nML과 함께하는 프로그래밍 여행

이 재형, 한 기범

KAIST

Preface

재형이는 2000년 봄, 기범이는 2002년 봄, KAIST CS320을 수강하였고, 이 때 nML 프로그래밍의 세계를 처음으로 접하게 되었다. nML에 대한 좋은 책을 쓰고 싶은 욕심이 있다고 하길래, 내가 다음의 책을 nML과 우리말로 번안해 보면 후배나 많은 사람들을 위해 좋은 일이 될거라고 추천하게 되었다:

> *Functional Approaches to Programming*, Guy Cousineau and Michel Mauny, Cambridge Univ. Press, 1998

위의 책은, 프랑스의 엘리트 전산 코스에서 사용되는 책으로서 ML의 프랑스 버전인 OCaml을 예제로 하고 있다. 이 번역은 위의 책의 제4장 까지의 번안에 해당된다. 번역이 가지는 어색한 면이 있을 것이지만, 작고 의미있고 아름다운 시작이기를!

현재 재형이와 기범이는 병력특례요원으로 캠퍼스를 떠나있다. 마음으로 비는 행운이 전달되기를 빌며.

이 광근 (2003년 3월)

I 편

기본 원리

함수 중심적 프로그래밍(Functional Programming)이란 함수(function)의 정의와 응용을 중요한 개념으로 사용하는 프로그래밍 스타일이다. 이 접근 방법에서는, 변수들은 수학에서와 똑같은 역할을 한다. 즉, 그것들은 값(value)들을 표현하는 기호(symbol)이고, 특히, 함수의 인자(parameter)로 작용한다. 문법 구조(syntax)를 표현하는 기본적인 방법은 표현식(expression)을 사용하는 것이고, 함수 역시 표현식(expression)을 이용하고, 변수들을 인자로 하여 정의된다.

예를 들면, nML 언어에서는

$$\text{fn } x \rightarrow 2 * x + 1$$

이라는 표현식은 x를 $2 * x + 1$에 대응시키는 함수를 나타낸다.

만일 C 또는 파스칼에서 같은 함수를 정의하고자 했다면, 다음과 같이 썼을 것이다.

```
function F(x:integer) : integer;
begin
  F := 2*x+1
end
```

또는

```
int F(int x){
  return (2*x+1);
}
```

파스칼이나 C에서는, 우리의 정의는 약간의 필요없는 부분 - 임의의 이름인 F의 도입 - 을 포함한다. 그와 같은 종류의 언어들에서는, 우리는 함수에 이름을 매기지 않고 함수를 정의할 수 없다. 더욱 더 낭비인 것은, 전산학에만 관련되고 함수의 정의와는 기본적으로 관련이 없는 할당(assignment)연산 또는 return문이 쓰였다는 점이다. 또한, 비록 매우 중요한 차이는 아니지만, 파스칼이나 C에서는 함수의 선언 부분에 타입(type)이 존재한다. nML에서도 타입은 존재하지만, 그것들은 시스템에 의해 계산되고, 따라서 사용자가 선언할 필요는 없다.

이 예제로부터 기억할 필요가 있는 것은 함수 중심 스타일(Functional style)은 수학적 표현의 간결성과 그 정신을 보존하는 경향이 있다는 점이다. 예를 들면, 함수가 표현식들만으로 명시될 수 있다는 사실은 함수 자체가 다른 값들과 마찬가지로 고려된다는 것을 의미한다. 결과적으로, 함수들은 다른 함수의 인자로 들어가거나, 다른 함수의 결과로 나올 수 있다. 이것은 nML에서는 분명 가능하지만, 전통적인 언어에서는 그렇지 않다.

여러분은 이 책의 내용을 두 가지 다른 관점에서 읽을 수 있다. 우선 여러분은 이 내용을 예제를 통해 nML 언어 사용법을 보여주는 것으로 이해할 수 있다. 하지만, 여러분은 또한 그 내용을 아주 간단한 가정에서 출발한 어떤 프로그래밍 언어의 논리적인 구성 과정이라고 생각할 수도 있다. 그 언어가 반드시

표현식과 함수의 개념에 기반해야 한다는 기본적인 생각에서 출발하여, 우리는 조금씩 우리가 효과적으로 프로그래밍을 할 수 있는 틀을 만드는 데 꼭 필요한 개념들을 소개해 나갈 것이다.

이 구성 과정을 따라가면서, 우리는 수학과 매우 비슷한 생각들과 표기법을 사용할 것이다. 사실상, 프로그래밍 언어가 수학적인 전통을 피할 실용적인 이유는 전혀 없다. 오히려, 우리는 그 안에 들어가는 개념을 보다 정확하고 명확하게 표기하기 위한 수학자들의 몇 세기에 걸친 노력으로부터 이득을 얻어야만 하는 것이다. 우리가 수학적 표현으로부터 잠시 떠나야 할 때는, 우리는 왜 그래야만 하는지를 설명하고 이로 인해 원 수학적 표현보다 더 명확하게 무언가를 표현할 수 있게 되었는지, 아니면 다른 이유로 인해 그럴 수밖에 없었는지를 보여 줄 것이다. 다시 말하면, 그것이 심오한 이유에서인지 아니면 주변 환경에 의한 것, 예를 들면, 현재 컴파일러 기술의 한계에 의한 것인지를 보여 줄 것이다.

다음의 몇 페이지에 수학에서 나타나는 표현식과 함수 중심 프로그래밍에서 나타나는 표현식의 차이를 분석해 놓았다.

우리의 표현식의 매우 명확한(그러나 그리 중요하지 않은) 수학적 전통과의 다른 점들 중의 하나로, 문법 구조(syntax)가 쓰여진 모습의 차이를 들 수 있다. 일반적으로, 프로그래머들은 전통적인 ASCII 알파벳에 속하는 문자들로 프로그램을 짠다. 따라서, 수학적 쓰기 표현식인 $\sqrt{x^2 + y^2}$은 sqrt(x*x + y*y) 또는 sqrt(x**2 + y**2)로 바뀌어 쓰여야 한다.

