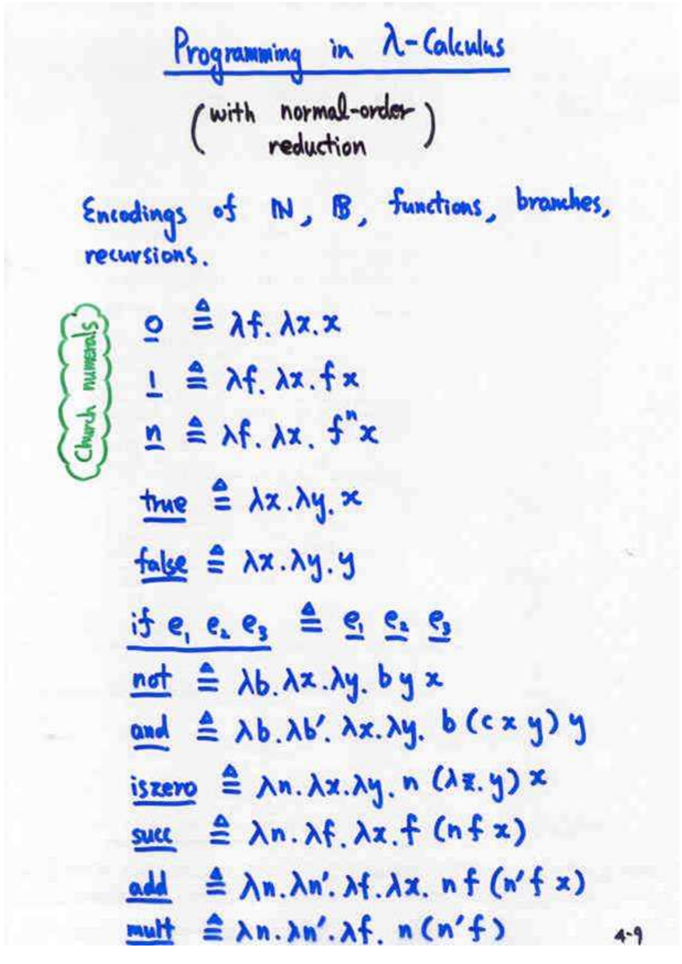
eg. $((\lambda x. e_1)\, e_2)\ ((\lambda x. x)\, y)$

4-8

5.1.2 람다로 프로그램하기

위와 같이 정의된 람다 언어로 프로그램을 한다? 기계적으로 계산가능한 모든 함수들을 작성할 수 있을까?

*정상 순서normal-order reduction*로 계산되는 람다를 가지고 자연수와 참/거짓을 다루는 연산들을 다음과 같이 정의할 수 있다.



하지만 기계적으로 계산 가능한 모든 함수를 람다로 짤 수 있으려면 재귀함수가 프로그램 될 수 있어야 한다. 다음과 같이 재귀함수도 정의할 수 있다.

Encoding of Recursive Functions in λ-Calculus

In ML, C, Java, etc.,

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

In λ-Calculus,

$$\underline{fac} \triangleq Y(\lambda f.\lambda n.\ \underline{if}\ \underline{n=0}\ \underline{1}\ \underline{n \times f(n-1)})$$

$$Y \triangleq \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$$

the fixpoint combinator

c.f.) the denotational semantics $[\![fac]\!]$ is

$$[\![fac]\!] \triangleq \underline{fix}\ \lambda f.\ \lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n \times f(n-1)$$

c.f.) $Yf = f(Yf)$, i.e., Y forms a fixpoint of its argument function.

Then

$$\underline{fac}\ \underline{n} \xrightarrow{*} \underline{n!}$$

4-10

이처럼 람다의 놀랄만한 능력이 간단한 문법과 계산방식에서 나온다. 모든 계산 가능한 함수들을 람다로 흉내낼 수 있다. 람다로 표현하고 람다의 계산법으로 실행하면 흉내내는 함수의 결과에 해당하는 람다 표현식이 등장한다.

Did you see/feel the power of λ's ?

* all "computable" functions are encoded in pure λ-calculus.

* 놀랄만큼 간단한 계산 (β reduction) 방식이 위의 것을 가능하게 하는구나.

* λx.e 를 함수라고 생각하고 β-reduction을 함수의 적용이라고 읽는다면 함수가 자유자재로 정의되고 함수가 적용되면서 함수가, 전달되고 결과로 나오고, 하는 등의 일이 계산의 모든 것이구나.

"high-order computation"

5.2 언어 키우기 0: M0

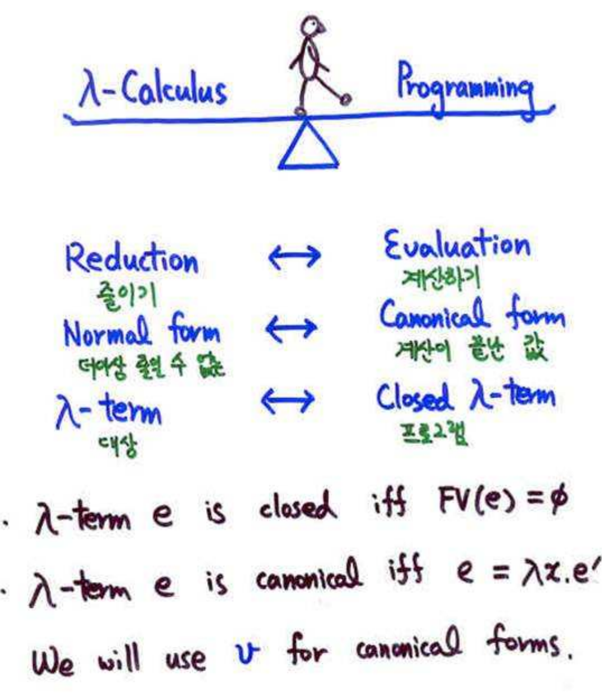
람다를 프로그래밍 언어로 바라보자. $\lambda x.E$ 는 함수 정의로 본다. 그런데 그 함수의 이름은 없다. 인자가 x이고 몸통이 E인 함수이다. $E_1\ E_2$는 함수를 적용하는 식이다.

람다에서 계산과 프로그램에서의 실행은 조금 다르다. 람다에서 계산의 끝은 식의 모든 구석구석에 *레덱스*redex가 없는 경우다. 함수식 $\lambda x.E$의 몸통 식 E에도 레덱스가 있다면 몸통식은 계산이 된다.

그러나 프로그램으로서 $\lambda x.E$는 최종 값이다. 자체로는 실행될 부품이 더 이상 없는. 몸통 E의 실행은 그 함수가 호출될 때 비로서 시작된다. 프로그램

에서 최종 값 $\lambda x.E$를 *기준 식*canonical term이라고 부른다.

앞으로 *기준 식*canonical term이 값이므로 종종 v("value")로 표기하겠다.



5.2.1 *소극적인 실행*lazy evaluation과 *적극적인 실행*eager evaluation

실행하면서 함수는 자체로 값이다. 실행할 부품이 더 이상 없다. 함수의 몸통은 그 함수가 호출될 때에 비로소 실행될 뿐이다.

그런데 함수가 호출될 때 E_1 E_2, 인자 식 E_2은 언제 실행해 주기로 할까? 두가지 선택이 있다. 일단 몸통을 실행하면서 인자 값이 필요할 때에 인자 식을 실행하기. 혹은, 몸통을 실행하기 전에 인자 식을 실행해서 인자 값을 구한 후에 몸통을 실행하기.

첫 번째 경우를 *소극적인 실행*lazy evaluation, normal-order evaluation 혹은 *식전*

달 호출*call-by-name, call-by-need*이라고 하고, 두 번째 경우를 *적극적인 실행eager evaluation* 혹은 *값전달 호출call-by-value*이라고 한다.

- Normal-order evaluation is also called "lazy evaluation" or "call-by-name."
  소극적 계산법
- Eager evaluation is 적극적 계산법.
  or "call-by-value."

* Eager evaluation is not "normal."
The normal (canonical) form cannot be found sometimes.
eg) $(\lambda x.y)((\lambda x.x\,x)(\lambda x.x\,x))$

* In eager evaluation we cannot encode if $e_1\,e_2\,e_3$ and recursive functions the same as before.
eg) if $e_1\,e_2\,e_3 \triangleq e_1\,e_2\,e_3$
$\Rightarrow_E v$ if (after) $e_2 \Rightarrow_E v_2 \wedge e_3 \Rightarrow_E v_3$

We need ifferent encoding to avoid the eager evaluation.

프로그래밍 언어로서 람다의 *소극적인 실행lazy evaluation, normal-order evaluation*과 *적극적인 실행eager evaluation*을 정의하는 규칙은 아래와 같다. 두번째 규칙이 두 실행 방식의 차이를 보여준다. 첫번째 규칙은 함수가 더 이상 실행될

부품이 없는 값임을 표현하고 있다.

lazy evaluation

$$\frac{}{\lambda x.E \Rightarrow_N \lambda x.E}$$

$$\frac{E_1 \Rightarrow_N \lambda x.E \quad \{x \mapsto E_2\}E \Rightarrow_N v}{E_1\ E_2 \Rightarrow_N v}$$

eager evaluation

$$\frac{}{\lambda x.E \Rightarrow_E \lambda x.E}$$

$$\frac{E_1 \Rightarrow_E \lambda x.E \quad E_2 \Rightarrow_E v \quad \{x \mapsto v\}E \Rightarrow_N v'}{E_1\ E_2 \Rightarrow_E v'}$$

소극적인 실행과 적극적인 실행은 장단점이 있다. 소극적으로 실행하면 적극적인 경우보다 빨리 계산되는 경우가 있고, 그 반대의 경우도 있다. 아래를 소극적으로 실행하면

$$(\lambda x.7)((\lambda x.x\ x)(\lambda x.x\ x)) \Rightarrow_N 7$$

이지만, 적극적으로 실행하면 영원히 돈다.

$$\begin{array}{ll} & (\lambda x.7)((\lambda x.x\ x)(\lambda x.x\ x)) \\ \Rightarrow_E & (\lambda x.7)((\lambda x.x\ x)(\lambda x.x\ x)) \\ \Rightarrow_E & \vdots \end{array}$$

한편, 아래를 소극적으로 실행하면

$$(\lambda x.(x \cdots x))E \Rightarrow_N (E \cdots E) \Rightarrow_N \cdots$$

E를 두 번 실행해야 하겠지만, 적극적으로 실행하면 한번만 실행하면 된다.

$$(\lambda x.(x \cdots x))E \Rightarrow_E^* (\lambda x.(x \cdots x))v \Rightarrow_E (v \cdots v) \Rightarrow_E \cdots .$$

5.2.2 시작 언어: 값은 오직 함수

람다 언어의 문법으로 시작하자.

$$\begin{array}{rcll} E & \to & x & \text{variable} \\ & \mid & \lambda x.E & \text{function} \\ & \mid & E\ E & \text{application} \end{array}$$

이 언어에서 이름 지을 수 있는 것은 값 뿐이다. 이름을 "변수"라고 하자. 변수 x의 *유효범위scope*는 그 이름을 묶은 함수 $\lambda x.E$의 몸통 E이다. 각 변수가 무슨 값을 뜻하는 지는, 그 변수를 묶은 함수가 호출될 때 전달되는 인자 값이 그 변수가 뜻하는 값이 된다.

의미정의는 프로그램 텍스트를 변화시키지 않으면서 정의하자. 기계중심 언어의 의미구조를 정의했을 때와 같은 스타일이다. 식 E의 의미 판단

$$\sigma \vdash E \Rightarrow v$$

은 환경 σ에서 식 E는 값 v를 계산한다, 는 뜻이다.

*환경environment*은 변수가 뜻하는 값을 결정해 준다. 의미정의에서 환경을 따로 가지는 이유는, 프로그램식을 변경시키지 않기 때문이다. 변수는 그대로 프로그램에 남아있는다. 예전같이 프로그램 이름이 텍스트에서 해당 값으로 바뀌써지는 방식이 아니다.

값은 정수와 함수뿐이다. 함수 값을 *클로져closure*라고 하는데, 함수 $\lambda x.E$ 텍스트 자체와 그 *함수가 정의될 때static scoping*의 환경이다. 환경의 역할은 $\lambda x.E$에서 자유로운 변수들의 값을 결정해준다.

$$\begin{array}{rcll} \sigma & \in & \mathit{Env} & = \mathit{Id} \xrightarrow{\text{fin}} \mathit{Val} \\ v & \in & \mathit{Val} & = \mathit{Closure} \\ \langle \lambda x.E, \sigma \rangle & \in & \mathit{Closure} & = \mathit{Exp} \times \mathit{Env} \end{array}$$

의미 규칙들은 다음과 같다. *적극적인 실행eager evaluation* 규칙을 가진다.

$\boxed{\sigma \vdash E \Rightarrow v}$

$$\frac{\sigma(x) = v}{\sigma \vdash x \Rightarrow v}$$

$$\frac{}{\sigma \vdash \lambda x.E \Rightarrow \langle \lambda x.E, \sigma}$$

$$\frac{\sigma \vdash E_1 \Rightarrow \langle \lambda x.E, \sigma' \rangle \quad \sigma \vdash E_2 \Rightarrow v \quad \sigma'\{x \mapsto v\} \vdash E \Rightarrow v'}{\sigma \vdash E_1\ E_2 \Rightarrow v'}$$

Evaluation Rules

$\boxed{\sigma \vdash e \Rightarrow v}$

separation of syntactic objects, codes / semantic objects, values

$\sigma \in Env = Var \xrightarrow{fin} Val$

$v \in Val = Lambda \times Env$ "closures"

$$\frac{\sigma(x) = v}{\sigma \vdash x \Rightarrow v}$$

$$\frac{}{\sigma \vdash \lambda x.e \Rightarrow \langle \lambda x.e, \sigma \rangle}$$

$$\frac{\sigma \vdash e_1 \Rightarrow \langle \lambda x.e', \sigma' \rangle \quad \sigma \vdash e_2 \Rightarrow v \quad \sigma'[v/x] \vdash e' \Rightarrow v'}{\sigma \vdash e_1\, e_2 \Rightarrow v'}$$

Dynamic scoping 이 무어냐고?

$$\frac{}{\sigma \vdash \lambda x.e \Rightarrow \lambda x.e}$$

$$\frac{\sigma \vdash e_1 \Rightarrow \lambda x.e' \quad \sigma \vdash e_2 \Rightarrow v \quad \sigma[v/x] \vdash e' \Rightarrow v}{\sigma \vdash e_1 e_2 \Rightarrow v}$$

4-17

5.3 언어 키우기 I: M1

M0를 가지고 모든 프로그램을 짤 수 있지만 불편하다. M0에 없었던 것들은 모두 설탕구조*syntactic sugar*지만, 프로그래밍의 편의를 위해 첨가하자.

5.3.1 다른 값: 정수, 참/거짓

정수, 참/거짓, 재귀함수 $\texttt{rec}\ f\ \lambda x.E$, 조건식 $\texttt{if}\ E\ E\ E$, 그리고 덧셈식 $E \texttt{+} E$과 동치식 $E \texttt{=} E$. (덧셈과 동치뿐 아니라 뺄셈 곱셈 또는 그리고 등이 모두 유용하겠지만 간략한 논의를 위해 생략한다.)

$$
\begin{array}{rcll}
E & \to & \vdots & \\
 & | & n & \text{integer} \\
 & | & \texttt{true} \mid \texttt{false} & \\
 & | & \texttt{let}\ x \texttt{=} E\ \texttt{in}\ E & \\
 & | & \texttt{if}\ E\ E\ E & \\
 & | & \texttt{rec}\ f\ \lambda x.E & \\
 & | & E \texttt{+} E & \\
 & | & E \texttt{=} E &
\end{array}
$$

설탕을 녹여서 모두 M0언어로 표현할 수 있겠으나, 그 자체로 그대로 놔두자. 특히

$$\texttt{let}\ x \texttt{=} E_1\ \texttt{in}\ E_2$$

는 다음의 설탕이다:

$$(\lambda x.E_2)\ E_1$$

이제 값들은 세 종류가 있다: 함수값과 정수 그리고 참/거짓.

$$
\begin{array}{rclcl}
v & \in & \mathit{Val} & = & \mathit{Closure} + \mathbb{Z} + \mathbb{B} \\
z & \in & \mathbb{Z} & & \text{the integer set} \\
b & \in & \mathbb{B} & = & \{\mathit{true}, \mathit{false}\} \\
\sigma & \in & \mathit{Env} & = & \mathit{Id} \stackrel{\text{fin}}{\to} \mathit{Val}
\end{array}
$$

프로그램의 모든 식들은 값을 계산한다. 의미 규칙들은

$\boxed{\sigma \vdash E \Rightarrow v}$

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

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

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

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

$$\frac{\sigma \vdash E_1 \Rightarrow \mathit{true} \quad \sigma \vdash E_2 \Rightarrow v}{\sigma \vdash \texttt{if}\ E_1\ E_2\ E_3 \Rightarrow v}$$

$$\frac{\sigma \vdash E_1 \Rightarrow \mathit{false} \quad \sigma \vdash E_3 \Rightarrow v}{\sigma \vdash \texttt{if}\ E_1\ E_2\ E_3 \Rightarrow v}$$

$$\frac{}{\sigma \vdash \texttt{rec}\ f\ \lambda x.E \Rightarrow \langle \texttt{rec}\ f\ \lambda x.E, \sigma\rangle}$$

$$\frac{\sigma \vdash E_1 \Rightarrow z_1 \quad \sigma \vdash E_2 \Rightarrow z_2}{\sigma \vdash E_1\ \texttt{+}\ E_2 \Rightarrow z_1 + z_2}$$

$$\frac{\sigma \vdash E_1 \Rightarrow v_1 \quad \sigma \vdash E_2 \Rightarrow v_2}{\sigma \vdash E_1\ \texttt{=}\ E_2 \Rightarrow v_1 = v_2}$$

그리고 함수의 호출식의 의미 규칙에 재귀함수의 경우가 필요하다.

$$\frac{\sigma \vdash E_1 \Rightarrow \langle \texttt{rec}\ f\ \lambda x.E, \sigma'\rangle \quad \sigma \vdash E_2 \Rightarrow v \quad \sigma'\{f \mapsto \langle \texttt{rec}\ f\ \lambda x.E, \sigma'\rangle\}\{x \mapsto v\} \vdash E \Rightarrow v'}{\sigma \vdash E_1\ E_2 \Rightarrow v'}$$

$z_1 + z_2$는 정수에서 정의된 덧셈이다. $v_1 = v_2$는 동치인지 아닌지를 계산한다. 이때 v_i들은 모두 정수거나 참/거짓 값이어야 한다. 함수 값들인 경우 동치를 판단할 수 없다. 일반적으로 두 함수를 보고 같은 인자에 대해서 같은 결과를 항상 내 놓는 함수인지는 기계적으로 판단 불가능하다. 가능하다면 그것을 이용해서 *멈춰요 문제**Halting Problem*를 푸는 프로그램을 정의할 수 있기 때문이다.