문법 구조를 어떤 문자열 형태로 나타낼지를 선택하는 것(전산학 용어로는 구체적 문법 구조(concrete syntax)라 한다.)은 쉬운 문제가 아니다. 사실, 그것은 중요한 분야인 문법 구조 분석(syntax analysis)에서 다루어지고 있다. 따라서, 지금 우리는 이를 깊게 다루지 않을 것이다. 문법 구조 분석을 위한 기술들의 개발은 우리가 주로 표현식의 깊은 구조, 전산학 용어로는 추상적 문법 구조(abstract syntax)에 집중할 수 있도록 해 주었다.

그렇다 하더라도, 문법 구조에 대한 논의에는 어려운 문제들이 끼어 있다. 예를 들면, 쓰여진 기호의 해석에 관한 문제들과 같은 것이다. 일반적으로, 수학적 표현식들은 문맥, 독자의 지능, 또는 복잡한 인쇄상의 관습을 쓰지 않고는 해석해 낼 수 없는 모호성을 가진 기호들을 포함한다. 에를 들면, 어떤 선형대수학 교과서에서도, $M_1 \times M_2$ 는 두 행렬의 곱을 의미하지만, 다른 문맥에서는, $\times$ 연산은 많은 다른 의미를 가질 수 있다. 프로그래밍 언어에서는, 기호의 정의와 그 사용을 잇는 방법은 정확하고 모호하지 않게 정의되어야 한다. 그렇게 하는 것은 유효범위(scope)와 환경(environment)에 대한 문제를 만들어 내고, 어떤 완전히 정형화된 언어의 사용에서도 이 문제들은 존재한다.

또한, 수학적 표현식은 일반적으로 한 단계 이상의 뜻을 지니고 있다. 예를 들면

$$x^2 + 2x + 1$$

은 변수 x에 대한 다항식을 표현함과 동시에 그것과 연결된 R에서 R 로의 함수를 표현한다.

전산학에서는, 우리가 작업하는 곳에서의 뜻의 단계는 반드시 완전히 명백해야 한다. 프로그래머에 의해 쓰여진 어떤 표현식도 유일한 방법으로 정의되는 값으로 값매김(evaluation) 가능해야 한다. 값과 표현식 사이의 대응 관계는

프로그래밍 언어에서는 의미 구조(semantics)라고 불리워지고, 반드시 모호하지 않게 정의되어야 한다.

따라서 우리는 표현식(프로그래밍 언어에서의)과 값(계산의 결과로 나오는)을 분명히 구분할 것이다. 이렇게 할 경우, 고려해야 할 문제들이 늘어난다. 예를 들면, 우리가 일반적인 계산을 하기 위해 프로그램을 짤 때 다루어지는 형식 표현들은 값들 그 자체이고, 따라서 프로그래밍 언어에서의 표현식과는 구분되어야 한다. 프로그래밍 언어에서의 표현식은 이 표현들과 비교할 때 일종의 중간 언어로서의 역할을 한다. 이렇게 보통의 언어와 중간 언어를 구분하는 일은 현재의 수학적 관례에서는 낯선 것이다. 반대로, 논리학에서는 전산학에서와 똑같은 이유로 그것은 중요한 구분이다.

마지막으로, 우리는 타입에 대한 개념에 대해 다시 이야기할 필요가 있다.그것은 수학에서는 분명히 표현되지 않고 쓰였지만, 전산학에서는 보다 정확하고 더욱 제한적인 의미로 쓰인다. 표현식이 값매김 가능하기 위해서는, 옳은 문법 구조를 가지는 것만으로는 충분하지 않다. 좀더 복잡한 문제가 있다. 예를 들어, $f(x)$와 같은 표현식(f는 함수이고 x는 함수 내의 독립 변수)에서, 이 표현식이 값매김되고 뜻을 가지기 위해서는 x는 함수 f의 정의역 내에 속해 있어야 한다. 만일 그렇지 않다면, 기계적인 값매김 규칙의 적용은 보통 수행이 불가능한 일을 요구하고 말 것이다. 예를 들면, 문자열의 제곱근을 구하려고 하는 일을 요구한다거나 그와 비슷한 일을 하려 할지도 모른다. 이런 종류의 행동은 실행 에러를 만들어 내거나 완전히 의미가 다른 답을 만들어 낼 수 있다. 이런 이유들 때문에, 우리는 언어의 값들을 그것들의 타입에 따라 나누게 되었다. 값들이 타입을 가지게 되면, 정확한 표현식 역시 타입을 가지게 된다. 즉, 그것들의 타입은 그것들의 값의 타입과 같게 된다. 만일 이 타입을 실제 값매김 전에 구할 수 있다면, 표현식에서의 틀릴 가능성이 있는 곳을 찾을 수 있을 것이고, 우리는 값매김이 적당하지 않은 인자에 연산자를 적용하거나 수행 에러를 내지 않는다고 보증할 수 있을 것이다.

우리가 사용할 프로그램 접근 방식은 값매김 가능한 표현식에 대한 개념에 기반하고 있다. 그것의 "가장 순수한 형태인" 1장과 2장에서는, 이 접근 방식은 등치만들기 논법(equational reasoning)을 쉽게 이해하게 해 줄 것이다.(3장에서 자세히 다룬다.) 3장에서는 "값매김 가능한 표현식" 이 무엇을 의미하는지 처음으로 좀더 정확하게 정의할 것이다. 표현식과 그 값과의 대응 관계에 대한 정형화된 정의는 우리가 이 언어의 의미 구조(semantics)라고 부를 어떤 것을 구성하게 된다. 의미 구조는 프로그램에 대한 논리적 생각의 기반을 제공하며 구현할 사항에 대한 명세서의 역할 또한 해 준다. 3.1절에서는 nML에서의 의미 구조를 정의하고, 이는 3.4절에서 프로그램이 맞는지를 확인하는 데 쓰인다. 의미 구조는 또한 **??**절에서 정의된 구현에서의 참고 자료의 역할을 한다.