5.4 언어 키우기 II: M2

5.4.1 구조가 있는 값: 쌍*pair*

구조를 가진 데이타를 프로그램하기 쉽게 하자. 쌍을 만들고 사용하는 방식을

제공하자. 이것들도 설탕이지만 첨가하자.

$$\begin{array}{rcl} E & \to & \vdots \\ & | & (E\texttt{,}E) \\ & | & E\texttt{.1} \\ & | & E\texttt{.2} \end{array}$$

이제 값들은 세가지 종류가 있다: 함수값, 정수, 참/거짓, 혹은 두 값의 쌍.

$$\begin{array}{rclcl} v & \in & Val & = & Closure + \mathbb{Z} + \mathbb{B} + Pair \\ \langle v, v' \rangle & \in & Pair & = & Val \times Val \\ \sigma & \in & Env & = & Id \stackrel{\text{fin}}{\to} Val \end{array}$$

(잠깐, 위의 Val 집합의 정의가 이상하다. Val이 자기 자신을 가지고 정의되었다: $Val = \cdots Val \times Val$. 그런 집합 Val은 물론 존재한다. 공집합도 아닌. 심지어는 $X = X \stackrel{\text{fin}}{\to} X$를 만족하는 공집합이 아닌 집합 X도 존재한다.)

의미규칙은 다음이 첨가된다.

$\boxed{\sigma \vdash E \Rightarrow v}$

$$\frac{\sigma \vdash E_1 \Rightarrow v_1 \quad \sigma \vdash E_2 \Rightarrow v_2}{\sigma \vdash (E_1\texttt{,}E_2) \Rightarrow \langle v_1, v_2 \rangle}$$

$$\frac{\sigma \vdash E \Rightarrow \langle v_1, v_2 \rangle}{\sigma \vdash E\texttt{.1} \Rightarrow v_1}$$

$$\frac{\sigma \vdash E \Rightarrow \langle v_1, v_2 \rangle}{\sigma \vdash E\texttt{.2} \Rightarrow v_2}$$

쌍을 만들고 쓸 수 있도록 하는 위의 것들은 모두 설탕이다. 어떻게 녹일 수 있을까? 어떻게 같은 일을 하는 프로그램을 M0가지고 만들 수 있을까? 다음 정의 $\underline{E}$ 대로 E를 녹일 수 있다:

$$\begin{array}{rcl} \underline{(E_1\texttt{,}E_2)} & = & \lambda m.(m\ \underline{E_1})\ \underline{E_2} \\ \underline{E\texttt{.1}} & = & \underline{E}(\lambda x.\lambda y.x) \\ \underline{E\texttt{.2}} & = & \underline{E}(\lambda x.\lambda y.y) \end{array}$$

쌍만 있으면 임의의 구조물을 만들 수 있다. 리스트도 표현할 수 있고, 나무 구조도 표현할 수 있고, 그래프도 표현할 수 있고, 함수나 테이블도 표현할 수 있다.

Example 24 예를 들어, 리스트를 구현해 보자. 어떤 리스트라도 두 가지만 있으면 만들 수 있다: 빈 리스트와 리스트 앞에 하나 덧 붙여 새로운 리스트를 만드는 방법. 이 두 개를 각각 `nil`과 `link`라고 하자. `nil`은 0으로 정의하자. (리스트의 원소로 0이 사용되지 않는다고 가정한다). `link`는 다음의 함수로 정의하자:

$$\lambda x.\lambda lst.(x\texttt{,}lst)$$

그러면 "`link 1 nil`" 은 1 하나 있는 리스트 [1]를, "`link 1 (link 2 nil)`"은 [1, 2]인 리스트를 만든다.

사용하는 방법으로는 리스트가 비어있는 지 판단하는 방법 `isNil?`과 리스트의 처음 원소와 나머지 원소를 계산하는 함수들 `head`와 `tail`만 있으면 된다. 다음과 같이 각각 정의하면 된다.

`isNil?`	`head`	`tail`
$\lambda lst.(lst = \texttt{nil})$	$\lambda lst.(lst\texttt{.1})$	$\lambda lst.(lst\texttt{.2})$

□

Example 25 또 다른 예로, 두 *갈래 나무구조binary tree*를 구현해 보자. 만드는 방법은 세가지: `empty`, `leaf`, `node`. 사용하는 방법은 세가지: `isEmpty?`, `left`, `right`. 다음과 같이 정의할 수 있다.

`empty`	`leaf`	`node`
0	$\lambda x.x$	$\lambda lt.\lambda rt.(\texttt{link}\ lt\ rt)$

`isEmpty?`	`left`	`right`
$\lambda t.(t = \texttt{empty})$	$\lambda t.(t\texttt{.1})$	$\lambda t.(t\texttt{.2})$

□

5.5 언어 키우기 III: M3

5.5.1 메모리 주소 값: 명령형 언어의 모습

컴퓨터 메모리를 사용하는 *명령형 언어의 모습imperative features*도 있으면 편리하다. 설탕이지만 첨가한다. 프로그래밍 편의를 위해서.

$$
\begin{array}{rcl}
E & \rightarrow & \vdots \\
  & \mid & \texttt{malloc}\, E \\
  & \mid & \texttt{!}E \\
  & \mid & E \;\texttt{:=}\; E \\
  & \mid & E \;;\; E
\end{array}
$$

새로운 메모리 주소를 할당받고, 메모리 주소에 저장된 값을 읽고, 메모리 주소에 값을 쓰게 된다. 메모리를 가지면서, 프로그램 실행 순서에 따라서 다른 결과를 계산하게 될 수 있다. 주소에 쓰고나서 읽느냐, 읽고나서 쓰느냐에 따라.

이제 값의 공간에는 메모리 주소가 포함된다:

$$
\begin{array}{rcll}
v & \in & \mathit{Val} & = \mathit{Closure} + \mathbb{Z} + \mathbb{B} + \mathit{Pair} + \mathit{Loc} \\
\ell & \in & \mathit{Loc} & \quad \text{an infinite coutable set} \\
\sigma & \in & \mathit{Env} & = \mathit{Id} \stackrel{\text{fin}}{\rightarrow} \mathit{Val}
\end{array}
$$

메모리 주소가 값이 되면서, 프로그램 식의 의미 판단에는 메모리가 필요하다. 메모리에는 프로그램의 식들이 계산되면서 일어난 반응들이 계속 쌓여간다. 의미 판단

$$\sigma, M \vdash E \Rightarrow v, M'$$

은 식 E가 환경 σ와 메모리 M에서 v를 계산하고 결과 메모리는 M' 이다.

메모리는 주소에서 값으로 가는 유한 함수다.

$$M \in Memory = Loc \xrightarrow{\text{fin}} Val$$

의미 규칙은 다음과 같다.

$\boxed{\sigma, M \vdash E \Rightarrow v, M'}$

$$\frac{\sigma, M \vdash E \Rightarrow n, M'}{\sigma, M \vdash \texttt{malloc}\, E \Rightarrow \ell, M'}\ n > 0, \{\ell, \ell+1, \cdots, \ell+n-1\} \not\subseteq \text{Dom}\, M'$$

$$\frac{\sigma, M \vdash E \Rightarrow \ell, M'}{\sigma, M \vdash \texttt{!}E \Rightarrow M'(\ell), M'}$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow \ell, M_1 \quad \sigma, M_1 \vdash E_2 \Rightarrow v_2, M_2}{\sigma, M \vdash E_1 \texttt{ := } E_2 \Rightarrow v_2, M_2\{\ell \mapsto v_2\}}\ \ell \in \text{Dom}\, M_2$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow M_1 \quad \sigma, M_1 \vdash E_2 \Rightarrow M_2}{\sigma, M \vdash E_1 \texttt{ ; } E_2 \Rightarrow M_2}$$

다른 모든 식들의 의미규칙에도 메모리 반응이 쌓여가는 것이 표현되야 할 것이다. 예를들어,

$$\frac{\sigma, M \vdash E_1 \Rightarrow \langle \lambda x.E, \sigma' \rangle, M_1 \quad \sigma \vdash E_2, M_1 \Rightarrow v, M_2 \quad \sigma'\{x \mapsto v\}, M_2 \vdash E \Rightarrow v', M_3}{\sigma, M \vdash E_1\ E_2 \Rightarrow v', M_3}$$

Exercise 2 메모리를 다루는 것들이 모두 설탕일 수 있다. 어떻게 M0로 그 설탕을 녹일 수 있을까?

M3의 문법을 다시 정리하면:

$$
\begin{array}{rcll}
E & \to & x & \text{variable} \\
& | & \lambda x.E & \text{function} \\
& | & E\ E & \text{application} \\
& | & n & \text{integer} \\
& | & \texttt{true} \mid \texttt{false} & \text{boolean} \\
& | & \texttt{let}\ x\ \texttt{=}\ E\ \texttt{in}\ E & \\
& | & \texttt{if}\ E\ E\ E & \\
& | & \texttt{rec}\ f\ \lambda x.E & \\
& | & E\ \texttt{+}\ E & \\
& | & E\ \texttt{=}\ E & \\
& | & \texttt{(}E\texttt{,}E\texttt{)} & \\
& | & E\texttt{.1} & \\
& | & E\texttt{.2} & \\
& | & \texttt{malloc}\ E & \\
& | & \texttt{!}E & \\
& | & E\ \texttt{:=}\ E & \\
& | & E\ \texttt{;}\ E &
\end{array}
$$

모든 것이 첫 3개 만이면 충분한 것들이지만, 프로그래밍의 편의 (혹은 프로그래밍 파라다임, 혹은 상위의 프로그래밍)을 위해서 설탕으로 제공되었다. 이제 더 필요한 것들이 있다면 비슷하게 설탕으로 첨가해갈 수 있을 것이다.

Exercise 3 반복문 `while` E `do` E 도 설탕이다. 우선 M3로 녹일 수 있다. M3는 모두 M0로 녹을 수 있었으므로, 결국에는 M0로 녹게된다.

5.6 오류 검증의 문제

M3는 유용하지만, 실행되서는 않될 프로그램들이 있다. 실행중에 타입이 맞지 않아서 실행을 멈추게 되는 경우를 가진 프로그램들이다.

Types

(품새, 꼴, 격)

- 프로그램을 기획하는 한 방법
- 프로그래밍 언어가 그러한 기획을 무리없이 할 수 있도록 구성되있어야.

  "너무 빡빡하지는 않게" (PASCAL, PL/1, Algol 68, Ada)

* 프로그램이 기획한대로 실행된다는 것이 보장되어야
* 프로그램 기획이 제대로 됐는지 실행전에 검증될 수 있어야.
* 검증이 자동으로 된다면.

M3 프로그램은 다섯 종류의 값들이 있고 각각 분별력있게 흘러다녀야 한다. 다섯종류는 함수, 정수, 참/거짓, 쌍, 메모리 주소이다. 함수 적용식 $E_1\ E_2$에서 식 E_1의 결과 값은 함수여야 한다. 덧셈식 $E_1 + E_2$ 의 두 부품식들의 결과는 정수여야 한다. 조건식 `if` $E_1\ E_2\ E_3$의 E_1의 결과는 참 또는 거짓이어야 한다. 쌍에서 첫번째 값을 뜻하는 식 E.1에서 E의 결과 값은 쌍이어야 한다. 메모리에 쓰는 식 $E_1 := E_2$에서 E_1의 결과 값은 메모리 주소여야 한다.

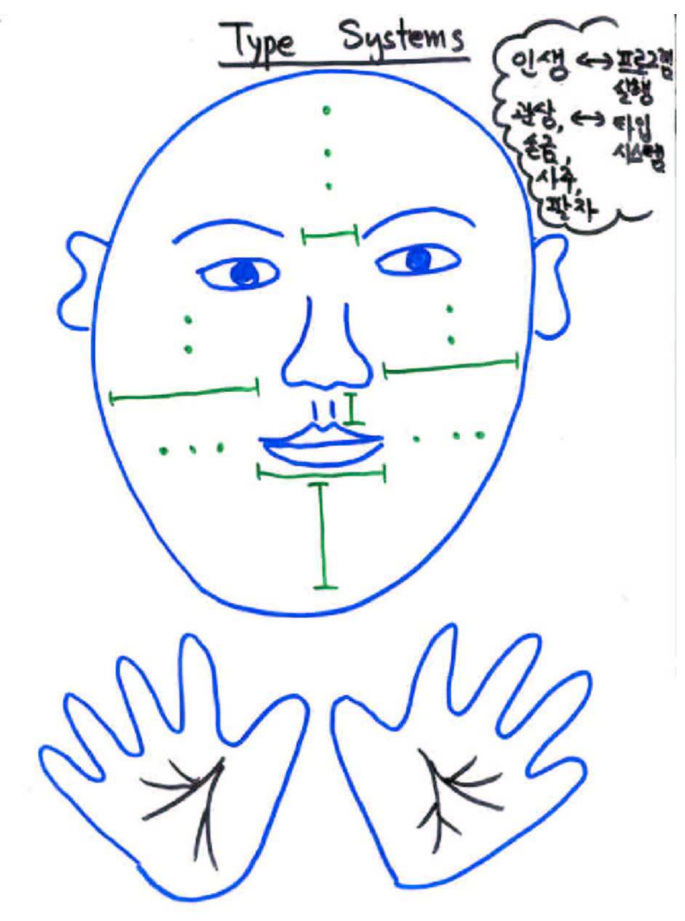
M3 프로그램이 위와 같은 타입의 조건을 실행중에 항상 만족하는 지를 미리 자동으로 검증하는 것이 가능했으면 한다. 또한 그 검증방법이 믿을 만 했

으면 한다.

*미리*static *안전*sound하게 검증하는 그러한 타입 시스템이 M3에 대해서는 가능해 진다.

5.6.1 *정적 타입 시스템static type system*

*정적 타입 시스템*static type system은 말이 되는 프로그램인지를 프로그램 텍스트만으로 판단하는 한 방법이다. 실행중에 확인하는 것이 아니라, 실행전에 미리 확인하는 시스템이다.



정적 타입 시스템은 *정적 의미구조*static semantics라고 부르기도 한다. 이에 반해, 지금까지의 의미정의는 프로그램 실행에 대한 것이므로 동적 *의미구조*dynamtic semantics라고 한다.

프로그램이 말이 되는 지를 미리 검증하는 방법들은 점점 정교해져가고 있는데, 정적 타입 시스템은 두번째로 정교한 방법이라고 할 수 있다.

첫번째 방법은 문법 검증이다. 제대로 생긴 프로그램인 지를 확인하는 방법이다. 문법 검증이 확인 하는 문법규칙은 간단한 귀납규칙들이다: 부품이 제대로 생긴 프로그램이기만 하면 그 것들을 가지고 어떻게 어떻게하면 제대로 생긴 프로그램이 된다, 는 식이다. "간단한" 이유는 조건이 간단해서다. 부품이 제대로 생긴 프로그램이기만 하면 된다. 이렇게 조건이 간단하다보니, 말이 되지 않는 프로그램을 모두 걸러내지 못하는 경우가 많다.

Example 26 프로그램식 E의 문법규칙이 다음과 같다고 하자:

$$
\begin{array}{rcll}
E & \to & n & \text{정수} \\
& | & s & \text{스트링} \\
& | & E + E &
\end{array}
$$

이때 2+''a''는 문법규칙에 맞는다. 2와 ''a''는 제대로 생긴 것 들이고, 그것들을 가지고 +식을 만들었으므로. 하지만 말이 되지 않는 프로그램이다. 정수와 스트링은 더할 수 없다. □

대게의 문법규칙들이 요구하는 조건은 너무 간단하다. 말이되는 프로그램을 판단하기에는. 문맥을 생각하지 않는 규칙들이기 때문이다. 고려해야 할 문맥은 이런 것이다: +식은 두개의 부품식들이 모두 정수식이어야 한다는. 이러한 문맥에 대한 조건을 대게의 "문법규칙"에는 표현하지 않는다. 표현하고 정교하게 다듬으면 그것은 우리가 "문법규칙"의 영역에서 "정적 타입 시스템"의 영역으로 응근슬쩍 넘어가게 되는 셈이다.

"정적 타입 시스템"에서는 어떤 문맥을 고려하는가? 프로그램의 겉모습만을 따지는 문맥이 아니라, 한 단계 더 파고 들어서, 텍스트가 가지는 속내용을 문맥으로 고려해야 한다. 정적 타입 시스템은 타입을 문맥으로 고려한다. 예를들어, "x+1"이 말이 되는 프로그램이려면 변수 x는 정수값을 가진 변수여야 한다. 타입 시스템은 x가 정수값을 가지는 상황에서만 말이된다고 판단한다. 타입은 서로 다른 종류의 값들에 붙이는 이름이다.

타입 시스템은 타입에 맞는 값들이 분별있게 계산되는 프로그램인지를 판단해 준다. 프로그램의 실행은 일종의 계산이고 그 계산중에 값들은 분별있게 흘러다닌다. 더하기 계산에는 숫자들이, 결혼이라는 계산에는 남자와 여자가, 라면끓이기 계산에는 라면과 물과 불이, 수력발전 계산에는 물과 터빈이, 등등. 숫자더하기에 남자와 여자가 들어가면 안되고, 수력발전에 불과 자동차가 들어가면 안된다. 이 분별은 더욱 정교할 수 도 있다. 결혼이라는 계산에는 나이차가 20년이하인 남자와 여자가, 라면끓이기 계산에는 라면 하나, 물 500cc 안팎, 가스레인지 불 정도 크기의 화력이 들어와야 한다는 식의.

타입 시스템은 일종의 *정적 프로그램 분석static program analysis*이다. 프로그램이 실행중에 가지는 어떤 성질을 실행하기 전에 예측하는 것이다. 분석하고자 하는 성질은, 프로그램이 타입에 맞게 실행될 것인지 여부다.