표현식의 타입을 정확하게 정의하기 위해, 우리는 언어의 모든 구성 방법(construction)에 대해 타입만들기 규칙(typing rule)을 정의해야 한다. 이와 같은 타입만들기 규칙은 때때로 동적인 의미 구조(dynamic semantics)에 대조하여, 정적인 의미 구조(static semantics)라고 알려져 있고, 이는 값매김 규칙을 정의한다. 정적인 의미 구조와 동적인 의미 구조 중 어느 것이 적절한지의 여부는 값매김이 표현식의 타입을 보존하는지를 증명하는 것으로 확인할 수 있다. 우리는 타입이 정적으로 계산(즉, 실행 전에 계산)된 표현식의 경우 타입 에러

를 일으키지 않고 값매김이 가능함을 보증할 수 있다. 우리의 언어에서 타입을 결정하는 규칙은 3.5절에 정의되어 있다.

우리가 말했던 내용에서, 할당(assignment)에 대한 내용이 전혀 언급되지 않았음을 눈여겨 보라. 할당 연산은 변수의 값을 여러 차례 바꾸는 것을 가능하게 해 준다. 그것은 대부분의 전통적인 프로그램 언어에서 필수적이었고, 프로그램 언어의 관점에서는, 그것은 저수준 기계 연산, 즉 메모리의 내용을 실제로 변경하는 것을 반영하고 있었다. 프로그램에 대해 논리적으로 접근할 때, 이 연산은 심각한 단점을 가지고 있다. 그것은 우리가 프로그램이 같은지 같지 않은지를 따지는 것을 힘들게 한다. 예를 들면, 만일 f가 함수이고 x가 변수라면, 만일 f가 내부에서 할당 연산을 수행할 경우 $f(x) + f(x)$는 $2f(x)$와 같지 않을 수도 있다. 따라서 할당 연산을 사용하는 것은 프로그램의 정확성을 어떤 형식을 사용하여 검증하는 것을 대단히 어렵게 한다.

운 나쁘게도, 어떤 알고리즘들을 효과적으로 사용하려면 할당 연산을 완전히 사용하지 않는 것은 힘들게 된다. 따라서, 우리는 4장에서 할당 연산에 대해 다룰 것이다.

우리의 프로그래밍 언어에 대한 정의는 4장에 걸쳐 이루어진다. "표현식에 대하여" 라는 제목을 가진 첫 번째 장에서는 언어 내의 구성 방법(constructions)에 대해 다루고 그것들을 어떻게 사용하는지에 대해 설명한다. 이 장에서, 우리는 주로 매우 간단한 숫자 타입을 이용한 프로그램들만을 다룬다. "데이터 구조" 라는 제목을 가진 두 번째 장에서는 복잡한 데이터를 다루기 위해 어떻게 사용자들이 그들만의 타입을 정의할 수 있는지에 대해 설명한다. "의미 구조에 대하여" 라는 제목을 가진 세 번째 장에서는 언어의 정적인 의미 구조와 동적인 의미 구조에 대하여 정확히 정의하고 프로그램이 정확한지를 증명하기 위해 어떻게 해야 하는지를 보여 준다. 이 처음의 3장은 순수하게 함수 중심인 내용만을 다룬다. "기계 중심적인 측면들" 이라는 제목을 가진 4장에서는 nML의 함수 중심적이지 않은 기능들에 대해 다루고 어떤 종류의 프로그램에서 그 기능들을 쓰는 것이 필요한지에 대해 설명한다. 또한 이 장에서는 그 기능들의 사용이 언어의 의미 구조에 어떤 영향을 미치는지에 대해 다룬다.

1 장

표현식에 대하여

이 책은 매우 많은 예제를 포함하고 있다. 이 예제들은 마치 nML의 대화식 모드에서 컴퓨터의 키보드에 타이프된 것처럼 표현될 것이다. 결국, 그것들은 사용자에 의해 입력된 줄과 시스템의 반응을 함께 포함할 것이다. 예제의 시작 부분에서 나타나는 #문자는 시스템 프롬프트이다. 사용자에 의해 쓰여진 글씨들은 #뒤에서 시작하여 이중 세미콜론(;;)으로 끝난다. 따라서 #과 ;; 사이의 모든 것은 사용자에 의해 입력된 것이다. 나머지는 시스템의 반응이다. 이 시스템 반응은 타입과 값에 대한 정보를 포함한다.

nML 시스템에서는, 사용자에 의해 입력된 표현식의 타입은 정적으로 계산된다. 즉, 값매김 전에 계산된다. 시스템은 모든 가능한 타입의 불일치를 자동으로 찾아서 에러 메세지로 보고한다. 이 타입 체킹은 사용자가 시스템에 타입에 대한 어떤 정보도 직접 써 줄 필요 없이 이루어진다. 즉, C나 파스칼에서처럼 타입을 선언해 줄 필요가 없다.

타입이 만족스럽게 만들어진 후에, 값매김이 시작되고, 결과가 계산되어 화면에 출력된다. 이 출력은 사용자가 입력한 내용이 간단한 표현식인지 아니면 정의인지에 따라 두 가지의 다른 형태 중 한 가지로 나온다.