대부분의 프로그램 분석이 그렇듯이, 대부분의 정적 타입 시스템은 *완전하지 못하다incomplete*. 타입에 맞게 잘 실행되는 프로그램이지만 타입 시스템에 따라 살펴보고 결론으로 "잘모르겠다"라고 할 수 있는 것이다. 그렇지만, *안전sound*하다는 것은 보장된다. "예"라고 판단된 프로그램은 반드시 타입에 맞게 실행된다는 것이 보장된다. 아니, 보장되도록 우리가 타입 시스템을 고안하게 된다.

*정적 타입 시스템static type system*은 *형식 논리formal logic* 시스템이다. 논리식의 생김새, 논리식의 의미(참 혹은 거짓), 참인 논리식을 기계적으로 *추론하는 규칙들inference rules, proof rules*, 그 규칙들이 좋다는 증명(*안전성과 완전성 증명soundness and/or completeness*)으로 구성된다.

그렇게 디자인이 끝나면, 구현하게 된다. 구현된 타입 시스템은 대상 프로그래밍 언어로 작성된 임의의 프로그램을 입력으로 받아 타입 오류가 있을 지 없을 지를 검증한다. 구현되는 알고리즘이 디자인 한 타입 시스템을 제대로 구현한 것이라는 사실도 확인하게 된다. 디자인에서 구현까지 단단한 과정을 밟는다.

5.6.2 형식 논리 시스템*formal logic* 리뷰

형식 논리 시스템을 우선 복습해 보자.

형식 논리 시스템은 다음의 것들로 구성된다: 논리식 집합의 정의, 논리식 의미의 정의, 참논리식 집합의 정의, 참논리식 추론의 방법, 추론방법 평가하기.

*선언 논리*propostional logic 시스템을 보자. 논리식 집합은 다음과 같은 귀납 규칙으로 정의된다:

$$
\begin{array}{rcl}
f & \to & T \mid F \\
 & \mid & \neg f \\
 & \mid & f \wedge f \\
 & \mid & f \vee f \\
 & \mid & f \Rightarrow f
\end{array}
$$

논리식의 정의는 다음과 같고:

$$
\begin{array}{rcl}
[\![T]\!] & = & \text{true} \\
[\![F]\!] & = & \text{false} \\
[\![\neg f]\!] & = & \text{not}[\![f]\!] \\
[\![f_1 \wedge f_2]\!] & = & [\![f_1]\!]\ \text{andalso}\ [\![f_2]\!] \\
[\![f_1 \vee f_2]\!] & = & [\![f_1]\!]\ \text{orelse}\ [\![f_2]\!] \\
[\![f_1 \Rightarrow f_2]\!] & = & [\![f_1]\!]\ \text{implies}\ [\![f_2]\!]
\end{array}
$$

이제 참인 논리식들의 집합을 정의한다: 즉, 쌍 $(\{f_1, \cdots, f_n\}, f)$(표기: "$\Gamma \vdash f$") 들의 집합이다. 참인 식 $f_1 \wedge \cdots \wedge f_n \Rightarrow f$ 들의 집합들:

$$\frac{}{\Gamma \vdash T} \quad \frac{}{\Gamma \vdash f}\ f \in \Gamma \qquad\qquad \frac{\Gamma \vdash F}{\Gamma \vdash f} \quad \frac{\Gamma \vdash \neg\neg f}{\Gamma \vdash f}$$

$$\frac{\Gamma \vdash f_1 \quad \Gamma \vdash f_2}{\Gamma \vdash f_1 \wedge f_2} \qquad\qquad \frac{\Gamma \vdash f_1 \wedge f_2}{\Gamma \vdash f_1}$$

$$\frac{\Gamma \vdash f_1}{\Gamma \vdash f_1 \vee f_2} \qquad\qquad \frac{\Gamma \vdash f_1 \vee f_2 \quad \Gamma \cup \{f_1\} \vdash f_3 \quad \Gamma \cup \{f_2\} \vdash f_3}{\Gamma \vdash f_3}$$

$$\frac{\Gamma \cup \{f_1\} \vdash f_2}{\Gamma \vdash f_1 \Rightarrow f_2} \qquad\qquad \frac{\Gamma \vdash f_1 \Rightarrow f_2 \quad \Gamma \vdash f_1}{\Gamma \vdash f_2}$$

$$\frac{\Gamma \cup \{f\} \vdash F}{\Gamma \vdash \neg f} \qquad\qquad \frac{\Gamma \vdash f \quad \Gamma \vdash \neg f}{\Gamma \vdash F}$$

이 규칙들은 참인 식을 만드는 규칙이고, 이것을 논리 시스템에서는 *추론규칙inference rules* 혹은 *증명규칙proof rules*이라고 한다.

여기서 살짝 다른 관점: 참논리식 집합을 만드는 과정은 곧, 참논리식을 유추한 증명을 만드는 과정이기도하다. 증명들의 집합을 만드는 귀납규칙으로 보는 것이다. 예를 들어, 증명규칙

$$\frac{\Gamma \vdash f_1 \quad \Gamma \vdash f_2}{\Gamma \vdash f_1 \wedge f_2}$$

은 증명을 만드는 귀납 규칙 $\Gamma \vdash f_1$와 $\Gamma \vdash f_2$의 증명들을 가지고 $\Gamma \vdash f_1 \wedge f_2$의 증명을 만든다.

이렇게 만들어지는 증명을 증명 *나무proof tree*라고 한다:

$$\dfrac{\dfrac{}{\{p \to \neg p, p\} \vdash p} \quad \dfrac{\dfrac{}{\{p \to \neg p, p\} \vdash p \to \neg p} \quad \dfrac{}{\{p \to \neg p, p\} \vdash p}}{\{p \to \neg p, p\} \vdash \neg p}}{\dfrac{\{p \to \neg p, p\} \vdash F}{\{p \to \neg p\} \vdash \neg p}}$$

이제 그러한 "기계적인"(아무생각없는, 의미를 생각하지 않는) 증명 *규칙* 혹은 추론 *규칙proof rules, inference rules*이 정말 좋은가?를 따진다. 즉, 추론 규칙들로 만드는 $\{g_1, \cdots, g_n\} \vdash f$ 는 어떤 것들인가? 예) $[\![g_1 \wedge \cdots \wedge g_n \Rightarrow f]\!] = \text{true}$ 인가? 여기, 두개의 기준이 있다

- 추론 규칙의 *안전성soundness*:

$$\Gamma \vdash f \text{ 이면 } [\![\Gamma \Rightarrow f]\!] = \text{true}$$

- 추론 규칙의 *완전성completeness*:

$$\Gamma \vdash f \text{ 면이 } [\![\Gamma \Rightarrow f]\!] = \text{true}$$

5.7 단순 타입 시스템*simple type system*

M0로 모든 것을 프로그램할 수 있겠으나, 값은 오직 함수 밖에 없었다. 값의 타입으로 함수와 정수만 있는 M1의 코아만 생각하자.

$$\begin{array}{rcl} E & \to & n \\ & | & x \\ & | & \lambda x.E \\ & | & E\ E \\ & | & E + E \end{array}$$

위의 의미 규칙

$$\sigma \vdash E \Rightarrow v$$

를 사용할 수 도 있으나, 타입 시스템의 안전성 증명에서 편리한 방식을 위해서 프로그램 텍스트를 다시 써가는 방식의 의미 정의를 하자. 같은 의미 정의이지만 *실행 문맥evaluation context*을 이용해서 다음과 같이 정의하자:

$$\begin{array}{rrcl} Value & v & \to & n \\ & & \mid & \lambda x.E \\ \\ EvaluationContext & C & \to & [] \\ & & \mid & C\ E \\ & & \mid & v\ C \\ & & \mid & C + E \\ & & \mid & v + C \end{array}$$

실행 문맥은 *적극적인 실행eager evaluation* 순서에 준해서 다음에 실행해야 할 부분을 결정해 준다.

프로그램의 실행 규칙은 다음과 같다.

$$\frac{E \to E'}{C[E] \to C[E']}$$

$$\frac{}{(\lambda x.E)v \to \{x \mapsto v\}E}$$

$$\frac{}{z_1 + z_2 \to z_1 + z_2}$$

프로그램 E의 의미는 위의 규칙에 따라서

$$E \to E' \to \cdots$$

진행되는 과정으로 정의된다.

5.7.1 단순 타입 추론 규칙

이제 준비는 끝났다. 타입 시스템을 정의하자.

프로그램 텍스트만을 가지고 타입 오류없이 실행되는 프로그램인지를 검증하는 논리 시스템이다. 프로그램 텍스트에 타입이 명시된 것이 없다. 그래도 그 논리 시스템은 프로그램 구석구석의 타입이 어떻게 기계적으로 유추될 수 있는 지를 정의한다.

그 유추 시스템이 어떻게 구현되야 하는지, 과연 구현될 수는 있는 건지, 등에 대해서는 나중에 논의하기로 하자. 일단은, 그 논리 시스템이 잘 디자인되었으면 좋겠다. 믿을 수 있는 시스템이었으면 좋겠다. 그런 지를 먼저 확인한 후에, 그것을 충실히 구현할 방안을 모색하겠다.

타입 τ 는 프로그램 식들이 실행 결과로 가지는 값들의 종류들이다. 우리 언어에서는 다음의 것들로 구분될 것이다:

$$
\begin{array}{llcll}
\mathit{Type} & \tau & \rightarrow & \iota & \text{primitive type} \\
 & & | & \tau \rightarrow \tau & \text{function type}
\end{array}
$$

논리식의 생김새는

$$\Gamma \vdash E : \tau$$

이다.

Γ는 *타입 환경type environment*으로 변수들의 알려진 타입을 가지고 있다:

$$\Gamma \in \mathit{Id} \xrightarrow{\text{fin}} \mathit{Type}$$

Notation 8 타입 환경 Γ에 대해서 $\Gamma\{x \mapsto \tau\}$ 대신해서 $\Gamma + x : \tau$로도 쓰겠다. $x \mapsto \tau \in \Gamma$ 대신에 $x : \tau \in \Gamma$로도 쓰겠다. □

논리식의 의미는

$$\llbracket \Gamma \vdash E : \tau \rrbracket = \text{true}$$
$$\textit{iff}$$
$$\forall \sigma \models \Gamma . E \text{ 이 영원히 돌던가, 값을 계산 } E \to^* v \text{ 하면} \vdash v : \tau .$$

이제 참인 논리식(타입 판단)을 유추하는 규칙을 정의하자. *타입 추론 규칙type inference rules*이라고 한다:

$\boxed{\Gamma \vdash E : \tau}$

$$\frac{}{\Gamma \vdash n : \iota}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma + x : \tau_1 \vdash E : \tau_2}{\Gamma \vdash \lambda x.E : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash E_2 : \tau_1}{\Gamma \vdash E_1\ E_2 : \tau_2}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash E_2 : \tau_1}{\Gamma \vdash E_1 + E_2 : \tau_2}$$

Example 27 타입 유추가 성공되는 과정을 살짝 살피면 아래와 같다:

$$\frac{\dfrac{\vdots}{\vdash \lambda x.x + 1 : \iota \to \iota} \quad \dfrac{\vdots}{\vdash (\lambda y.y)\ 2 : \iota}}{\vdash (\lambda x.x + 1)\ ((\lambda y.y)\ 2) : \iota}$$

□

Example 28 타입 유추가 불가능한 경우는 아래와 같다:

$$\frac{\dfrac{\vdots}{\vdash \lambda x.x + 1 : \iota \to \iota} \quad \dfrac{\dfrac{\vdots}{\vdash \lambda y.y : \iota \to \iota} \quad \dfrac{!}{\vdash \lambda z.z : \iota}}{\vdash (\lambda y.y)\ (\lambda z.z) : \iota}}{\vdash (\lambda x.x + 1)\ ((\lambda y.y)\ (\lambda z.z)) : \iota}$$

□

위의 규칙의 특성을 살펴보자.

- 식 E의 생김새마다 오직 하나의 규칙이 있다. 이 사실은 증명규칙의 성질을 증명할 때 우리를 편하게 한다. 예를 들어서 $\Gamma \vdash \lambda x.E : \tau \to \tau'$이 유추되었다면 반드시 한 경우 밖에는 없다: $\lambda x.E$의 타입유추 규칙이 하나밖에 없으므로, 그 규칙에 있는 조건이 만족한 경우밖에는 없다. 즉,

$$\begin{array}{rcl} \Gamma \vdash \lambda x.E : \tau \to \tau' & \text{이면이} & \Gamma + x : \tau \vdash E : \tau' \\ \Gamma \vdash E\ E' : \tau & \text{이면이} & \Gamma \vdash E : \tau' \to \tau\ \wedge\ \Gamma \vdash E' : \tau' \\ & \vdots & \end{array}$$

- 어떤 식 E에 대해서, $\Gamma \vdash E : \tau$인 τ가 여럿가능하다.

$$\frac{\{x : \iota\} \vdash x : \iota}{\vdash \lambda x.x : \iota \to \iota} \qquad \frac{\{x : \iota \to \iota\} \vdash x : \iota \to \iota}{\vdash \lambda x.x : (\iota \to \iota) \to (\iota \to \iota)}$$

- 문제 없이 실행되는 프로그램이 타입 유추가 불가능한 경우가 있다. *완전complete*하지는 못한 것이다.

$$\dfrac{\dfrac{\dfrac{\vdots}{\{f : \tau \to \tau'\} \vdash f : \tau \to \tau'} \quad \dfrac{\vdots}{\{f : \tau \to \tau'\} \vdash f : \tau}\ \color{red}{\tau = \tau \to \tau'}}{\{f : \tau \to \tau'\} \vdash f\ f : \tau'}}{\vdash \lambda f.f\ f : (\tau \to \tau') \to \tau'}$$

Static Semantics

Note 1 ∃1 typing rule for each expression construct.

Note 2 Some expression can have multiple types.

$\lambda x.x : \iota \to \iota$

$\qquad (\iota \to \iota) \to (\iota \to \iota)$

$\qquad \vdots$

Note 3 let $I = \lambda x.x$ in $I\,I$ cannot have a type.

6

5.7.2 추론 규칙의 안전성

우리는 추론 규칙의 *안전성soundness*을 증명할 수 있다. 안전성이란, 프로그램이 타입 추론이 성공하면 그 프로그램은 타입 오류 없이 실행된다는 것이 보장됨, 이다. 즉, 프로그램 (자유 변수가 없는 식) E에 대해서

$$\vdash E : \tau$$

이 추론 될 수 있다면, 프로그램 E는 문제 없이 실행되고, 끝난다면 결과 값이 τ 타입이라는 것이다.

타입 시스템의 안전성을 증명하는 방법으로 대표적인 두 가지를 살펴보자.

5.7.2.1 안전성 증명 방법 I

두개의 정리를 증명하는 것이다.

- Progress Lemma: 값이 나올 때 까지 문제없이 진행한다.

  $\vdash e : \tau$ 이고 e가 값이 아니면 반드시 $e \rightarrow e'$.

- Subject Reduction Lemma: 진행은 타입을 보존한다.

  $\vdash e : \tau$ 이고 $e \rightarrow e'$ 이면 $\vdash e' : \tau$.

위의 두 정리가 증명되면 타입 유추 규칙의 *안전성soundness*이 증명되는 것이다. 타입 유추가 성공한 프로그램은 항상 문제 없이 실행되는 것이고, 끝난다면 유한 스텝만에 값으로 결과를 내는 것이므로 그 결과는 유추한 프로그램의 타입을 가진다. 두 정리의 증명과정을 살펴 보자. 모두 귀납법으로 진행되는 손 쉬운 증명이다.

Lemma 1 (Progress) *$\vdash E : \tau$ 이고 E가 값이 아니면 반드시 진행 $E \rightarrow E'$ 한다.*

Proof. $\vdash E : \tau$의 증명에 대한 귀납법으로. (증명 규칙이 E의 구조를 따라 귀납하므로, "E의 구조에 대한 귀납법으로"("By structural induction on E")라고 해도 된다.)

$E = E_1\ E_2$인 경우: $\vdash E_1\ E_2 : \tau$이므로 타입추론 규칙에 의해 $\vdash E_1 : \tau' \rightarrow \tau$이고 $\vdash E_2 : \tau'$ 이다. 따라서, 귀납 가정에 의해서,

- E_1이 값이 아니면 진행 $E_1 \rightarrow E_1'$ 하고, 이는 곧 프로그램 실행 $\rightarrow$의 정의에 의해 $E_1\ E_2 \rightarrow E_1'\ E_2$과 같다.

- 마찬가지로, E_1이 값이고 E_2가 값이 아니라면 진행 $E_2 \rightarrow E_2'$ 하고, 이는 곧 프로그램 실행 $\rightarrow$의 정의에 의해 $E_1\ E_2 \rightarrow E_1\ E_2'$과 같다.

- E_1과 E_2가 모두 값이라면, $\vdash E_1 : \tau' \rightarrow \tau$일 수 있는 값 E_1은 오직 $\lambda x.e'$경우 뿐이다. 따라서 프로그램 실행 $\rightarrow$의 정의에 의해 반드시 진행 $E_1\ E_2 = \lambda x.E'\ E_2 \rightarrow \{v/x\}E'$한다.

다른 경우도 마찬가지로 증명. □

Lemma 2 (Subject Reduction, Preservation) *$\vdash E : \tau$ 이고 $E \rightarrow E'$ 이면 $\vdash E' : \tau$.*

Proof. $\vdash E : \tau$의 증명에 대한 귀납법으로 진행한다. [1]

$e = E_1\ E_2$인 경우: $\vdash E_1\ E_2 : \tau$이므로 타입추론 규칙에 의해 $\vdash E_1 : \tau' \rightarrow \tau$이고 $\vdash E_2 : \tau'$ 이다. $E_1\ E_2 \rightarrow E'$이라면 세가지 경우밖에 없다:

- $E_1 \rightarrow E_1'$이라서 $E_1\ E_2 \rightarrow E_1'\ E_2$ 인 경우. 귀납 가정에 의해 $\vdash E_1' : \tau' \rightarrow \tau$. $\vdash E_2 : \tau'$이므로, 타입추론 규칙에 의해 $\vdash E_1'\ E_2 : \tau$.
- E_1은 값이고 $E_2 \rightarrow E_2'$이라서 $E_1\ E_2 \rightarrow E_1\ E_2'$ 인 경우. 위의 경우와 유사.
- E_1과 E_2가 모두 값인 경우. $\vdash E_1 : \tau' \rightarrow \tau$인 값 E_1은 타입추론 규칙에 의해 $\lambda x.E'$ 밖에는 없다. 즉, $E_1\ E_2 = (\lambda x.E')\ v$ 이고, $(\lambda x.E')\ v \rightarrow \{v/x\}E'$이다. $\vdash \lambda x.E' : \tau' \rightarrow \tau$이라면 타입추론 규칙에 의햇 $x : \tau' \vdash E' : \tau$이다. $\vdash v : \tau'$ 이므로, "Preservation under Substitution Lemma"에 의해 $\vdash \{v/x\}E' : \tau$이다.