여기에서 주어진 예제는 실제 동작할 때 스크린상에 나타나는 것과는 약간 글자의 모양이 다르다. 즉, 우리는 읽기 쉽고 보기 좋게 하기 위해 글자 모양에 약간의 수정을 가했다. 예를 들면, 언어에서 쓰이는 예약어는 두꺼운 글씨로 표시했고, 터미널에서 두 개의 아스키 문자(->)로 나타나는 화살표는 여기에서는 → 한 문자만으로 나타냈다. 그렇다 하더라도, 여러분은 이 예제들에서 쓰인 표현 방식과 여러분이 스크린에서 보게 될 아스키 형식간의 대응 관계를 명확히 파악할 수 있을 것이다.

1.1 표현식과 정의의 구분

방금 전 이야기했듯이, nML의 대화식 모드에서는 표현식과 정의는 다르게 표시된다. 이 절에서는 둘 사이의 차이에 대해 설명한다.

1.1.1 표현식

가장 전통적인 표현식들은 산술에 대한 것들이다. 우리의 첫번째 예제에서, 우리는 2+3이라는 표현식을 입력하고 시스템의 반응을 살펴볼 것이다.

```
# 2+3;;
- : int = 5
```

이 반응에서, "-"기호는 사용자에 의해 입력된 표현식임을 나타낸다. int 는 그 표현식의 타입을 나타내고(즉, 정수형의 값임을 나타낸다), 물론 5 는 그 값을 나타낸다.

말로 표현하면, 시스템의 반응은 다음과 같이 해석될 수 있다.

> 당신이 방금 입력한 표현식의 타입은 int이고 그 값은 5입니다.

완전히 전통적이고 익숙한 방법으로, nML에서의 산술 표현식은 모호함을 해결하기 위해 괄호를 사용할 수 있다. 괄호가 없을 때, *기호(곱하기)와 /기호(나누기) 는 +기호(더하기)와 -기호(빼기)에 대해 우선권을 가진다. 같은 우선 순위의 기호 사이의 모호성을 해결하기 위해, 기본적으로, nML은 왼쪽 결합에 우선 순위를 둔다. 말하자면, $a+b+c$는 $(a+b)+c$로 취급된다. 물론, 여러분이 괄호를 쓰면, 여러분이 원하는 대로 표현식을 덩어리로 묶어 조직화할 수 있다.

```
# 1+2*3;;
- : int = 7
# (1+2)*3;;
- : int = 9
# 1-2+3;;
- : int = 2
# 1-(2+3);;
- : int = -4
# 4/2*2;;
- : int = 4
# 4/(2*2);;
- : int = 1
```

나중에 보게 되겠지만, 괄호는 간단한 산술 표현식 뿐만이 아니라 모든 nML 표현식에 대해 사용할 수 있도록 확장되어 있다.

1.1.2 정의

정의는 여러분들이 여러분의 환경에 새로운 변수를 만들 수 있게 해 주고 이 새로운 변수들에 값을 연결시켜 준다. 수학에서처럼, 변수라는 개념은 여러분이 값들을 기호로 나타낼 수 있게 해 준다. 또한 변수의 사용은 계산된 값에 이름을 붙여서 여러번 재활용 가능하게 해 준다. 이를 이용해 큰 계산을 여러 부분으로 나누어서 할 수 있다. nML에서는 변수는 let 이라는 구성 방법을 이용하여 만들 수 있다. 이미 알고 있듯이, 프로그래밍 언어에서 변수의 이름은 식별자(identifier)라고 불려진다. nML에서는, 차차 이야기하겠지만, 값, 타입 구성

자, 자료(data) 구성자, 레코드(record)의 필드, 타입형식(pattern) 변수에 식별자를 붙일 수 있다. 여기서는 우선 값에 식별자를 붙여 보기로 한다.

nML에서는, 값의 식별자는 앞에 val을 붙여서 나타낸다. 값의 식별자는 반드시 영어 소문자 혹은 한글로 시작해야 하며, 문자, 숫자, _(밑줄), 또는 '(다시)로 이루어진다.

```
# val x = 2+3;;
val x :  int = 5
# val pi = 3.14;;
val pi :  real = 3.14
```

nML에서의 위와 같은 정의는 수학에서의 다음과 같은 말과 같은 뜻이다.

x가 2+3이라고 가정하자.

또는

pi가 3.14라고 가정하자.

변수 x와 pi를 정의할 때 let이 쓰이지 않은 것에 주의하라. 대화식 모드에서 위와 같이 정의되면, 이 변수들은 작업의 나머지 부분 전체에서 사용 가능하다. 예를 들면 위 정의 다음에 다음과 같은 표현식을 쓰는 것이 가능하다.

```
# x*x;;
- :  int = 25
```

1.1.3 지역에 한정된 정의

여러분은 또한 nML에서 무언가를 지역적으로 정의할 수 있다. 즉, 여러분은 어떤 표현식의 유효 범위를 지정할 수 있다.

```
# let val x = 2 in x+1 end;;
- :  int = 3
# let val a = 3 and val b = 4 in a*a+b*b end;;
```

이 정의들은 전체 범위에서 유효하지는 않다. 예를 들면, x의 정의는 x+1이라는 표현식 내에서만 유효하다. 값매김 후에, 변수 x는 다시 이전에 정의되었던 값이 된다.

```
# x;;
- :  int = 5
```

let val x=2 in x+1 end 과 같은 입력은 nML에 의해서 전체가 표현식 하나로 취급되고, 정의로 취급되지 않는다.

1.2 기본적인 타입들

nML에 의해 제공되는 주요 기본 타입들은 정수(int), 실수(real), 논리값(bool), 문자열(string), 그리고 문자(char)이다.

1.2.1 정수 타입

nML에서, 정수는 −1073741824 에서 1073741823 까지이다. 즉, 30비트를 이용하여 표현된다. 여러분은 숫자 앞에 0x, 0o, 0b등을 명시하여 각각 16진수, 8진수, 2진수를 표현할 수 있다. 여러분이 int형 타입에 쓸 수 있는 연산자들 중 기본적인 것들은 다음과 같다.

`+`	더하기
`-`	빼기
`*`	곱하기
`/ or div`	정수 나누기
`mod`	나머지(modulo) 연산

nML에서의 정수 연산은 계산에서 넘침(overflow)이 일어나지 않는 이상 정확한 산술 연산을 수행한다. 그리고, 임의의 크기의 정수값을 계산할 수 있게 해주는 nML 라이브러리 역시 존재한다. 10장에서, 우리는 그 라이브러리를 구현하는 데 기본이 되는 원리들에 대해 살펴 볼 것이다.

1.2.2 실수 타입

실수는 근사값 연산에 사용된다. 이들은 내부적으로 숫자 $m \times 10^n$ 에 대해 밑(m)과 지수(n) 으로 나타내어진다. 이와 같은 숫자는 시스템에 의해 밑을 나타내는 소수 부분, 알파벳 e, 다음에 앞의 숫자에 10의 몇 제곱을 곱해야 하는지를 나타내는 상대적인 정수값이 차례로 출력되어 나타내어진다. 소수점 아랫 부분이 없는, 즉 int 형으로 표시될 수 있는 실수의 경우, nML은 이를 소숫점 없이 출력해 준다. 이 경우에는, 그 숫자의 타입만이 여러분에게 그 숫자가 실수 타입이라는 것을 알려 줄 것이다.

```
# 1.0;;
- :  real = 1
```

nML 타입 시스템에 의하면, int타입과 real타입은 서로 섞어 쓸 수 없다. 그들 사이에는 자동 타입 변환이 가능하지 않다. 이 선택이 옳은지는 논쟁의 여지가 있지만, 이는 정확한 계산을 근사값 계산과 명확히 나누어 준다는 장점이 있다. 그럼에도 불구하고, 정수를 실수로 바꾸어 주는 float_of_int 함수가 존재한다.

다음은 실수에 대해 적용될 수 있는 연산자들이다.

`+`	더하기
`-`	빼기
`*`	곱하기
`/`	나누기

익숙한 삼각 함수들이 실수형 숫자들에 적용된다.

sqrt	제곱근
log	로그
exp	e^x
sin	사인
cos	코사인
tan	탄젠트
asin	아크사인
acos	아크코사인
atan	아크탄젠트

여기 약간의 예제가 있다.

```
# sqrt(2);;
This expression has type int but is here used with type real
```

sqrt 함수는 정수형이 아닌 실수형을 인자로 받기 때문에 타입 에러가 나게 된다.

```
# sqrt(2.0);;
- :  real = 1.41421356237
# sqrt(float_of_int(2));;
- :  real = 1.41421356237
# acos(-1.0);;
- :  real = 3.14159265359
# let val x=sin(1.0) and y=cos(1.0) in (x *** x)+++(y *** y) end;;
- :  real = 1
```

1.2.3 논리값 타입

논리값은 두 가지의 상수 true와 false로 구성된다. 비교를 하는 함수들 ($=, <, >, \leq, \geq$)은 논리값으로 된 결과를 낸다.

```
# 1<2;;
- :  bool = true
```

논리값은 **not**(부정), **andalso**(논리곱), **orelse**(논리합) 과 같은 논리 연산에 의해 묶여질 수 있다.

```
# 1<=2 andalso (0<1 orelse 1<0) andalso not(2<2);;
- :  bool = true
```

논리값은 if – then – else 라는 구성 방법에서 특히 유용하다.

```
# let val x=3 in
  if x<0 then x else x*x
  end;;
- :  int = 9
```

이 구성 방법은 기존의 언어에서의 if문과 다르다. 그것은 표현식(expression)에 영향을 미치고, 그것 자체가 표현식으로 취급된다.

1.2.4 문자열 타입

nML에서, 문자열은 **string** 타입으로 취급된다. 문자열은 "(큰 따옴표) 기호 사이에 써야 한다. ^(악센트) 기호로 문자열을 붙일 수 있다.

```
# "a string" ^" of characters";;
- :  string = "a string of characters"
```

1.2.5 문자 타입

문자 타입(**char**)은 주로 입출력에 사용된다. 문자는 '(작은 따옴표) 사이에 쓴다.

```
# 'a';;
- :  char = 'a'
```

문자에 대한 아스키 코드값은 int_of_char 함수로 구할 수 있다.

```
# int_of_char 'a';;
- :  int = 97
```

1.3 데카르트 곱

2장에서는 어떻게 데이타가 사용자에 의해 정의된 타입에 대응하여 복잡한 구조 속으로 조직되어 들어가는지에 대해 설명한다. 하지만 이 시점에서는, 우리는 단순히 데카르트 곱이라는 수학적 개념을 이용하여 복잡한 값을 만들어 내는 방법에 대해 알아 볼 것이다.

값들은 2개씩 또는 3개 이상씩 같이 묶일 수 있다. 이를 위해서, 전통적인 방법과 똑같이 필드(field) 사이의 경계를 나타내는 데 ,(쉼표)가 사용된다. (더 자세한 내용이 1.6절에 있다.) "데카르트 곱"의 타입은 * 기호를 이용하여 나타내어진다.

```
# (1,1.2);;
- :  int * real = (1, 1.2)
# (1+2, true orelse false, "hello");;
- :  int * bool * string = (3, true, "hello")
# ((1,2), (3,4));;
- :  (int * int) * (int * int) = ((1, 2), (3, 4))
```

다음과 같은 타입들은 서로 다른 것으로 취급된다.

$$t_1 * t_2 * t_3, \qquad (t_1 * t_2) * t_3, \qquad t_1 * (t_2 * t_3)$$

첫번째 것은 3개가 한 묶음으로 취급되고, 두번째 것은 첫번째 필드가 2개 한 묶음으로 된 2개짜리 묶음으로 취급된다. 세번째 것은 두번째 필드가 2개 한 묶음으로 된 2개짜리 묶음으로 취급된다.

fst 와 snd 함수는 여러분이 2개짜리 묶음의 첫번째 필드와 두번째 필드를 얻을 수 있게 해 준다.

```
# snd(1+2, 3+4);;
- :  int = 7
# fst((1,2),3);;
- :  int * int = (1, 2)
# let val p=(1,2) in
  (snd(p),fst(p))
  end;;
- :  int * int = (2, 1)
```

나중에 임의의 갯수의 필드가 묶인 경우 어떻게 그중 원하는 필드를 얻는 함수를 정의할 수 있는지에 대해 다룰 것이다.