□

치환은 타입을 보존한다.

Lemma 3 (Preservation under Substitution) *$\Gamma \vdash v : \tau'$ 이고 $\Gamma + x : \tau' \vdash E : \tau$ 이면 $\Gamma \vdash \{v/x\}e : \tau$.*

Proof. $\Gamma + x : \tau' \vdash E : \tau$의 증명에 대한 귀납법으로 증명한다.

$E = \lambda y.E'$ 인 경우: 항상 $y \notin \{x\} \cup FV v$인 $\lambda y.E'$로 간주할 수 있으므로 $\{v/x\}\lambda y.E' = \lambda y.\{v/x\}E'$. 따라서, 보일 것은 $\Gamma \vdash \lambda y.\{v/x\}E' : \tau \stackrel{\text{let}}{=} \tau_1 \rightarrow \tau_2$.

가정 $\Gamma + x : \tau' \vdash \lambda y.E' : \tau_1 \rightarrow \tau_2$ 으로부터 타입추론 규칙에 의해 $\Gamma + x : \tau' + y : \tau_1 \vdash E' : \tau_2$ 이고, $\Gamma \vdash v : \tau'$와 $y \notin FV(v)$ 으로부터 $\Gamma + y : \tau_1 \vdash v : \tau'$[2]

[1] 증명 규칙이 e의 구조를 따라 귀납하므로, "e의 구조에 대한 귀납법으로 진행한다"고 해도 된다.

[2] 물타기*Weakening Lemma* 정리(정의와 증명이 쉽다.)

이므로, 귀납 가정에 의해 $\Gamma + y : \tau_1 \vdash \{v/x\}E' : \tau_2$. 즉, 타입추론 규칙에 의해 $\Gamma \vdash \lambda y.\{v/x\}E' : \tau_1 \to \tau_2$.

다른 경우는 더욱 단순한 귀납.□

5.7.2.2 안전성 증명 방법 II

다음 두개만으로도 타입 유추 시스템의 *안전성soundness* 증명이 가능하다.

- $E \to error$를 정의한다.
- Subject Reduction Lemma: 진행은 타입을 유지한다, 를 증명한다.

  $\vdash E : \tau$ 이고 $E \to E'$ 이면 $\vdash E' : \tau$ 이다.

왜 일까?

- 안전성이란, 값이 아닌 식 E가 타입이 있으면, E는 문제없이 진행하며 끝난다면 그 타입의 값이어야한다.
- 값이 아닌 식 E가 타입이 있다고 하자. 잘 진행하는가?

  그렇다, $E \to error$로 진행할 수 없다. 왜냐하면, 그렇게 진행한다면 "Subject Reduction Lemma"에 의해서 모순이기 때문이다. *error*가 타입이 있어야 하는데 *error*의 타입을 결정하는 규칙은 없기 때문이다.
- 그리고 진행이 타입을 항상 보존하므로, 값으로 진행이 끝나게 되면 그 값도 같은 타입을 가진다.

5.7.3 추론 규칙의 구현

위의 *단순 타입 시스템simple type system*은 논리 시스템으로 우리가 바라는 바를 가지고 있슴을 증명할 수 있었다. 이제는 구현이다.

하지만 구현이 간단해 보이지는 않는다. 다음의 규칙을 보면 그렇다:

$$\frac{\Gamma + x : \tau_1 \vdash E : \tau_2}{\Gamma \vdash \lambda x.E : \tau_1 \to \tau_2}$$

함수의 몸통 E를 타입 유추하려면 인자의 타입 τ_1 를 알고 있어야 한다. 그러나 τ_1은 우리가 최종적으로 유추해야 할 함수 타입의 일부이다.

유추해내야 하는 것을 미리 알고 있어야 한다. 어떻게 해야 하나? 유추*inference*를 포기해야하나? 프로그래머에게 함수 선언마다 인자 타입등을 프로그램 텍스트에 써달라고 강요해야 하나?

이번엔 놀라운 방식이 있다. 타입에 대한 연립방정식을 세우고 풀어가는 과정이다.

그 과정이 놀라운 이유는 두가지다. 우선, 타입을 자동으로 유추해 주는 알고리즘이기 때문이다. 프로그램 텍스트에 타입에 대해서 프로그래머가 아무것도 명시할 필요가 없다. 자동으로 유추가 된다.

다른 이유는, 타입 시스템을 충실히 구현한 알고리즘이기 때문이다. 충실히? 타입 시스템이 유추할 수 있는 것을 그 알고리즘은 똑 같이 유추해 준다. 그 반대도 성립한다. 알고리즘이 유추해 내었으면 타입 시스템도 같은 것을 유추해 낼 수 있다. 서로 iff 관계로 논리 시스템과 구현 시스템이 물려있다.

5.7.3.1 연립 방정식의 도출

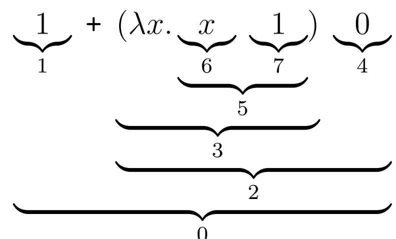
주어진 프로그램 E를 주욱 훑으면서 각 부품식들의 타입에 대한 연립방정식을 도출한다. 부품식이 서로 엉겨있는 문맥을 통해서 그 식의 값이 어떤 타입이 되야 할 지에 대한 방정식(제약사항)이 도출된다. 그 연립 방정식을 풀면된다.

그 연립 방정식에서 미지수는 프로그램의 각 부품식마다 하나씩 있고, 변수마다 하나씩 있다. 그 부품식과 변수의 타입을 뜻하는 미지수이다.

Example 29 예를 들어, 다음의 식을 생각해 보자. 다음 프로그램

$$1 + (\lambda x.x\ 1)\ 0$$

모든 부품식들마다 번호를 붙이자.



부품식 i가 가져야 하는 타입 α_i와 변수 x가 가져야 하는 타입 α_x에 대한 방정식 "$lhs \doteq rhs$"는 다음과 같다:

$$
\begin{array}{rclcrcl}
\alpha_0 & \doteq & \iota & & \alpha_1 & \doteq & \iota \\
\alpha_2 & \doteq & \iota & & \alpha_3 & \doteq & \alpha \to \beta \\
\alpha & \doteq & \alpha_x & & \beta & \doteq & \alpha_5 \\
\alpha & \doteq & \alpha_4 & & \beta & \doteq & \alpha_2 \\
\alpha_4 & \doteq & \iota & & \alpha_5 & \doteq & \alpha_2 \\
\alpha_6 & \doteq & \alpha_x & & \alpha_6 & \doteq & \alpha' \to \beta' \\
\alpha' & \doteq & \alpha_7 & & \beta' & \doteq & \alpha_2 \\
\alpha_7 & \doteq & \iota & & & &
\end{array}
$$

위의 방정식의 해는 없다. α_x는 ι 이기도 해야 하고 ($\alpha \doteq \alpha_x$로 부터) $\iota \to \iota$이기도 해야 하므로($\alpha_6 \doteq \alpha_x$로 부터). □

타입 연립 방정식 u는 다음과 같은 모습이다:

$$
\begin{array}{llcll}
\mathit{TyEqn} & u & \to & \tau \doteq \tau & \text{타입 방정식} \\
 & & \mid & u \wedge u & \text{연립} \\
\mathit{Type} & \tau & \to & \alpha \in \mathit{TyVar} & \text{타입 변수} \\
 & & \mid & \iota \mid \tau \to \tau &
\end{array}
$$

타입 연립 방정식의 해 S를 *치환*$_{substitution}$으로 표현하면 편하다. TyVar을

*Type*으로 바꾸는. 즉,

$$S \in TyVar \stackrel{\text{fin}}{\hookrightarrow} Type$$

인. ($A \stackrel{\text{fin}}{\hookrightarrow} B$는 A 에서 B로가는 치환들의 집합이다.)

Notation 9 $S \models u$은 "S는 방정식 u의 해*model*"라는 뜻이다. 다음과 같이 정의된다:

$$\frac{S\tau_1 = S\tau_2}{S \models \tau_1 \doteq \tau_2} \qquad \frac{S \models u_1 \quad S \models u_2}{S \models u_1 \ \wedge \ u_2} \qquad \frac{S \models \{S(\alpha)/\alpha\}u}{S \models \exists\alpha.u}$$

여기서

$$\begin{array}{rcl} S\alpha & = & \begin{cases} \tau & \text{if } \alpha \mapsto \tau \ \in S \\ \alpha & \text{otherwise} \end{cases} \\ S\iota & = & \iota \\ S(\tau \to \tau') & = & (S\tau) \to (S\tau') \\ S\Gamma & = & \{x \mapsto S\tau \mid x \mapsto \tau \in \Gamma\} \end{array}$$

□

연립 방정식을 세우는 알고리즘 $V(\Gamma, e, \tau)$ 는 다음과 같다:

$$\begin{array}{rcll} V(\Gamma, n, \tau) & = & \tau \doteq \iota & \\ V(\Gamma, x, \tau) & = & \tau \doteq \tau' & \text{if } x : \tau' \in \Gamma \\ V(\Gamma, E_1 \texttt{+} E_2, \tau) & = & \tau \doteq \iota \ \wedge \ V(\Gamma, E_1, \iota) \ \wedge \ V(\Gamma, E_2, \iota) & \\ V(\Gamma, \lambda x.E, \tau) & = & \tau \doteq \alpha_1 \to \alpha_2 \ \wedge \ V(\Gamma + x : \alpha_1, e, \alpha_2) & \text{new } \alpha_1, \alpha_2 \\ V(\Gamma, E_1 \ E_2, \tau) & = & V(\Gamma, E_1, \alpha \to \tau) \ \wedge \ V(\Gamma, E_2, \alpha) & \text{new } \alpha \end{array}$$

V Example

$$V(\emptyset, (\lambda x.x)(\lambda x.x), \tau) =$$

$$\exists\alpha_1.\ V(\emptyset, \lambda x.x, \alpha_1 \to \tau) \land V(\emptyset, \lambda x.x, \alpha_1)$$

$$= \exists\alpha_1.(\exists\alpha_2,\alpha_3.\ V(x:\alpha_2, x, \alpha_3) \land \alpha_2 \to \alpha_3 \doteq \alpha_1 \to \tau \quad \exists\alpha_4,\alpha_5.\ V(x:\alpha_4, x, \alpha_5) \land \alpha_1 \doteq \alpha_4 \to \alpha_5)$$

$$= \exists\alpha_1 (\exists\alpha_2,\alpha_3,\ \alpha_2 \doteq \alpha_3 \land \alpha_2 \to \alpha_3 \doteq \alpha_1 \to \tau \quad \exists\alpha_4,\alpha_5.\ \alpha_4 \doteq \alpha_5 \land \alpha_1 \doteq \alpha_4 \to \alpha_5)$$

$$= \exists\alpha_1\alpha_2\alpha_3\alpha_4\alpha_5.\ (\alpha_2 \doteq \alpha_3 \land \alpha_2 \to \alpha_3 \doteq \alpha_1 \to \tau \land \alpha_4 \doteq \alpha_5 \land \alpha_1 \doteq \alpha_4 \to \alpha_5)$$

A system of equations to be solved.

6

$V(\Gamma, e, \tau)$는 옳은가? 즉,

$$S \models V(\Gamma, E, \tau) \quad \Leftrightarrow \quad S\Gamma \vdash E : S\tau$$

인가?

Proof. E의 구조에 대한 귀납법으로.

$\lambda x.E$인 경우: $S \models V(\Gamma, \lambda x.E, \tau)$ 은

$$
\begin{array}{ll}
= & S \models \tau \doteq \alpha_1 \to \alpha_2 \ \wedge\ V(\Gamma + x:\alpha_1, e, \alpha_2) \quad \text{new } \alpha_1, \alpha_2 \\
\Leftrightarrow & S \models \tau \doteq \alpha_1 \to \alpha_2 \\
& \wedge\ S \models V(\Gamma + x:\alpha_1, e, \alpha_2) \\
\Leftrightarrow & S\tau = S\alpha_1 \to S\alpha_2 \\
& \wedge\ S\Gamma + x : S\alpha_1 \vdash E : S\alpha_2 \quad (\text{귀납가정}) \\
\Leftrightarrow & S\tau = S\alpha_1 \to S\alpha_2 \\
& \wedge\ S\Gamma \vdash \lambda x.E : S\alpha_1 \to S\alpha_2 \\
\Leftrightarrow & S\Gamma \vdash \lambda x.E : S\tau.
\end{array}
$$

다른 경우도 비슷하게.□

5.7.3.2 연립 방정식의 해

우리가 세우는 타입 연립 방정식을 어떻게 풀 수 있을까?

우선 연립 방정식의 해는 우리가 정의한 *Type*이라는 집합의 원소여야 한다. 그 집합은 두 가지 방식으로 귀납적으로 유한하게 만들어지는 τ들의 무한한 집합이다.

$$
\begin{array}{llll}
\mathit{Type} & \tau & \to & \iota \qquad\qquad \text{primitive type} \\
& & \mid & \tau \to \tau \quad \text{function type}
\end{array}
$$

그 해는, *동일화 알고리즘unification algorithm*을 이용해서 풀 수 있다.

> "A Machine-Oriented Logic Based on the Resolution PrinciplE", J.A.Robinson, *Journal of ACM*, Vol.12, No.1, pp.23-41, 1965.

- 타입 방정식들($\tau \doteq \tau'$)과 해 공간(*Type*)은 위 논문의 *동일화unification* 알고리즘으로 풀 수 있는 클래스이다.
- 알고리즘 $\mathcal{U}$는 $T \models u$인 T 중에서 *가장 일반적인 해most general unifier*를 구해준다.

– 알고리즘 $\mathcal{U}$가 방정식 u에 대해서 해 S를 만들어 냈으면

– u의 다른 해 T 는 항상 R로 부터 나온다: $T \models u$ 이면 $T = RS$ 인 R이 있다.

따라서, 주어진 프로그램 (자유변수가 없는 식) E의 타입 방정식을 푸는 알고리즘

$$\mathcal{U} : \mathit{TyEqn} \to (\mathit{TyVar} \overset{\text{fin}}{\hookrightarrow} \mathit{Type})$$

는

$$\mathcal{U}(V(\emptyset, E, \alpha)) \qquad (\text{new } \alpha)$$

을 실행하는 것이고, $\mathcal{U}$의 정의는

$$\mathcal{U}(u) = \texttt{unify-all}(u, \emptyset).$$

여기서

$$\begin{array}{rcl}
\texttt{unify-all}(\tau \doteq \tau', S) & = & (\texttt{unify}(\tau, \tau'))S \\
\texttt{unify-all}(u \wedge u', S) & = & \text{let } T = \texttt{unify-all}(u, S) \text{ in } \texttt{unify-all}(Tu', T)
\end{array}$$

이며, 동일화*unification* 알고리즘

$$\texttt{unify}(\tau, \tau') : \mathit{TyVar} \overset{\text{fin}}{\hookrightarrow} \mathit{Type}$$

의 정의는

$$
\begin{array}{rcl}
\texttt{unify}(\iota,\iota) & = & \emptyset \\
\texttt{unify}(\alpha,\tau) \ \mathit{or}\ \texttt{unifty}(\tau,\alpha) & = & \{\alpha \mapsto \tau\} \quad \text{if } \alpha \text{ does not occur in } \tau \\
\texttt{unify}(\tau_1 \to \tau_2, \tau_1' \to \tau_2') & = & \text{let } S = \texttt{unify}(\tau_1,\tau_1') \\
 & & \quad\ \ S' = \texttt{unify}(S\tau_2, S\tau_2') \\
 & & \text{in } SS' \\
\texttt{unify}(_) & = & \text{fail}
\end{array}
$$

이다.

위의 알고리즘은 우리의 단순 타입 시스템*simple type system*의 충실한*sound & complete* 구현이다.

$$
\text{안전}\mathit{sound}\text{:} \qquad \mathcal{U}(V(\Gamma,e,\alpha)) = S \quad \text{이면} \quad S\Gamma \vdash E : S\alpha
$$

$$
\text{완전}\mathit{complete}\text{:} \quad \left.\begin{array}{l} \mathcal{U}(V(\Gamma,e,\alpha)) = S \\ \wedge\ \Gamma' = RS\Gamma \\ \wedge\ \tau' = RS\alpha \end{array}\right\} \ \text{면이} \quad \Gamma' \vdash E : \tau'
$$

Type Inference

Determine, for given Γ and e
the τ such that $\Gamma \vdash e : \tau$.

Use the same procedure $\mathcal{V}(\Gamma, e, \alpha)$
but with a type var α.

Thm Let $\mathcal{V}(\Gamma, e, \alpha) = \exists \alpha_1, \cdots, \alpha_n. \overline{\mathcal{V}(\Gamma, e, \alpha)}$

Either

① not $\vdash \exists \alpha. \mathcal{V}(\Gamma, e, \alpha)$ or

② $\exists$ principal substitution
$S \in \{\alpha, \alpha_1, \cdots, \alpha_n\} \rightarrow \mathit{TypesWithVars}$
such that
$\vdash S(\overline{\mathcal{V}(\Gamma, e, \alpha)})$

- Principal S
  if $\vdash S'(\overline{\mathcal{V}(\Gamma, e, \alpha)})$ then $\exists T$ s.t. $S' \doteq TS$.
- TypesWithVars $\tau \rightarrow \iota \mid \tau \rightarrow \tau \mid \alpha$ (type variable)

pf) Robinson's Unification Theorem & Resolution Principle

5.7.3.3 온라인 알고리즘들

타입 방정식을 도출하면서 그때 그때 풀어가는 알고리즘도 가능하다.

$$M : \mathit{TyEnv} \times \mathit{Exp} \times \mathit{Type} \rightarrow (\mathit{TyVar} \xrightarrow{\text{fin}} \mathit{Type})$$

$$
\begin{array}{rcl}
M(\Gamma, n, \tau) & = & \text{unify}(\iota, \tau) \\
M(\Gamma, x, \tau) & = & \text{unify}(\tau, \tau') \quad \text{if } x : \tau' \in \Gamma \\
M(\Gamma, \lambda x.E, \tau) & = & \text{let } S = \text{unify}(\alpha_1 \to \alpha_2, \tau) \quad \text{new } \alpha_1, \alpha_2 \\
& & \qquad S' = M(S\Gamma + x : S\alpha_1, e, S\alpha_2) \\
& & \text{in } S'S \\
M(\Gamma, E\ E', \tau) & = & \text{let } S = M(\Gamma, e, \alpha \to \tau) \quad \text{new } \alpha \\
& & \qquad S' = M(S\Gamma, E', S\alpha) \\
& & \text{in } S'S \\
M(\Gamma, E + E', \tau) & = & \text{let } S = \text{unify}(\iota, \tau) \\
& & \qquad S' = M(S\Gamma, e, \iota) \\
& & \qquad S'' = M(S'S\Gamma, E', \iota) \\
& & \text{in } S''S'S
\end{array}
$$

위의 알고리즘도 단순 타입 시스템*simple type system*의 충실한*sound & complete* 구현이다.

$$
\begin{array}{lll}
\text{안전}\textit{sound}: & M(\Gamma, e, \alpha) = S \ \text{ 이면 } & S\Gamma \vdash E : S\alpha \\
\text{완전}\textit{complete}: & \left.\begin{array}{l} M(\Gamma, e, \alpha) = S \\ \wedge\ \Gamma' = RS\Gamma \\ \wedge\ \tau' = RS\alpha \end{array}\right\} \text{ 면이 } & \Gamma' \vdash E : \tau'
\end{array}
$$

혹은, 약간 다르게.

$$
W : \textit{TyEnv} \times \textit{Exp} \to \textit{Type} \times (\textit{TyVar} \xrightarrow{\text{fin}} \textit{Type})
$$

$$
\begin{array}{rcl}
W(\Gamma, n) & = & (\iota, \emptyset) \\
W(\Gamma, x) & = & (\tau, \emptyset) \quad \text{if } x \mapsto \tau \in \Gamma \\
W(\Gamma, \lambda x.E) & = & \text{let } (\tau, S) = W(\Gamma + x : \alpha, e) \quad \text{new } \alpha \\
& & \text{in } (S\alpha \to \tau, S) \\
W(\Gamma, E\ E') & = & \text{let } (\tau, S) = W(\Gamma, e) \\
& & \quad (\tau', S') = W(S\Gamma, E') \\
& & \quad S'' = \text{unify}(\tau' \to \alpha, S'\tau) \quad \text{new } \alpha \\
& & \text{in } (S''S'S, S''\alpha) \\
W(\Gamma, E + E') & = & \text{let } (\tau, S) = W(\Gamma, e) \\
& & \quad S' = \text{unify}(\tau, \iota) \\
& & \quad (\tau', S'') = W(S'S\Gamma, E') \\
& & \quad S''' = \text{unify}(\tau', \iota) \\
& & \text{in } (\iota, S'''S''S'S)
\end{array}
$$

위의 알고리즘도 *단순 타입 시스템simple type system*의 *충실한sound & complete* 구현이다.

$$
\begin{array}{rlll}
\text{안전}\textit{sound}: & W(\Gamma, e) = (\tau, S) & \text{이면} & S\Gamma \vdash E : \tau \\
\text{완전}\textit{complete}: & \left.\begin{array}{l} W(\Gamma, e) = (\tau, S) \\ \wedge\ \Gamma' = RS\Gamma \\ \wedge\ \tau' = R\tau \end{array}\right\} & \text{면이} & \Gamma' \vdash E : \tau'
\end{array}
$$

5.7.4 안전한 그러나 경직된

단순 타입 시스템이 안전하기는 하지만 아쉬움이 많다.

완전하지 못하다. 타입 시스템을 통과한 (타입 유추에 성공한) 프로그램은 실행중에 타입 오류가 발생하지 않는 다는 것이 보장된다. 하지만, 타입 오류 없이 잘 실행될 수 있는 모든 프로그램이 단순 타입 시스템을 통과하는 것은

아니다. *안전sound*하지만 *빠뜨리는 프로그램들이 있는incomplete* 것이다. 빠뜨리는게 특히 많다.

유연하지 못하다. 인자 타입에 상관없이 실행되는 함수들이 많이 있다. 그런 함수들이 단순 타입 시스템을 통과하면서는 하나의 타입으로 고정된다. 다양한 타입의 인자에 적용되는 경우가 프로그램에 있다면 단순 타입 시스템을 통과할 수 없다.

다음에 예들이 있다.

Example 30 다음의 프로그램을 보자. 단순 타입 시스템을 통과할 수 없는, 하지만 잘 실행될 수 있는 프로그램이다.□

```
(* polymorphic functions *)
let
    I = \x.x
    const = \n.10
in
    I I;
    const 1 + const true
end
```

Example 31 다음의 프로그램도 단순 타입 시스템에서는 타입 유추가 불가능한 경우다. 하지만 문제 없이 실행되는 프로그램이다. □

```
(* S K I combinators *)
let
    I = \x.x
    K = \x.(\y.x)
    S = \x.(\y.(\z.(x z)(y z)))
in
    S (K (S I)) (S (K K) I) 1 (\x.x+1)
end
```

Example 32 쌍을 다루는 언어로 단순 타입 시스템이 확장된다고 해도, 타입 시스템을 통과하지 못하는 유용한 프로그램을 쉽게 만나게 된다. □

```
(* polymorphic swap *)
let
   swap =
        \order_pair.
           if order_pair.1 order_pair.2
           then order_pair.2
           else (order_pair.2.2, order_pair.2.1)
in
   swap(\pair.(pair.1 + 1 = pair.2), (1,2));
   swap(\pair.(pair.1 or pair.2), (true, false))
end
```

5.7.5 M3로 확장하기

5.7.5.1 재귀 함수

재귀 함수는 *Y 컴비네이터Y combinator*를 통해서 표현할 수 있었다. Y 컴비네이터는

$$\lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

이고, 그 핵심에는 자신을 인자로 받는 함수의 적용 "$x\,x$"이 있다.

단순 타입 시스템에서는 함수가 자신을 인자로 받는 식은 타입 유추가 될 수 없다:

$$\underbrace{\underbrace{x}_{1}\ \underbrace{x}_{2}}_{0}$$

에서 각 부품식 i가 가져야 하는 타입 α_i를 따져보면

$$\begin{array}{rclcrcl} \alpha_1 & \doteq & \alpha_x & \quad & \alpha_1 & \doteq & \alpha \to \beta \\ \alpha_2 & \doteq & \alpha_x & & \alpha_2 & \doteq & \alpha \end{array}$$

이 되는데, 결국 $\alpha \to \beta \doteq \alpha$를 만족하는 타입을 *Type*에서는 찾을 수 없다. 동일화*unification* 알고리즘은 위의 양변을 같게 만드는 α와 β를 찾는것에 실패한다.

따라서, 단순 타입 시스템이 갖추어진 언어에서는 재귀함수를 Y 컴비네이

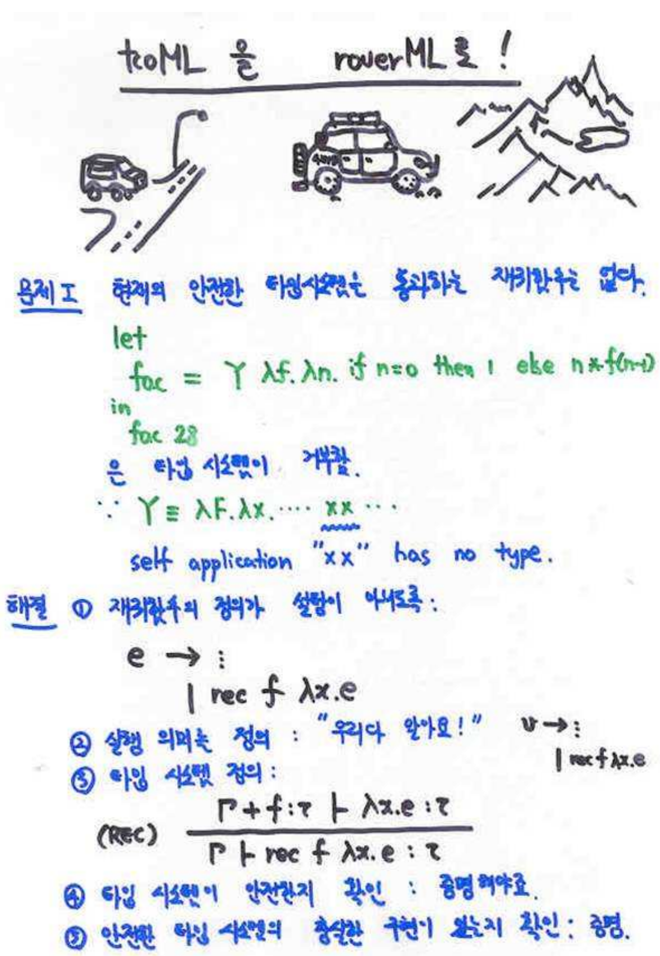
터로 표현해서는 타입 시스템을 통과할 수 없다. 재귀함수

$$\texttt{rec}\ f\ \lambda x.E$$

는 더 이상 설탕이 아니다.

타입 유추 규칙은 간단하다:

$$\frac{\Gamma + f : \tau_1 \rightarrow \tau_2 \vdash \lambda x.E : \tau_1 \rightarrow \tau_2}{\Gamma \vdash \texttt{rec}\ f\ \lambda x.E : \tau_1 \rightarrow \tau_2}$$



Exercise 4 안전성*soundness* 증명도 위의 경우에 대해서 기존의 얼개 (Progress Lemma와 Subject Reduction Lemma)안에 추가하면 된다.

Exercise 5 오프라인 알고리즘과 온라인 알고리즘은 어떻게 확장될 수 있을

까?

5.7.5.2 메모리 주소 값

메모리 주소를 값으로 사용하는 경우는 타입 시스템을 어떻게 확장할 수 있을까?

$$
\begin{array}{rcl}
E & \rightarrow & \vdots \\
 & | & \texttt{malloc}\, E \\
 & | & \texttt{!}E \\
 & | & E \;\texttt{:=}\; E \\
 & | & E \;;\; E
\end{array}
$$

우선 타입의 종류가 하나는다. 메모리 주소 값의 타입들.

$$
\begin{array}{llcll}
\mathit{Type} & \tau & \rightarrow & \iota & \text{primitive type} \\
 & & | & \tau \rightarrow \tau & \text{function type} \\
 & & | & \tau\ \mathit{loc} & \text{pointer type}
\end{array}
$$

타입 $\tau\ \mathit{loc}$는 메모리 주소 타입인데, 그 주소에는 τ 타입의 값이 저장되는 타입이다. 그리고, 타입 유추 규칙들이 자연스럽게 확장된다. 다음과 같다.

$\boxed{\Gamma \vdash E : \tau}$

$$\frac{\Gamma \vdash E : \tau}{\Gamma \vdash \texttt{malloc}\, E : \tau\ \mathit{loc}}$$

$$\frac{\Gamma \vdash E : \tau\ \mathit{loc}}{\Gamma \vdash \texttt{!}E : \tau}$$

$$\frac{\Gamma \vdash E_1 : \tau\ \mathit{loc} \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash E_1 \;\texttt{:=}\; E_2 : \tau}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash E_1 \;;\; E_2 : \tau_2}$$

Exercise 6 *안전성soundness* 증명도 위의 경우에 대해서 기존의 얼개 (Progress Lemma와 Subject Reduction Lemma)안에 추가하면 된다.

Exercise 7 오프라인 알고리즘과 온라인 알고리즘은 어떻게 확장될 수 있을

까?

③ 타입 시스템 정의: $\tau \to \iota \mid \tau \to \tau \mid \tau\ loc$

$$(\text{MALLOC})\quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{malloc}\ e : \tau\ loc}$$

$$(\text{ASS})\quad \frac{\Gamma \vdash e_1 : \tau\ loc \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \tau}$$

$$(\text{ACC})\quad \frac{\Gamma \vdash e : \tau\ loc}{\Gamma \vdash\ !e : \tau}$$

④ 타입 시스템이 안전한지 확인: 증명.

⑤ 안전한 타입 시스템의 충실한 구현이 맞는지 확인: 증명.

$$\begin{aligned}
&\nabla(\Gamma, \text{malloc}\ e, \tau) \\
&\quad = \exists\alpha.\ \tau \doteq \alpha\ loc \\
&\qquad \wedge \nabla(\Gamma, e, \alpha) \\
&\nabla(\Gamma, e_1 := e_2, \tau) \\
&\quad = \nabla(\Gamma, e_1, \tau\ loc) \wedge \nabla(\Gamma, e_2, \tau) \\
&\nabla(\Gamma, !e, \tau) \\
&\quad = \nabla(\Gamma, e, \tau\ loc)
\end{aligned}$$

5.8 다형 타입 시스템*polymorphic type system*

단순 타입 시스템*simple type system*을 정교하게 확장하자. 안전성을 유지하면서 "잘 모르겠다"고 하는 경우를 줄여보자. 안전하면서 유연한 타입 시스템이 필요하다.

다형 타입 시스템*polymorphic type system*이 그러한 타입 시스템이다. 특히, 타입 유추가 가능하고 실제 언어에 실용적으로 구현될 수 있는 다형 타입 시스템이 *let*-다형 타입 시스템*let-polymorphic type system*이다.

VI.2. Polymorphic Type Systems

to learn:
- let-polymorphic type system
  : implicit polymorphism
- polymorphic lambda calculus
  : system F
  : explicit polymorphism

History (à la Reynolds @ POPL '98)

Typed Set Theory	(Russell)	1908
Simply Typed Lambda Calculus	(Church)	1940
Type Inference	(Hindley)	1969
System F	(Girard)	1971
Polymorphic Lambda Calculus	(Reynolds)	1974
ML	(Milner et.al)	1977
Subtypes	(Reynolds)	1980
Existential Abstract Types	(Mitchell, Plotkin)	1985
Bounded Polymorphism	(Cardelli, Wegner)	1985
Dependent Types & Modularity	(MacQueen)	1986
Linear Types	(Girard)	1987
SML	(Milner et.al)	1990
Typed Assembly Language	(Morrisett et.al)	1998

6-18

타입 시스템에서는

$$\text{“잘 모르겠다”} = \text{“타입방정식의 해가 없다”}$$

이다. *다형 타입 시스템polymorphic type system*($\vdash_p$)에서는 “잘 모르겠다”고 하는 경우가 단순 타입 시스템*simple type system*($\vdash$) 보다 적다. 즉,

$$\Gamma \vdash E : \tau \quad \text{이면} \quad \Gamma \vdash_p E : \tau$$

이다. 이런 경우를 $\vdash_p$는 $\vdash$를 *포섭하면서 확장conservative extension*한 경우라고 한다.

*다형 타입 시스템polymorphic type system*에서는 유추되는 타입이 *다형 타입polymorphic*

*type*일 수 있다. 다형 타입들은 다음과 같은 것들이다:

$$\forall\alpha.\alpha \to \iota, \quad \forall\alpha_1\alpha_2.\alpha_1 \to \alpha_2, \quad \cdots$$

$\forall$이 타입 변수에 붙어서 표현되는 것이다. 모든 단순 타입의 경우를 모두 포섭하는 타입이라는 뜻이다. 즉, 다형 타입의 의미 $[\![\forall\alpha.\tau]\!]$는 다음과 같은 값들의 집합이라고 볼 수 있다:

- 모든 타입 α에 대해서 τ 타입인 값들.
- 예를들어, $\forall\alpha.\alpha \to \alpha$의 의미는:

$$[\![\forall\alpha.\tau]\!] = \bigcap_{t \in SimpleType} [\![\{\alpha \mapsto t\}\tau]\!]$$

예를들어,

$$\begin{array}{rcl} [\![\forall\alpha.\alpha \to \alpha]\!] & = & \bigcap_{t \in SimpleType} [\![t \to t]\!] \\ & = & [\![\iota \to \iota]\!] \cap [\![bool \to bool]\!] \cap \cdots \\ & = & \{\lambda x.x, \lambda x.1, \lambda x.x + 1, \cdots\} \cap \cdots \\ & & \{\lambda x.x, \cdots\} \end{array}$$

단순 타입 시스템에서 실패한 경우가 어떻게 다형 타입 시스템에서는 넘어갈 수 있는 지 예를 들어 보자. 아래 두 경우는 단순 타입 시스템에서는 타입 유추될 수 없는 경우이다.

$$\dfrac{\dfrac{\dfrac{\vdots}{\{f : \tau \to \tau'\} \vdash f : \tau \to \tau'} \quad \dfrac{\vdots}{\{f : \tau \to \tau'\} \vdash f : \tau} \; \tau = \tau \to \tau'}{\{f : \tau \to \tau'\} \vdash f\ f : \tau'}}{\vdash \lambda f.f\ f : (\tau \to \tau') \to \tau'}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\{f:\tau\to\tau\}\vdash f:\tau\to\tau}\quad\dfrac{\vdots}{\{f:\tau\to\tau\}\vdash f:\tau}\ {\color{red}\tau=\tau\to\tau}}{\{f:\tau\to\tau\}\vdash f\,f:\tau\to\tau}}{\vdash\lambda f.f\,f:(\tau\to\tau)\to(\tau\to\tau)}\qquad\vdots}{\vdash(\lambda f.f\,f)(\lambda x.x):\tau\to\tau}$$

다형 타입을 이용해서 위의 두 예제를 타입 유추할 수 있게 된다:

$$\dfrac{\dfrac{\dfrac{\vdots}{\{f:\forall\alpha.\alpha\to\alpha\}\vdash f:(\iota\to\iota)\to(\iota\to\iota)}\quad\dfrac{\vdots}{\{f:\forall\alpha.\alpha\to\alpha\}\vdash f:\iota\to\iota}}{\{f:\forall\alpha.\alpha\to\alpha\}\vdash f\,f:\iota}}{\vdash\lambda f.f\,f:(\forall\alpha.\alpha\to\alpha)\to\iota}$$

f의 타입이 다형 타입 $\forall\alpha.\alpha\to\alpha$이 필요에 따라서 $(\iota\to\iota)\to(\iota\to\iota)$타입으로도 사용되고, $\iota\to\iota$타입으로도 사용된다. 아래의 경우도 마찬가지다.

$$\dfrac{\dfrac{\vdots}{\vdash\lambda f.f\,f:(\forall\alpha.\alpha\to\alpha)\to(\iota\to\iota)}\quad\dfrac{\vdots}{\vdash\lambda x.x:\iota\to\iota}}{\vdash(\lambda f.f\,f)(\lambda x.x):\iota\to\iota}$$

그러나 조심해야 한다. 항상 타입 시스템의 확장은 안전성을 유지해야 한다. 함부로 일반화를 하면 안전하지 않게 된다. 다음의 타입 추론 과정을 보자. 함부로 일반화해서 불안전*unsound*해 지는 경우이다. 아래의 예와 같이, 타입 오류가 있는 프로그램이지만 타입이 유추된다.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\{f:\forall\alpha.\alpha\to\iota\}\vdash f:\iota\to\iota}\quad\cdots}{\{f:\forall\alpha.\alpha\to\iota\}\vdash f\,1:\iota}\quad\cdots}{\{f:\forall\alpha.\alpha\to\iota\}\vdash(f\,1,f\,\mathrm{true}):\iota\times\iota}}{\vdash\lambda f.(f\,1,f\,\mathrm{true}):(\forall\alpha.\alpha\to\iota)\to(\iota\times\iota)}\quad\dfrac{\vdots}{\vdash\lambda x.x+1:\iota\to\iota}}{\vdash(\lambda f.(f\,1,f\,\mathrm{true}))(\lambda x.x+1):\iota\times\iota}$$

안전하지 않은 또 다른 예는 다음의 프로그램이다. y의 타입이 다형타입으로

일반화되면 타입 유추에 성공한다. 그러나 타입 오류가 있는 프로그램이다.

$$\frac{\vdots}{\vdash (\lambda x.(\texttt{let}\ y = x\ \texttt{in}\ (y\,1, y\,\text{true})))(\lambda z.z + 1) : \iota \times \mathit{bool}}$$

5.8.1 다형 타입 추론 규칙

Hindley-Milner 스타일의 *let-다형 타입let-polymorphism* 추론 규칙이다.