데카르트 곱은 단지 정해진 크기의 필드들의 묶음만을 다룰 수 있다는 사실에 주의하라. 크기가 변하는 것들을 다루려면, 여러분은 **list** 라는 타입을 써야만 한다.(리스트에 대해 더 알려면, 2.2.6절을 참고하라.)

1.4 함수에 대하여

nML에서, 함수들은 수학에서와 매우 비슷한 방법으로 정의된다.

1.4.1 간단한 함수 정의하기

실수 x를 그 제곱인 X^2과 대응시키는 함수 sq를 정의하려면, 다음과 같이 쓴다.

```
# fun sq(x) = x *** x;;
val sq :  real -> real = <fun>
```

함수 타입은 전통적인 방법과 마찬가지로 화살표를 이용하여 표시된다. 함수를 정의한 후에 시스템이 알려주는 정보는 타입에 대한 것이 전부이다. 값 부분에 시스템은 ¡fun¿만을 표시한다.

이론적으로는 함수의 내부를 표시하는 것이 가능하지만, 이 표시는 함수를 정의하는 확장된 내용이 될 수밖에 없고, 이는 매우 클 수도 있다. 게다가, 함수값의 내부적 표기 방식에서 이 내용을 다시 만들어 내는 것은 상당히 까다롭다. 12장에서, 여러분은 실제 함수의 내부 표기 방식을 볼 수 있을 것이다.

사용자에 의해 정의된 함수는 미리 정의된 함수와 똑같은 방법으로 쓰일 수 있다.

```
# sq(2.0);;
- :  real = 4
```

2개 이상의 인자를 가지는 함수들 역시 같은 방법으로 정의되고, 물론 인자 1개가 들어갈 자리에 인자의 묶음을 쓰면 된다.

```
# fun module(x,y) = sqrt(sq(x) +++ sq(y));;
val module :  real * real -> real = <fun>
# module(3.0,4.0);;
- :  real = 5
```

nML에서는, 값매김은 우리가 이전에 했던 함수 정의들이 유효하다는 사실에 기초한다. f가 fun `f(x) = e` 에 의해 정의된 함수일 때 `f(v)` 의 값을 계산하기 위해서는, x를 v로 대체한 표현식 e 를 값매김하면 된다.

nML 프로그램의 기반이 되는 것은 함수이다. 프로그램은 여러 개의 함수 정의와 그에 뒤따라 나오는 값매김할 표현식으로 이루어진다. 그 표현식의 값, 즉 값매김 결과가 이 프로그램의 결과가 된다.

연습 문제

1.1 실수 숫자 a,b,c를 인자로 받아 $ax^2 + bx + c = 0$ 이 해를 가지는지 판별하는 함수를 써 보아라.

1.4.2 함수 표현식에 대하여

nML에서는, 함수에 이름을 주지 않고 함수를 나타내는 것이 가능하다. 예를 들면, 우리는 바로 전 절에서, x와 y를 $\sqrt{x^2 + y^2}$ 에 대응시켜 주는 함수에 `module` 이라는 이름을 주었다. 그것은 이런 방법으로 표시될 수도 있다.

```
# fn (x,y) => sqrt(sq(x) +++ sq(y));;
- :  real * real -> real = <fun>
```

변수 $x_1, \cdots, x_n$ 을 포함하는 표현식 e로부터 함수 (fn $x_1, \cdots, x_n \rightarrow$ e)를 만들어 내는 과정을 추상화(abstraction)이라 한다. 표현식 (fn $x_1, \cdots, x_n \rightarrow$ e) 은 함수 표현식(function expression)이라 불린다. 이와 같은 표현식은 함수의 이름이 들어갈 자리에 쓰일 수 있고, 특별하게는, 함수의 인자로도 들어갈 수 있다.

```
# (fn (x,y) => sqrt(sq(x) +++ sq(y))) (3.0,4.0);;
- :  real = 5
```

이렇게 함수를 표시하면 함수를 값의 상태로 표시할 수 있다. nML에서의 이와 같은 표현식은 다른 표현식들이 숫자나 묶음을 표시하는 방법과 마찬가지 방법으로 함수를 표시한다. 따라서 이 언어에서 함수를 선언하기 위해 특별한 구성 방법을 마련하는 것은 기본적으로 불필요한 것이다. 사실상, 다음과 같이 쓰는 것이 가능하다.

```
# val sq = (fn x => x *** x);;
val sq :  real -> real = <fun>
```

이 정의는 우리가 예전에 보다 전통적인 모습으로 함수 `sq`를 정의했을 때와 동등하게 취급된다.

1.4.3 더 높은 차수의 함수 표현식

함수들이 그 자체로 값으로 고려되기 때문에, 그것들이 다른 함수의 인자나 결과로 나타나는 것 역시 자연스런 것이다. 다른 함수를 인자로 받거나 함수를 인자로 돌려 주는 함수들은 더 높은 차수를 가진다고 불리고, 우리는 그들을 평범한 함수들과 구분하기 위해 "일반화된 함수(functinoal)" 라고 부른다. 예를 들어 모든 함수 $f(x)$ 를 함수 $\frac{f(x)}{x}$ 와 대응시키는 일반화된 함수 h에 대해 생각해 보자. 우리는 그것을 다음과 같이 정의할 수 있다.

```
# val h = (fn f => (fn x => f(x)///x));;
val h :  (real -> real) -> real -> real = <fun>
```

여러분은 h가 쓰인 방법에서 괄호가 부족하다고 생각할지도 모른다. 사실, 우리가 나중에 문법 구조에 대해 다룰 1.6절에서 언급할 nML 결과표기 방법의 관습 때문에 저런 결과가 나온 것이다. 지금은, 여러분은 그것을 `(real → real) → (real → real)` 로 받아들여야 한다.