- 프로그램이 특별히 생긴 경우만 그렇게 정교한 분석이 작동하도록 한다.

- 함수가 어디서 무슨 인자로 어떻게 사용되는 지를 알 수 있는 경우 즉,

  $$(\lambda x.\underbrace{\cdots x \cdots x \cdots}_{E'})E$$

  즉,

  $$\texttt{let}\ x = E\ \texttt{in}\ E'$$

  인 경우만 x의 타입을 다형 타입으로 일반화 시킨다.

- 이 경우, E가 다형 타입일 수 있는 지 "안전하게" 분석한 후에, E'안에서 x가 어떻게 사용되는 지 유추해 간다. 이때 x의 타입이 다양한 다른 타입으로 사용될 수 있다.

- 다형타입은 *1단rank-1 polymorphism*까지만 이용한다:

  $$\iota \rightarrow \iota, \forall\alpha.\alpha \rightarrow \alpha, \forall\alpha_1\alpha_2.\alpha_1 \rightarrow \alpha_2$$

  즉, $\forall$*이 제일 바깥에 붙는 타입prenex form*만 사용한다.

Let- polymorphism

- let $x = e_1$ in e_2 v.s. $(\lambda x.e_2)e_1$

let-expression is included to allow less restricted static semantics.

FACT 1. The static semantics was sound w.r.t. the dynamic semantics.

FACT 2. Some programs that "runs" have no static semantics.
e.g.) $(\lambda y.y\,y)(\lambda x.x)$
Was static semantics too restrictive?

An approach to alleviate FACT 2 is let-polymorphism : combination of syntax and static semantics.

6-19

*let-*다형 타입 시스템*let-polymorphic type system*에서는 let-식이 설탕이 아니라 필수다.

$$
\begin{array}{rcl}
E & \to & n \\
& | & x \\
& | & \lambda x.E \\
& | & E\ E \\
& | & \texttt{let}\ x = E\ \texttt{in}\ E \\
& | & \texttt{rec}\ f\ \lambda x.E \\
& | & E + E
\end{array}
$$

타입*type*과 타입틀*type scheme*은 다음과 같다.

$$
\begin{array}{llcll}
\textit{Type} & \tau & \rightarrow & \iota & \text{primitive type} \\
 & & \mid & \tau \rightarrow \tau & \text{function type} \\
 & & \mid & \alpha & \text{type variable} \\
\textit{TypeScheme} & \sigma & \rightarrow & \tau & \text{simple type} \\
 & & \mid & \forall \alpha.\sigma & \text{generalized type}
\end{array}
$$

타입틀*type scheme*은 단순타입과 다형타입을 포함한다.

Notation 10 $\vec{\alpha}$ 는 $\{\alpha_1, \cdots, \alpha_n\}$를, $\forall\vec{\alpha}.\tau$ 는 $\forall\alpha_1 \cdots \alpha_n.\tau$를 뜻한다.

타입틀*type scheme* $\sigma = \forall\vec{\alpha}.\tau$에 대해서 $\mathit{ftv}(\sigma)$ 는 σ에 있는 자유로운 타입 변수들 $\mathit{ftv}(\tau) \setminus \vec{\alpha}$ 이다.

타입 τ에 대해서 $\mathit{ftv}(\tau)$는 τ에 있는 타입 변수들의 집합이다. 타입 환경 Γ에 대해서, $\mathit{ftv}(\Gamma) = \bigcup_{x \in \mathrm{Dom}\,\Gamma} \mathit{ftv}(\Gamma(x))$.

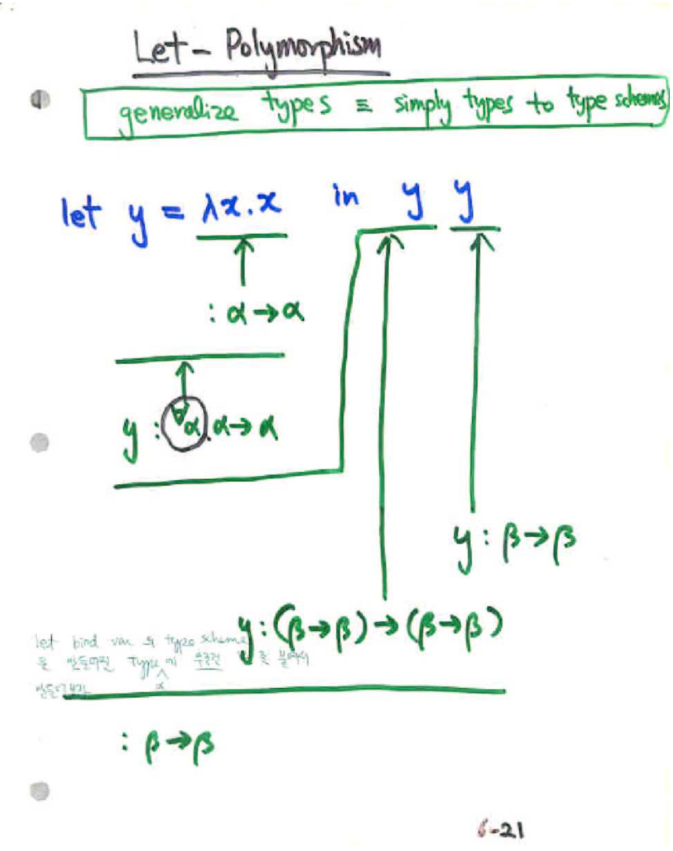
타입 치환 S에 대해서 $\mathit{itv}(S) = \{\alpha \mid \beta \in \mathit{supp}(S), \alpha \in \{\beta\} \cup \mathit{ftv}(S\beta)\}$이다.

타입 치환 S 과 타입틀*type scheme* σ에 대해서 $S\sigma = \forall\vec{\beta}.S\{\vec{\alpha} \mapsto \vec{\beta}\}\tau$이고, 이때 $\vec{\beta} \cap (\mathit{itv}(S) \cup \mathit{ftv}(\sigma)) = \emptyset$ 인 경우다.

타입 치환 S 과 타입 환경 Γ에 대해서, $S\Gamma = \{x \mapsto S\sigma \mid x \mapsto \sigma \in \Gamma\}$ 이다.

$\forall\vec{\alpha}.\tau' \succ \tau$는 타입틀의 한 예가 τ가 된다는 의미이다. 정확히는, 타입 치환 S 가 있어서 $S\tau' = \tau$ 이고 $\mathit{Supp}(S) \subseteq \vec{\alpha}$ 인 경우를 뜻한다.

$\mathit{GEN}_\Gamma(\tau) = \forall\vec{\alpha}.\tau$ 를 뜻하고 $\vec{\alpha} = \mathit{ftv}(\tau) \setminus \mathit{ftv}(\Gamma)$인 경우이다. □



let-다형 타입 시스템*let-polymorphic type system*의 타입 추론 규칙은 다음과 같다.

추론규칙*inference rules*은 "$\Gamma \vdash E : \tau$"를 유추하는 규칙들이다. Γ는 변수들의 타입틀*type scheme*에 대한 가정을 가지고 있다:

$$\Gamma \in Id \xrightarrow{\text{fin}} TypeScheme$$

$\boxed{\Gamma \vdash E : \tau}$

$$\frac{}{\Gamma \vdash n : \iota} \qquad \frac{}{\Gamma \vdash x : \tau}\ \sigma \succ \tau, x : \sigma\ \in \Gamma$$

$$\frac{\Gamma + x : \tau \vdash E : \tau'}{\Gamma \vdash \lambda x.E : \tau \to \tau'} \frac{\Gamma \vdash E_1 : \tau' \to \tau \quad \Gamma \vdash E_2 : \tau'}{\Gamma \vdash E_1\ E_2 : \tau}$$

$$\frac{\Gamma \vdash E : \tau \quad \Gamma + x : GEN_\Gamma(\tau) \vdash E' : \tau}{\Gamma \vdash \texttt{let}\ x \texttt{=} E\ \texttt{in}\ E' : \tau}$$

$$\frac{\Gamma + f : \tau \to \tau' \vdash \lambda x.E : \tau \to \tau'}{\Gamma \vdash \texttt{rec}\ f\ \lambda x.E : \tau \to \tau'}$$

$$\frac{\Gamma \vdash E_1 : \iota \quad \Gamma \vdash E_2 : \iota}{\Gamma \vdash E_1 \texttt{+} E_2 : \iota}$$

추론 규칙의 안전성을 위해서는 다형 타입으로 만들 때 선택적으로 해야 한다. 아무 타입이 아무 때나 다형 타입으로 일반화되서는 않된다.

왜 타입 τ를 "일반화" $\forall\vec{\alpha}.\tau$ 시키는데 α가 Γ 에 나타나면 제외해야 할까?

$$GEN_\Gamma(\tau) = \forall \alpha_1, \cdots, \alpha_n.\tau \quad \text{여기서 } \{\alpha_1, \cdots, \alpha_n\} = ftv(\tau) \setminus ftv(\Gamma)$$

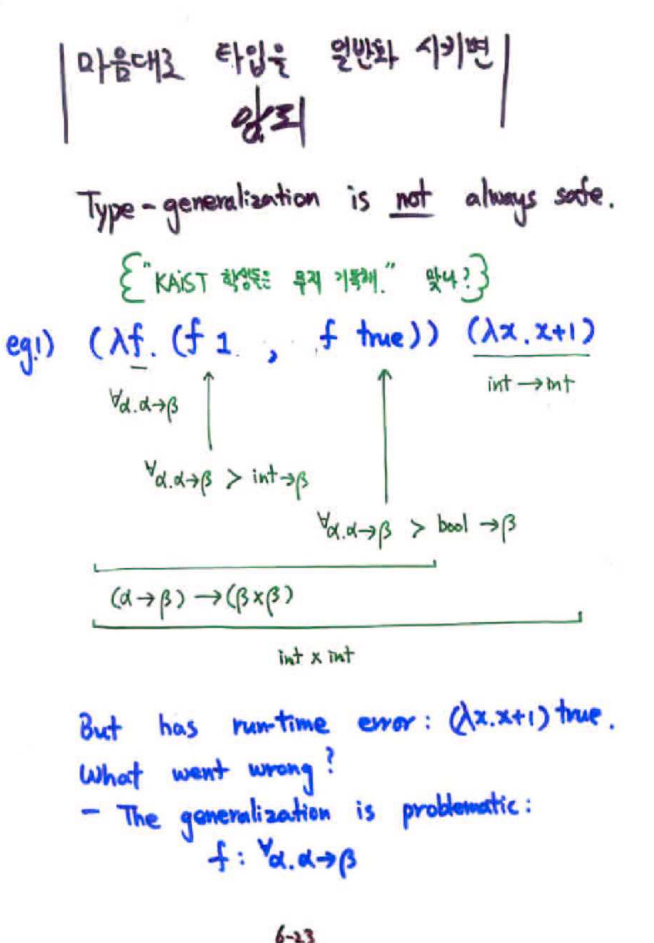
그 이유는:

- Γ에 가정$(x : \sigma)$이 첨가 되는 경우는 $\lambda x.E$의 경우밖에 없다.

- Γ에 있는 가정을 사용하는 경우는 함수안에서 함수의 인자 타입을 유추할 때이다.

- 함수의 인자 타입을 일반화시키고 나서 함수 몸통이 타입 유추되면 불안전해 진다.

- 함수가 호출되면서 전달받는 실제 인자들이 일반화될 수 있는 타입의 값이 아닐 수 있기 때문이다.



예를 들어,

$$\lambda x.(\texttt{let}\ y = x\ \texttt{in}\ (y\ 1, y\ \text{true}))$$

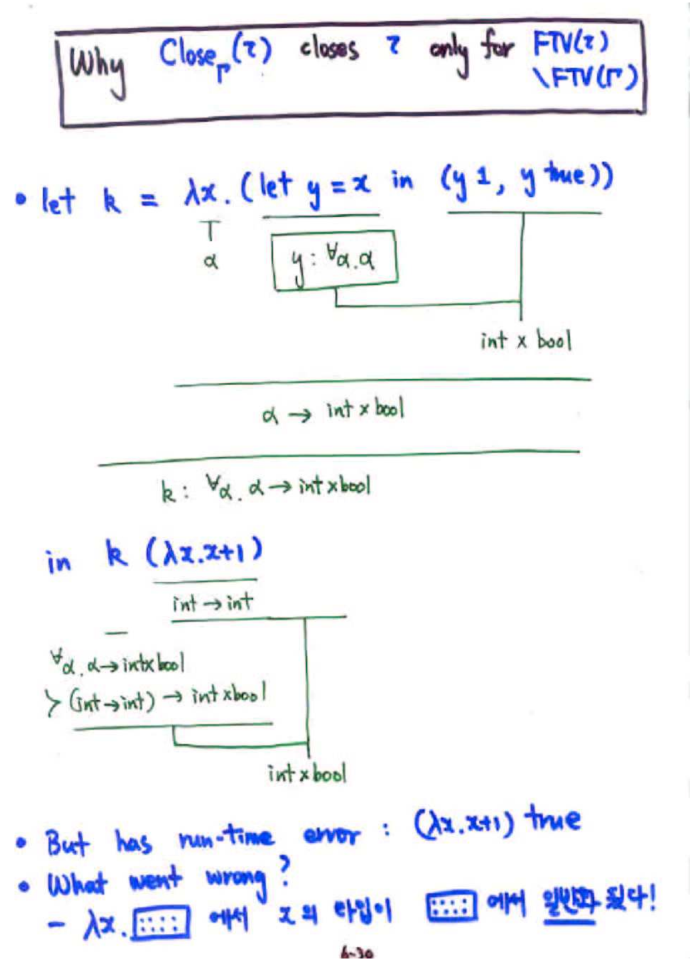
를 타입 유추하는 데, 인자 x의 타입을 다형 타입 $\forall\alpha.\alpha \to \alpha$ 으로 일반화 시키면 함수 몸통도 모두 타입 유추가 되고, 위의 함수의 최종 타입은

$$\forall\alpha.(\alpha \to \alpha) \to (int \times bool)$$

로 유추가능할 것이다. 그러나 불안전해 진다. 위의 함수가 받아들이는 인자는 위의 다형 타입을 가질 수 있는 함수가 아닐 수 있기 때문이다.

위의 함수가 바로 $\lambda z.z + 1$에 적용된다고 해도 타입 유추는 문제 없이 된다. 위의 다형 타입에서 α를 int로 구체화해서 타입 유추의 아구를 맞출 수 있기 때

문이다. 하지만 그러한 함수적용 식은 실행중 타입 오류가 발생한다.



5.8.2 추론 규칙의 안전성

단순 타입 시스템에서 증명하는 방법을 그대로 이용하는데, 다형 타입으로 확장된 것 뿐이다.