0근처에서, h(sin)은 1에 근접한다.

```
# (h(sin))(0.1);;
- :  real = 0.998334166468
```

h(sin)은 그것만으로 의미를 지닌 표현식이다. 그것은 `float → float` 타입의 함수를 표시한다. 그것은 우리가 이미 해 보았듯이 인자로 넘겨질 수도 있고, 또한 그것은 다양한 방법으로 쓰일 수 있다. 예를 들면, 그것은 식별자(identifier)와 연결될 수도 있다.

```
# val k = h(sin);;
val k :  real -> real = <fun>
# k(0.1);;
- :  real = 0.998334166468
# k(0.00001);;
- :  real = 0.999999999983
```

k를 val `k = h(sin)`로 정의하는 대신에, 우리는 val `k = fn x => (h(sin))(x)` 로 정의할 수도 있다. 사실 우리는 x에 대해 독립인 모든 표현식 e에 대해, 표현식 (fn `x => e(x)`)은 표현식 e와 같은 함수를 나타내게 된다. 여러분이 보았듯이, `fun k(x) = (h(sin))(x)` 는 val `k = fn x => (h(sin))(x)`, 즉 val `k = h(sin)` 과 동치이다.

만일 우리가 fn 구성 방법을 연이어 쓰면, 우리는 조금 읽기 어려운 것을 얻게 된다. 따라서 관습으로, 우리는 다음과 같이 쓰는 대신,

$$\text{fn } x_1 \to (\ldots \to (\text{fn } x_n \to e)\ldots)$$

다음과 같이 좀더 단순하게 쓴다.

$$\text{fn } x_1 \ldots x_n \to e$$

예를 들면, h에 대해서는, 우리는 다음과 같이 쓸 수 있다.

```
# val h = fn f x => f(x)///x;;
val h :  (real -> real) -> real -> real = <fun>
```

nML을 보다 읽기 쉽게 만들기 위한 다른 관습들은 문법 구조에 대해 다룬 1.6절에서 다룬다.

연습 문제

1.2 주어진 `f:real → real` 과 작은 구간 `dx` 에 대해, f의 미분값은 `(f(x+dx)-f(x))/dx`

로 근사될 수 있다. 어떤 함수 f와 주어진 작은 구간 dx에 대해서 항상 이를 f' = (f(x+dx)-f(x))/dx 로 정의된 f' 와 연관시키는 미분 함수를 써 보아라.
1.3 함수 f 와 작은 구간 dx를 인자로 받아서 결과로 x에 대해 f(x), f(x-dx), f(x+dx)의 평균을 돌려주는 함수를 돌려주는 '부드럽게 하는' 함수를 써 보아라.

1.4.4 자기 참조 함수

fun x = e 형태의 정의에서, 표현식 e 내에 나타나는 변수들은 반드시 미리 정의되어 있어야 한다. x 의 경우 e 내에 나타나는 것이 가능하지만, 만일 나타날 경우 우리가 지금 정의하고 있는 x가 아닌 이전에 정의된 x여야 한다. 다음 예제가 이를 보여준다.

```
# val x = 1;;
val x :  int = 1
# val x = x+2;;
val x :  int = 3
```

반면, 우리가 함수를 정의할 때는, 종종 함수 자체의 정의에서 그 함수 자신을 호출하는 것이 유용하다. 그런 경우를 우리는 자기 참조 함수(recursive function)라고 부른다. 예를 들면, 우리가 자기 참조에 의해 정의되는, 즉, 스스로 반복되는 함수를 짜게 되면, f(n) 의 정의는 사실상 f(n-1) 을 호출한다. 이런 정의의 경우, 우리는 val 이라는 구성 방법을 val rec 이라는 명시적으로 자기 참조를 하는 구성 방법으로 바꾼다. 예를 들면, 계승(factorial)을 구하는 함수의 자기 참조 정의는

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) \times n! \end{aligned}$$

다음과 같이 표시된다.

```
# val rec fact =
  fn n => if n=0 then 1 else n*fact(n-1);;
val fact :  int -> int = <fun>
# fact(5);;
- :  int = 120
```

한편, fun을 이용하여 함수를 구성할 경우, 기본적으로 자기 참조가 가능하게 된다. 즉, fun fact(n) = if n=0 then 1 else n*fact(n-1);; 과 같이 쓸 수 있다.

우리의 예전 목표 - 그 값과 연관된 변수들을 정의함으로써 값의 정의를 완전히 그 값의 이름과는 독립시키는 것 - 를 생각하면 자기 참조 함수를 표시하기 위해 fn rec 과 같은 표기법을 만들어 내야 한다는 생각이 들지도 모른다. 그러나, 그와 같은 표기법은 nML에는 존재하지 않는다. 하지만 우리는 값매김에 대해 다루는 3.1절에서 이에 대해 다시 살펴 볼 것이다.

연습 문제

1.4 실수 a와 정수 n에 대해 a^n을 계산하는 함수를 써라. 먼저, $a^0 = 1, a^{n+1} =$

aa^n 이라는 사실을 이용하라. 다음에 $a^{2n} = (a^n)^2, a^{2n+1} = a(a^n)^2$ 라는 특징을 이용하는 두 번째 함수를 써라. 이 두 함수의 시간 복잡도를 비교하라.
1.5 두 양의 정수의 최대 공약수를 계산하는 함수를 써라.
1.6 어떤 정수가 소수인지를 판별하는 함수를 써라.