Lemma 4 (Progress) *$\vdash E : \tau$ 이고 e가 값이 아니면 반드시 진행 $e \rightarrow E'$ 한다.*

Lemma 5 (Subject Reduction, Preservation) *$\vdash E : \tau$ 이고 $E \rightarrow E'$ 이면 $\vdash E' : \tau$.*

5.8.3 추론 규칙의 구현

온라인 알고리즘

$$\mathcal{W} : \mathit{TyEnv} \times \mathit{Exp} \to ((\mathit{TyVar} \xrightarrow{\text{fin}} \mathit{Type}) \times \mathit{Type})$$

$$\begin{array}{lcl}
\mathcal{W}(\Gamma, n) & = & (\emptyset, \iota) \\
\mathcal{W}(\Gamma, x) & = & (\emptyset, \{\alpha_i \mapsto \beta_i\}_{i=1}^{n}\tau) \quad \text{where } \Gamma(x) = \forall\vec{\alpha}.\tau, \ \text{new } \vec{\beta} \\
\mathcal{W}(\Gamma, \lambda x.E) & = & \text{let} \quad (S_1, \tau_1) = \mathcal{W}(\Gamma + x\!: \beta, E), \ \text{new } \beta \\
 & & \text{in} \quad (S_1, S_1\beta \to \tau_1) \\
\mathcal{W}(\Gamma, E_1\ E_2) & = & \text{let} \quad (S_1, \tau_1) = \mathcal{W}(\Gamma, E_1) \\
 & & \qquad (S_2, \tau_2) = \mathcal{W}(S_1\Gamma, E_2) \\
 & & \qquad S_3 = \mathcal{U}(S_2\tau_1, \tau_2 \to \beta), \ \text{new } \beta \\
 & & \text{in} \quad (S_3S_2S_1, S_3\beta) \\
\mathcal{W}(\Gamma, \texttt{let}\ x \texttt{=} E_1\ \texttt{in}\ E_2) = & & \\
 & & \text{let} \quad (S_1, \tau_1) = \mathcal{W}(\Gamma, E_1) \\
 & & \qquad (S_2, \tau_2) = \mathcal{W}(S_1\Gamma + x : \mathit{GEN}_{S_1\Gamma}(\tau_1), E_2) \\
 & & \text{in} \quad (S_2S_1, \tau_2) \\
\mathcal{W}(\Gamma, \texttt{rec}\ f\ \lambda x.E) & = & \text{let} \quad (S_1, \tau_1) = \mathcal{W}(\Gamma + f\!: \beta, \lambda x.E), \ \text{new } \beta \\
 & & \qquad S_2 = \mathcal{U}(S_1\beta, \tau_1) \\
 & & \text{in} \quad (S_2S_1, S_2\tau_1)
\end{array}$$

위의 알고리즘은 단순타입 유추 알고리즘 W의 “자연스러운” 확장이다. 알고리즘 $\mathcal{W}$ 충실한 구현이다:

$$\begin{array}{rl}
\text{안전}\mathit{sound}: & W_p(\Gamma, E) = (\tau, S) \ \text{ 이면 } \ S\Gamma \vdash E : \tau \\
\text{완전}\mathit{complete}: & \left.\begin{array}{l} W_p(\Gamma, E) = (\tau, S) \\ \wedge\ \Gamma' = RS\Gamma \\ \wedge\ R(\mathit{GEN}_{S\Gamma}(\tau)) \succ \tau' \end{array}\right\} \text{ 면이 } \ \Gamma' \vdash E : \tau'
\end{array}$$

5.8.4 M3로 확장하기

메모리 주소를 값으로 사용하는 경우는 다형 타입 시스템을 어떻게 확장할 수 있을까?

$$\begin{array}{rcl} E & \to & \vdots \\ & \mid & \texttt{malloc}\,E \\ & \mid & \texttt{!}E \\ & \mid & E \texttt{ := } E \\ & \mid & E \texttt{ ; } E \end{array}$$

우선 타입의 종류가 하나 는다. 메모리 주소 값의 타입들.

$$\begin{array}{lllll} \mathit{Type} & \tau & \to & \iota & \text{primitive type} \\ & & \mid & \alpha & \text{type variable} \\ & & \mid & \tau \to \tau & \text{function type} \\ & & \mid & \tau\ \mathit{loc} & \text{pointer type} \end{array}$$

타입 틀은 그대로다:

$$\begin{array}{lllll} \mathit{TypeScheme} & \sigma & \to & \tau & \text{simple type} \\ & & \mid & \forall\alpha.\sigma & \text{generalized type} \end{array}$$

타입 $\tau\ loc$는 메모리 주소 타입인데, 그 주소에는 τ 타입의 값이 저장되는 타입이다.

타입 유추 규칙들이 자연스럽게 확장해 보자. 단순 타입 시스템의 경우와 똑 같다.

$$\frac{\Gamma \vdash E : \tau}{\Gamma \vdash \texttt{malloc}\,E : \tau\ loc}$$

$$\frac{\Gamma \vdash E : \tau\ loc}{\Gamma \vdash \texttt{!}E : \tau}$$

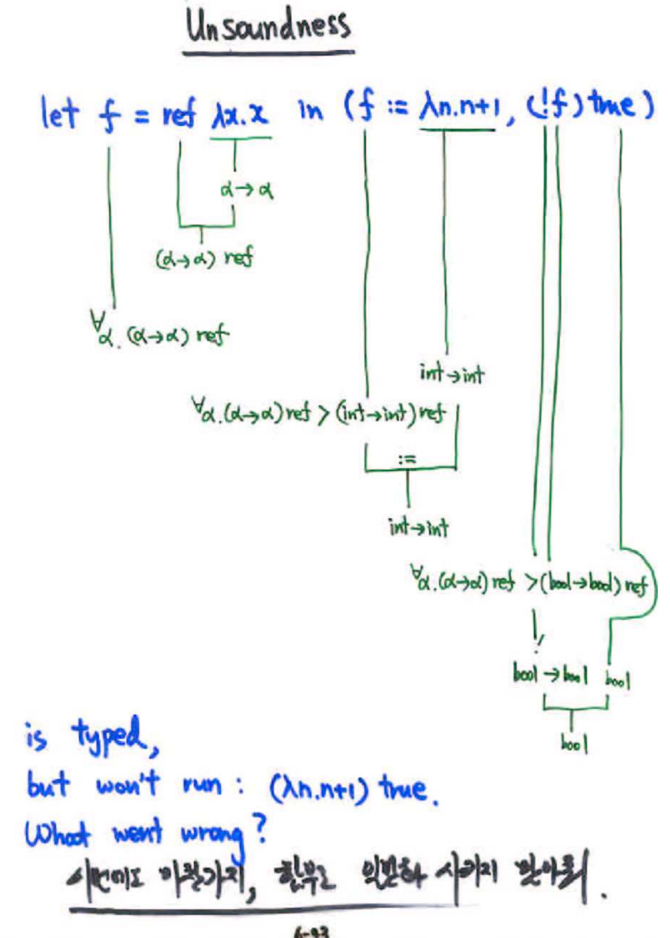
$$\frac{\Gamma \vdash E_1 : \tau\ loc \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash E_1 \texttt{ := } E_2 : \tau}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash E_1 \ ; \ E_2 : \tau_2}$$

놀랍게도 이렇게 자연스러운 확장이 타입 일반화 과정등을 거치면서 안전성이 깨지게 된다.

다음과 같은 타입 오류가 있는 프로그램이 타입 유추가 된다.

```
let
  I = malloc \x.x
in
  I := \x.x+1;
  (!I) true
```



이번에도 단순 타입을 다형 타입으로 일반화 할 때 조심스러워야 했다:

$$\frac{\Gamma \vdash E : \tau \quad \Gamma + x : GEN_\Gamma(\tau) \vdash E' : \tau}{\Gamma \vdash \texttt{let}\ x = E\ \texttt{in}\ E' : \tau}$$

를 항상 사용할 수 있는 게 아니고, 식 E가 실행중에 메모리 주소를 새롭게 할당받는 일이 없는 경우에만 안전하다.

문제는, 다형 타입 시스템은 실행전에 타입을 유추해 내야 한다. 어느 식 E가 실행중에 메모리 주소를 할당받게 될 지를 어떻게 유추하나?

정확히는 할 수 없다. 불가능하다. 가능하다면 그것을 이용해서 *멈춰요 문제Halting Problem*를 푸는 프로그램을 정의할 수 있기 때문이다.

하지만, 안전하게는 판단할 수 있다. 메모리 주소를 할당받지 않는 다는 것이 확실한 경우에만 그 타입을 일반화 시키자. 불확실하다면 일반화 시키지 말자.

그래서 `let`-식의 타입 유추 규칙이 다음과 같이 세분화 된다:

$$\frac{\Gamma \vdash E : \tau \quad \Gamma + x : GEN_\Gamma(\tau) \vdash E' : \tau}{\Gamma \vdash \texttt{let}\ x = E\ \texttt{in}\ E' : \tau}\ \neg expansive(E)$$

$$\frac{\Gamma \vdash E : \tau \quad \Gamma + x : \tau \vdash E' : \tau}{\Gamma \vdash \texttt{let}\ x = E\ \texttt{in}\ E' : \tau}\ expansive(E)$$

여기서 $expansive(E)$의 정의가

$$\begin{array}{rcl}
expansive(n) & = & false \\
expansive(x) & = & false \\
expansive(\lambda x.E) & = & false \\
expansive(\texttt{rec}\ f\ \lambda x.E) & = & false \\
expansive(E_1\ E_2) & = & true \\
expansive(\texttt{let}\ x = E_1\ \texttt{in}\ E_2) & = & expansive(E_1) \vee expansive(E_2) \\
expansive(E_1 + E_2) & = & expansive(E_1) \vee expansive(E_2)
\end{array}$$

이면 안전하다. *expansive*(E)가 *false*이면 식 E는 실행중에 절대로 새로운 메모리 주소를 할당받지 않는다. 하지만, 식 E가 *expansive*(E) 라고 해서 E가 실행중에 반드시 메모리 주소를 새로 만드는 것은 아니다.

5.9 정리

이 장에서는 프로그래밍 언어를 제대로 디자인하는 과정을 따라가 보았다. 기계적인 계산 과정을 모두 표현할 수 있는 간단한 모델 언어인 *람다 계산법Lambda Calculus*에서 부터 출발하였다. 이 언어는 함수만 있는 언어다. 값은 오직 함수뿐이다. 그러나 함수가 *자유롭다first-class value*. 함수가 인자로 전달되기도 하고, 함수 적용의 결과로 함수가 나오기도 한다.

그 위에, 실제적인 프로그래밍이 편리하도록 여러가지 *설탕syntactic sugar*들을 첨부하였다. 정수식, 참/거짓식, 조건식, let-식, 재귀함수, *쌍pair*을 다루는 식, 그리고 메모리 주소를 사용하는 명령형 언어의 모습들. 메모리 주소를 다루는 명령형 언어의 모습도 이 언어에는 간단하고 자연스럽게 포섭된다. 실제적인 프로그래밍을 편리하게 담아낼 수 있는 언어가 된다.

이렇게 확장된 언어의 의미는 매우 간단하다. 모든 식들의 실행 결과는 어떤 값을 만들어낸다. 정수, 참/거짓, 함수, 두 값의 쌍, 혹은 메모리 주소. 의미가 간단하므로, 언어를 배우기 쉽다. 의미가 간단하므로 프로그램을 이해하기 쉽다. 간단하다는 것은 매우 상위의 언어라는 것과 상통한다. 작고 간단하지만 강력한 언어다.

이렇게 디자인 한 언어가 "제대로"인가? 이런 언어에 대해서 우리가 확인하고 싶은 것은, 이 언어로 짜여진 프로그램들의 타입 오류 검증을 안전하게 미리 자동으로 할 수 있겠느냐는 것이다.

이 목표가, 안전한 타입 시스템의 디자인과 충실한 구현 방안을 찾아내면서 달성되었다. 우선, 간단한 *단순 타입 시스템simple type system*을 디자인하였고, 그 안전성을 증명해 보았다. 그 타입 시스템의 충실한 구현도 알아보았다. 그 시스템은 프로그램에서 타입을 자동으로 유추할 수 있었고, 타입 오류가 있으면 자동으로 오류를 찾아줄 수 있었다. 프로그래머는 프로그램 텍스트에 아무

런 힌트를 첨가할 필요가 없다.

타입 시스템이 언어에 장착되면서, *설탕 구조*syntactic sugar였던 것들이 필수가 되는 경우가 있다. 예를 들어, 재귀 함수가 *Y-컴비네이터*Y combinator를 통해서 녹아버리면 단순 타입시스템에서는 타입 검증을 통과할 수 없게 된다. 재귀 함수를 표현하는 식이 언어에서 제공되면 문제가 없어진다.

이 단순 타입 시스템은 안전하기는 하지만 유연성이 많이 떨어진다. 타입 오류 없이 잘 실행되는 많은 프로그램들이 타입 시스템을 통과하지 못한다.

단순 타입 시스템을 정교하게 다듬은 *let-다형 타입 시스템*let-polymorphic type system을 알아보았다. 단순 타입 시스템으로 타입 유추 할 수 있는 모든 프로그램을 받아 들이고, 그 보다 더 많은 프로그램을 타입 유추할 수 있게 되었다. 이 타입 시스템의 안전성도 보장될 수 있고, 그 구현 방안도 알아보았다. 타입이 자동으로 유추되고 타입 오류가 자동으로 찾아진다.

타입 시스템을 정교하게 다듬으면서 (좀더 *완전한*complete 모습에 가까워지도록, 좀더 많은 제대로 실행되는 프로그램을 받아들일 수 있도록), 안전성이 쉽게 깨지는 경우가 있다. 메모리 주소를 사용하는 프로그램의 다형 타입 시스템은 자연스러운 확장으로는 안전성을 보장할 수 없었다. 세심한 주의가 필요했다.

6 장

번역과 실행

프로그래밍 언어를 디자인했다면, 구현하고 싶다. 그 언어로 짜여진 프로그램을 실행시키고 싶은 것이다. 사람이 실행할 수 도 있겠다. 의미 정의를 이해하는 사람이라면. 그러나 물론 우리는 컴퓨터를 통해서 실행시키고 싶다.

프로그램을 컴퓨터의 외부에서 입력받을 수 있도록 하고, 입력된 프로그램을 실행시키는 방법을 고안해야 한다.

6.1 프로그램 입력

우선 프로그램을 컴퓨터에 입력하는 방법이 필요하다. 현재의 컴퓨터 시스템에서 입력할 수 있는 방법은 문자들의 일차원 배열밖에는 없다. 키보드 장치를 통해서 프로그램 텍스트를 일렬로 쭈욱 처가는 수 밖에는 없다. 그렇게 쓴 일차원 문자열이 프로그램의 입력형태다.

그 일차원 문자열이 어떤 생김새의 프로그램인지를 파악해 내야 한다. 이 과정이 *문법검증parsing*이라는 방법으로 자동으로 이루어 진다. (이 방법에 대해서는 이 강의에서는 다루지 않는다.) 이 과정을 마치면 입력된 프로그램이 어떤 모습의 프로그램인지 파악된다. 프로그램을 만드는 방법(문법 규칙)들이 어떻게 조합되어서 구성한 프로그램인지.

"프로그램"은 이제 그 구조가 명백한 모습으로 준비가 된다. 2차원 나무구

조로. (3.1절 참고)

6.2 프로그램 실행

입력된 프로그램을 그 의미 정의대로 실행해야 한다. 두가지 방법이 있다.

- 한 방법은 디자인한 언어 A의 *실행기interpreter*를 곧바로 만드는 것이다.

  A 실행기에 A 프로그램을 넣어주면 그 프로그램을 실행해 줄 것이다. 그런데 그 A 실행기를 어떻게 구현할 수 있을까?

  이미 실행기가 있는 언어가 있으면 그 언어로 A 실행기를 만들면 된다. 예를들어, 디지탈 컴퓨터의 기계어는 이미 실행기가 있다. 전깃줄로 구현된 *중앙처리장치cpu*가 기계어의 *실행기interpreter*이다. 이 기계어로 A 실행기를 작성하면 된다. 그 A 실행기는 cpu가 실행해 줄 것이다. 아니면, A 실행기를 전깃줄로 구현할 수도 있다. 이 경우 그러한 cpu를 가진 컴퓨터가 A 실행기가 되는 셈이다.

- 다른 방법은 다른 언어의 실행기를 이용하는 것이다.

  이미 다른 언어의 실행기가 있다면, 그 언어로 번역해주는 *번역기compiler*를 만들면 된다. B 언어의 *실행기interpreter*가 존재한다고 하자. A 프로그램을 B 프로그램으로 번역하고 B 실행기에 넣어주면 된다.

  예를 들어, 디지탈 컴퓨터의 *중앙처리장치cpu*는 기계어의 *실행기interpreter*이다. 그 기계에서 A 언어를 실행시키는 방법은 A 프로그램을 기계어 프로그램으로 번역하는 것이다. 번역 결과를 *중앙처리장치cpu*로 실행시키면 된다.

6.2.1 *실행기interpreter*

*실행기interpreter*는 주어진 프로그램을 실행해 준다. 의미 정의가 실행기의 정의라고 볼 수 있다.

하나의 *실행기interpreter*는 대상 언어의 모든 프로그램을 실행해 줄 수 있다. 놀라운 일이다. 대상 언어로 작성할 수 있는 프로그램이 무한히 많이 있을 수 있는데, 하나의 실행기가 모두 실행해 낼 수 있다니.

이것이 가능한 이유는 뭘까? 귀납구조 때문이다. *의미 정의semantics*가 대상 언어의 모든 프로그램의 의미를 정의해 줄 수 있었던 것과 같은 이치다. 모든 임의의 프로그램은 유한한 가짓수의 조립 방법을 가지고 만들어진다. 기본 부품들이 있고, 부품을 가지고 부품을 만들고. 이 과정을 통해서 하나의 프로그램이 조립된다.

프로그램의 실행도 프로그램을 만드는 귀납 규칙을 따라 맴돈다. 실행 규칙은 프로그램을 만드는 귀납 규칙마다 하나씩 정의된다. 그래서, 프로그램이 조립된 과정을 따라서, 부품을 실행하고 실행된 부품을 조립해 간다. 기본 부품들의 실행 규칙이 있다. 귀납적인 조립 규칙마다, 실행 규칙이 있다. 실행된 부품을 가지고 어떻게 조립하면 실행된 전체가 나오는 지.

실행기가 가지는 실행 규칙은 *의미구조semantics* 정의를 그대로 따라간다.

6.2.2 *번역기compiler*

*번역기compiler*는 주어진 프로그램을 다른 언어의 같은 프로그램으로 바꾸어준다. 영어 소설을 한글 소설로 바꾸어주듯이. 대상 언어와 결과 언어의 한 쌍마다 하나의 *번역기compiler*가 필요하다.

하나의 *번역기compiler*는 번역될 대상 언어의 모든 프로그램을 결과 언어의 해당 프로그램으로 바꾸어 준다. 모든 프로그램을 바꿀 수 있다는 게 놀라운 일이다. 대상 언어로 작성할 수 있는 프로그램이 무한히 많이 있을 수 있는데, 하나의 번역기가 모두 번역해 낼 수 있다니.

이것이 가능한 이유는 뭘까? 귀납구조 때문이다. 모든 임의의 프로그램은 유한한 가짓수의 조립 방법을 가지고 만들어진다. 기본 부품들이 있고, 부품을 가지고 부품을 만들고. 이 과정을 통해서 하나의 프로그램이 조립된다.

번역은 프로그램을 만드는 귀납 규칙을 따라 맴돈다. 번역 규칙은 프로그램을 만드는 귀납 규칙마다 하나씩 정의된다. 그래서, 프로그램이 조립된 과

정을 따라서, 부품을 번역하고 번역된 부품을 조립해 간다. 기본 부품들의 번역 규칙이 있다. 귀납적인 조립 규칙마다, 번역 규칙이 있다. 번역된 부품을 가지고 어떻게 조립하면 번역된 전체가 나오는 지.

6.2.3 자가발전

참고로, 맨 처음은 어땠을까? 디지탈 컴퓨터가 있고, 그것은 "기계어"의 *실행기interpreter*를 가지고 있다. *중앙처리장치cpu*가 기계어의 *실행기interpreter*이다.

6.2.3.1 번역기의 자가발전

언어 A를 디자인했다. A를 기계어로 바꾸어주는 *번역기compiler*가 필요하다. 만들어야 한다. 방법은 오직 하나다. 기계어를 사용할 수 밖에 없다. 기계어로 A 번역기를 작성해야 한다. 처음이기 때문에 대안이 없다.

단, 처음은 작동되기만 하는 *번역기compiler*이면 된다. 효율적이고 정교한 번역결과를 내지 않아도 좋다. 같은 일을 하는 기계어로 변환해 주는 번역기이기만 하면 된다. 이 번역기가 나오면, 임의의 A 프로그램을 짜고 실행할 수 있게 된다. A 프로그램을 기계어로 번역하고, 기계에 넣어주면 된다.

자 이제, 임의의 A 프로그램을 실행할 수 있게 되었다. 그렇다면, A 번역기도 A 언어로 다시 작성할 수 있다. 이 때, A 언어가 기계어 보다는 보다 상위의 편리한 언어이므로, 좀 더 효율적인고 정교한 번역물이 나오도록 구현할 수 있을 것이다. 이렇게 A로 구성한 정교한 A 번역기도 A 프로그램이므로 물론 실행시킬 수 있다. 처음에 기계어로 얼추 만든 A 번역기보다는 번역 결과가 우수할 것이다.

6.2.3.2 실행기의 자가발전

언어 A를 디자인했다. A를 실행시켜 주는 *실행기interpreter*가 필요하다. 만들어야 한다. 방법은 오직 하나다. 기계어로 A 실행기를 작성해야 한다. 처음이기 때문에 대안이 없다.

단, 처음은 작동되기만 하는 *실행기*interpreter이면 된다. 효율적인 실행을 구현하지 않아도 좋다. 의미정의대로 실행해 주는 실행기이기만 하면 된다. 이 실행기가 나오면, 임의의 A 프로그램을 짜고 실행할 수 있게 된다. A 프로그램을 그 *실행기*interpreter에 넣어주면 된다.

자 이제는, 임의의 A 프로그램을 실행할 수 있게 되었다. 그러면, A 실행기도 A 언어로 다시 작성할 수 있다. 이 때, A 언어가 기계어 보다는 보다 상위의 편리한 언어이므로, 좀 더 효율적으로 실행하도록 구현할 수 있을 것이다. 이렇게 A로 구성된 정교한 A 실행기를 실행할 수 있다. 처음에 기계어로 얼추 만든 A 실행기가 A로 만든 정교한 A 실행기를 실행해 줄 것이다.

6.3 번역: *불변성질*invariant 유지하기

부품에서 전체로, *불변성질*invariant 유지하기. 이것이 번역의 게임이다. 재미있고 쉬운 번역의 세계가 이것 때문이다.

위에서 이야기한 대로, 번역은 프로그램을 만드는 귀납 규칙을 따라 맴돈다. 번역 규칙은 프로그램을 만드는 귀납 규칙마다 하나씩 정의된다. 그래서, 프로그램이 조립된 과정을 따라서, 부품을 번역하고 번역된 부품을 조립해 간다. 기본 부품들의 번역 규칙이 있다. 귀납적인 조립 규칙마다, 번역 규칙이 있다. 번역된 부품을 가지고 어떻게 조립하면 번역된 전체가 나오는 지를 명시한.

이때 각 번역 규칙마다 항상 지켜지는 성질을 가지고 있으면, 번역이 쉬워진다. 모든 부품의 번역이 그 성질을 가지고 있고, 번역된 부품들이 조립될 때 그 성질을 이용해서 편리하게 조립된다.

마치 다음과 같다. 대한민국의 인구를 조사한다고 하자. 대한민국의 인구는 각 도의 인구를 조사하면된다. 각 도의 인구는 각 군의 인구를 조사하면 되고. 인구 조사 결과는 항상 10진법으로 표현되는 성질이 있다고하자. 부품의 결과를 가지고 전체의 결과를 만들기 쉬워진다. 각 군의 인구를 조사한다. 결과를 10진법의 숫자로 보고한다. 도의 인구는 각 군의 인구 숫자들의 10진법 합이면 된다. 10진법으로 모두 올라오므로 곧바로 더하면 된다. 그 결과를 다

시 10진법의 숫자로 표현해서 올린다. 최종적으로 대한민국의 인구는 각 도의 인구를 모으면 된다. 쉽다. 10진법으로 표현된 각 도의 인구의 수를 10진법으로 합하면 된다.

프로그램의 번역도 마찬가지이다. 프로그램은 부품들의 조립물이다. 프로그램의 번역은 번역된 부품들의 조립물이다. 모든 부품의 번역된 결과가 가지는 성질이 하나 정해져 있다고하자. 그 성질을 가지도록 각 부품을 번역해서 올린다. 그 성질 덕택에, 올라온 번역 부품을 모아서 보다 큰 부품의 번역을 구성하는 것이 쉬워진다. 그렇게해서 구성된 번역물에도 항상 지키기로 한 성질이 있도록 한다. 그 번역 결과를 다시 위로 올린다. 올라온 번역 부품들의 성질을 이용해서 다시 쉽게 조립된다.

예를 들어 보자. 정수식 언어를 기계어로 번역하는 것을 생각하자. 정수식 언어는 다음과 같다. 의미는 명백하다.

$$\begin{array}{rcll} E & \rightarrow & n & \text{integer} \\ & | & \texttt{-}E & \text{negation} \\ & | & E\,\texttt{+}\,E & \text{addition} \end{array}$$

기계어는 다음과 같다:

$$\begin{array}{rcll} C & \rightarrow & \epsilon & \text{(빈 명령어)} \\ & | & \texttt{push}\,n.C & (n \in \mathbb{Z}) \\ & | & \texttt{pop}.C & \\ & | & \texttt{add}.C & \\ & | & \texttt{rev}.C & \end{array}$$

기계어의 의미는 기계어를 실행하는 기계의 작동과정으로 정의한다. 기계는 “스택머신”이다. 그 기계는 스택 S와 명령어 C로 구성되어 있다:

$$\langle S, C \rangle$$

스택은 정수들이 차곡차곡 쌓인 것이다:

$$\begin{array}{rcll} S & \rightarrow & \epsilon & (\text{빈 스택}) \\ & | & n.S & (n \in \mathbb{Z}) \end{array}$$

기계 작동과정의 한 스텝은:

$$\begin{array}{rcll} \langle S, \texttt{push}\, n.C \rangle & \rightarrow & \langle n.S, C \rangle & \\ \langle n.S, \texttt{pop}.C \rangle & \rightarrow & \langle S, C \rangle & \\ \langle n_1.n_2.S, \texttt{add}.C \rangle & \rightarrow & \langle n.S, C \rangle & (n = n_1 + n_2) \\ \langle n.S, \texttt{rev}.C \rangle & \rightarrow & \langle -n.S, C \rangle & \end{array}$$

C에 있는 첫 명령어를 수행하면서 기계상태가 변하는 과정이다.

이제 정수식 E를 기계어 명령 C로 번역해주는 규칙을 만들어 보자. 정수식 부품의 번역이 지키는 성질은 이것이다.

> 정수식이 번역된 명령을 실행하면 그 정수식의 결과가 기계의 스택 맨 위에 놓인다.

위의 성질이 지키도록 번역규칙 $E \triangleright C$ 를 만들면 다음과 같다:

$$\frac{}{n \triangleright \texttt{push}\, n}$$

위의 성질이 유지된다. 식 "n"의 값은 n이다. "push n"은 n을 스택에 올린다.

$$\frac{E \triangleright C}{\texttt{-}\, E \triangleright C.\texttt{rev}}$$

위의 성질이 유지된다. 부품식 E의 번역 결과인 C를 실행하면 E의 결과가 스택 위에 오를 것이다. 곧이어 rev 명령을 실행하면 "- E"식의 결과가 스택 위에 오르게 된다.

$$\frac{E_1 \triangleright C_1 \quad E_2 \triangleright C_2}{E_1 \texttt{+} E_2 \triangleright C_1.C_2.\texttt{add}}$$

위의 성질이 유지된다. 부품식 E_1의 번역 결과인 C_1를 실행하면 E_1 의 값이 스택 위에 오를 것이다. 연이어 E_2의 번역 결과인 C_2를 실행하면 E_2의 결과가 스택 위에 쌓인다. 이제 스택의 꼭대기 두 개는 E_2와 E_1의 결과들이다. 곧이서 add 명령을 실행하면 "E_1 + E_2"의 결과가 스택 위에 오를것이다.

Exercise 8 한가지 확인할 것이있다. E_1 + E_2를 번역한 것 $C_1.C_2$를 실행하면, 스택의 꼭대기에는 E_2의 값이 쌓인다. 그리고, 그 바로 아래가 E_1의 값이어야 한다. 스택에 쌓인 E_1과 E_2의 값 사이에 다른 값이 쌓여있지 말아야, 위의 번역이 옳다. 그렇다고 증명할 수 있는가?

6.4 *가상의 기계*virtual machine

*"가상의 기계"virtual machine*는 언어일 뿐이다. 왜 "기계"인가? 그 언어가 대게 "기계어" 수준으로 낮기 때문이다. 왜 "가상"인가? 그 기계어의 *실행기interpreter*가 하드웨어(전깃줄)로 손에 잡히지 않고, 소프트웨어로 구성되어 있기때문이다.

*가상 기계virtual machine*의 용도는 무엇인가? 프로그래밍 언어를 구현하는데, *번역compilation*의 징검다리 역할을 한다. X 라는 언어를 디자인했다고 하자. 주어진 디지탈 컴퓨터의 기계어 Z 로 번역해야 한다. X 를 Z 로 곧바로 번역하기는 그 차이가 너무 크다. 중간 단계의 언어 Y 를 마련해서 그 곳을 경유하자. X 에서 Y 로 번역하고, Y 에서 Z 로 번역하자. 작은 차이를 건너는 번역은 큰 차이를 한 숨에 건너려는 번역보다는 쉽다.

Example 33 6.3절의 스택 머신은 가상의 기계이다.□

Example 34 다음의 기계는 K--나 K- 프로그램을 번역하는 데 어렵지 않을 언어를 정의한다.

SM5라고 부르자. "SM"은 "Stack Machine"을 뜻하고, "5"는 그 기계의 부품이 5개이기 때문이다:

$$(S, M, E, C, K)$$

S는 스택, M은 메모리, E는 환경, C는 명령어, K는 남은 할 일("continuation"이라고 부름)을 뜻하고 다음 집합들의 원소이다:

$$
\begin{array}{rcrcl}
S & \in & \mathit{Stack} & = & \mathit{Svalue}\ \mathit{list} \\
M & \in & \mathit{Memory} & = & \mathit{Loc} \to \mathit{Value} \\
E & \in & \mathit{Environment} & = & (\mathit{Var} \times (\mathit{Loc} + \mathit{Proc}))\ \mathit{list} \\
C & \in & \mathit{Command} & = & \mathit{Cmd}\ \mathit{list} \\
K & \in & \mathit{Continuation} & = & (\mathit{Command} \times \mathit{Environment})\ \mathit{list}
\end{array}
$$

$$
\begin{array}{rcrcl}
v & \in & \mathit{Value} & = & \mathit{Integer} + \mathit{Bool} + \mathit{Unit} + \mathit{Record} + \mathit{Loc} \\
x & \in & \mathit{Var} & & \\
\langle b, o \rangle, l & \in & \mathit{Loc} & = & \mathit{Base} \times \mathit{Offset} \\
 & & \mathit{Offset} & = & \mathit{Integer} \\
z & \in & \mathit{Integer} & & \\
b & \in & \mathit{Bool} & & \\
r & \in & \mathit{Record} & = & (\mathit{Var} \times \mathit{Loc})\ \mathit{list} \\
w & \in & \mathit{Svalue} & = & \mathit{Value} + \mathit{Proc} + (\mathit{Var} \times \mathit{Loc}) \qquad \text{(* stackable values *)} \\
p & \in & \mathit{Proc} & = & \mathit{Var} \times \mathit{Command} \times \mathit{Environment} \\
 & & \mathit{Cmd} & = & \{\texttt{push}\ v,\ \texttt{push}\ x,\ \texttt{push}(x, C), \\
 & & & & \quad \texttt{pop},\ \texttt{store},\ \texttt{load},\ \texttt{jtr}(C, C), \\
 & & & & \quad \texttt{malloc},\ \texttt{box}\ z,\ \texttt{unbox}\ x,\ \texttt{bind}\ x,\ \texttt{unbind},\ \texttt{get},\ \texttt{put},\ \texttt{call}, \\
 & & & & \quad \texttt{add},\ \texttt{sub},\ \texttt{mul},\ \texttt{div},\ \texttt{eq},\ \texttt{less},\ \texttt{not}\}
\end{array}
$$

기계의 작동은 다음과 같이 기계의 상태가 변화하는 과정으로 정의할 수 있다:

$$(S, M, E, C, K) \Rightarrow (S', M', E', C', K')$$

언제 어떻게 위의 기계작동의 한 스텝($\Rightarrow$)이 일어나는 지는 다음과 같다:

$$\begin{array}{llrrrl}
 & (S, & M, & E, & \texttt{push}\ v :: C, & K) \\
\Rightarrow & (v :: S, & M, & E, & C, & K)
\end{array}$$

$$\begin{array}{llrrrl}
 & (S, & M, & E, & \texttt{push}\ x :: C, & K) \\
\Rightarrow & (w :: S, & M, & E, & C, & K) \text{ if } (x, w) \text{ is the first such entry in } E
\end{array}$$

$$\begin{array}{llrrrl}
 & (S, & M, & E, & \texttt{push}\ (x, C') :: C, & K) \\
\Rightarrow & ((x, C', E) :: S, & M, & E, & C, & K)
\end{array}$$

$$\begin{array}{llrrrl}
 & (w :: S, & M, & E, & \texttt{pop} :: C, & K) \\
\Rightarrow & (S, & M, & E, & C, & K)
\end{array}$$

$$\begin{array}{llrrrl}
 & (l :: v :: S, & M, & E, & \texttt{store} :: C, & K) \\
\Rightarrow & (S, & M[v/l], & E, & C, & K)
\end{array}$$

$$\begin{array}{llrrrl}
 & (l :: S, & M, & E, & \texttt{load} :: C, & K) \\
\Rightarrow & (M(l) :: S, & M, & E, & C, & K)
\end{array}$$

$$\begin{array}{llrrrll}
 & (\mathit{true} :: S, & M, & E, & \mathtt{jtr}(C_1, C_2) :: C, & K) & \\
\Rightarrow & (S, & M, & E, & C_1 :: C, & K) & \\
\\
 & (\mathit{false} :: S, & M, & E, & \mathtt{jtr}(C_1, C_2) :: C, & K) & \\
\Rightarrow & (S, & M, & E, & C_2 :: C, & K) & \\
\\
 & (S, & M, & E, & \mathtt{malloc} :: C, & K) & \\
\Rightarrow & (\langle b, 0\rangle :: S, & M, & E, & C, & K) & \text{new } b \\
\\
 & (w_1 :: \cdots :: w_z :: S, & M, & E, & \mathtt{box}\, z :: C, & K) & \\
\Rightarrow & ([w_1, \cdots, w_z] :: S, & M, & E, & C, & K) & \\
\\
 & ([w_1, \cdots, w_z] :: S, & M, & E, & \mathtt{unbox}\, x :: C, & K) & \\
\Rightarrow & (v_k :: S, & M, & E, & C, & K) & w_k = (x, v_k), 1 \le k \le z \\
\\
 & (w :: S, & M, & E, & \mathtt{bind}\, x :: C, & K) & \\
\Rightarrow & (S, & M, & (x, w) :: E, & C, & K) & \\
\\
 & (S, & M, & (x, w) :: E, & \mathtt{unbind} :: C, & K) & \\
\Rightarrow & ((x, w) :: S, & M, & E, & C, & K) & \\
\\
 & (l :: v :: (x, C', E') :: S, & M, & E, & \mathtt{call} :: C, & K) & \\
\Rightarrow & (S, & M[v/l], & (x, l) :: E', & C', & (C, E) :: K) & \\
\\
 & (S, & M, & E, & \mathit{empty}, & (C, E') :: K) & \\
\Rightarrow & (S, & M, & E', & C, & K) &
\end{array}$$

$$
\begin{array}{llllll}
& (S, & M, & E, & \texttt{get} :: C, & K) \\
\Rightarrow & (z :: S, & M, & E, & C, & K) \quad \text{read } z \text{ from outside}
\end{array}
$$

$$
\begin{array}{llllll}
& (z :: S, & M, & E, & \texttt{put} :: C, & K) \\
\Rightarrow & (S, & M, & E, & C, & K) \quad \text{print } z \text{ and newline}
\end{array}
$$

$$
\begin{array}{llllll}
& (v_2 :: v_1 :: S, & M, & E, & \texttt{add} :: C, & K) \\
\Rightarrow & (\mathit{plus}(v_1, v_2) :: S, & M, & E, & C, & K)
\end{array}
$$

$$
\begin{array}{llllll}
& (v_2 :: v_1 :: S, & M, & E, & \texttt{sub} :: C, & K) \\
\Rightarrow & (\mathit{minus}(v_1, v_2) :: S, & M, & E, & C, & K)
\end{array}
$$

$$
\begin{array}{llllll}
& (z_2 :: z_1 :: S, & M, & E, & \texttt{mul} :: C, & K) \\
\Rightarrow & ((z_1 * z_2) :: S, & M, & E, & C, & K) \quad \text{similar for } \texttt{div}
\end{array}
$$

$$
\begin{array}{llllll}
& (v_2 :: v_1 :: S, & M, & E, & \texttt{eq} :: C, & K) \\
\Rightarrow & (\mathit{equal}(v_1, v_2) :: S, & M, & E, & C, & K)
\end{array}
$$

$$
\begin{array}{llllll}
& (v_2 :: v_1 :: S, & M, & E, & \texttt{less} :: C, & K) \\
\Rightarrow & (\mathit{less}(v_1, v_2) :: S, & M, & E, & C, & K)
\end{array}
$$

$$
\begin{array}{llllll}
& (b :: S, & M, & E, & \texttt{not} :: C, & K) \\
\Rightarrow & (\neg b :: S, & M, & E, & C, & K)
\end{array}
$$

SM5의 프로그램 C를 실행한다는 것은, C만 가지고 있는 빈 기계상태를 위에서 정의한 방식으로 변환해 간다는 뜻이다:

$$(\mathit{empty}, \mathit{empty}, \mathit{empty}, C, \mathit{empty}) \Rightarrow \cdots$$

K- 프로그램을 SM5 기계어로 번역하는 규칙을 정의할 수 있다. □