1.4.5 상호 참조 함수

우리는 또한 교차 순환에 대응하여 상호 참조 함수를 정의할 필요가 있다. 예를 들면, nML에서는 다음과 같이 쓰는 것이 가능하다.

```
# val rec even = fn n => if n=0 then true else odd(n-1)
  and odd = fn n => if n=0 then false else even(n-1);;
val even :  int -> bool = <fun>
val odd :  int -> bool = <fun>
```

1.4.6 경우별로 정의된 함수

사실, fn 이라는 구성 방법은 우리가 지금까지 보아 왔던 것보다 더 일반적이다. 그것은 여러분이 함수를 경우별로 정의하게 해 준다. 예를 들면, true를 false에 대응시키고 false를 true에 대음시키는 논리 부정 함수는 다음과 같이 짤 수 있다.

```
# val neg = fn true => false
          | false => true;;
val neg :  bool -> bool = <fun>
```

비슷한 방법으로, 배타적 논리합은 다음과 같이 쓰여질 수 있다.

```
# val xor = fn (false,false) => false
          | (false,true) => true
          | (true,false) => true
          | (true,true) => false;;
val xor :  bool * bool -> bool = <fun>
```

이 두 개의 예제에서, 유한한 정의역을 가진 함수는 모든 가능성을 포함하는 케이스별 처리들로 정의되었다. 그러나, 함수를 케이스별로 정의하는 것은 여기서의 두 예제가 보여주는 것보다 훨씬 더 중요하다. 사실, 케이스별 정의에 변수들을 섞어 쓰는 것이 가능하다. 예를 들면, 배타적 논리합을 정의하기 위해, 우리는 다음과 같이 쓸 수도 있다.

```
# val xor = fn (false,x) => x
          | (true,x) => neg x;;
val xor :  bool * bool -> bool = <fun>
```

여기에서 함수의 각 케이스는 화살표의 오른쪽에서 사용될 수 있는 변수 x를 포함한다. 우리는 변수를 인자로 쓰는 평범한 경우와 이전의 neg 함수에서처럼 케이스별 정의를 사용하는 것의 중간 위치에 있다. 각 케이스는 가능한 값

의 덩어리에 대응된다. 예를 들면, 첫 번째 경우는 앞 필드가 false인 모든 2묶음 값들과 대응한다. 변수 x는 묶음의 두 번째 필드에 이름을 매긴다. 그렇게 함으로써, 우리는 다음의 정의에서 그 변수를 사용할 수 있다. `(false,x)`와 같은 구조는 패턴[1](pattern)이라고 부른다. 이는 어떤 조건을 만족하는 값들의 집합으로 생각할 수 있다. 그리고 받아온 인자값을 어떤 패턴으로 대응시키는 과정을 패턴 매칭(pattern matching)이라 한다.

이전의 예에서, 경우들은 함수의 정의역을 몇 개로 나눈다. 다시 말하면, 그것들은 서로 중첩되지 않고 모든 경우를 포함한다. nML은 또한 서로 중첩되는 경우 역시 허용한다. 예를 들면, 우리는 계승 함수를 다음과 같이 이전보다 더 깔끔하게 쓸 수 있다.

```
# val rec fact = fn 0 => 1
          | n => n*fact(n-1);;
val fact :  int -> int = <fun>
```

두번째 패턴이 매칭될 값에 제한이 없는 단순한 변수로 되어 있기 때문에 첫번째 경우는 두번째 경우에 포함된다. 이렇게 서로 겹치는 패턴을 쓸 경우, 사용자에 의해 먼저 쓰여진 패턴이 우선 순위를 가진다.

함수의 평범한 정의가 단순히 케이스별 정의의 특별한 형태라는 것에 주의하라. 즉, 그 경우 단 하나의 케이스가 있고 패턴은 변수 하나 또는 변수의 묶음이 된다.

계승 함수의 정의와 같은 방법으로, 피보나치 수열의 정의는 다음과 같이 나타낼 수 있다.

```
# val rec fib = fn 0 => 1
          | 1 => 1
          | n => fib(n-1)+fib(n-2);;
val fib :  int -> int = <fun>
```

함수의 케이스별 정의에서, 어떤 경우를 실수로 놓치는 일이 있을 수 있다. 또한, 케이스에 순서가 있기 때문에, 어떤 케이스들은 절대 쓰이지 않게 될 수도 있다. nML은 이런 상황이 일어나면 경고를 내 준다.

```
# (fn 0 => 1);;
Warning:  this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(1)
- :  int -> int = <fun>
# (fn x => true | 0 => false);;
Warning:  this match case is unused.
- :  int -> bool = <fun>
```

한 패턴 내에 나타나는 변수들은 반드시 서로 구분되어야 한다. 그렇다 해도, 패턴의 어떤 부분이 임의의 값과 매칭될 수 있게 특별한 기호 _를 쓰는 것이 가능하다. 이 기호는 한 패턴 내에서 여러 번 쓰일 수 있고, 다른 타입들과 대응하는 것조차 가능하다. 예를 들면, 논리곱은 다음과 같이 쓸 수 있다.

[1] fn 구성 방법에서 나타난 패턴 중 상수나 변수 한 개로 줄어들지 않는 것들은 반드시 괄호로 둘러싸야 한다.

```
# val conj = fn (true,true) => true
          | (_,_) => false;;
val conj :  bool * bool = <fun>
```

또는 다음과 같은 것도 가능하다.

```
# val conj = fn (true,true) => true
          | _ => false;;
val conj :  bool * bool = <fun>
```

1.4.7 case of 구성 방법

케이스별로 처리하는 것은 함수 정의에만 한정되어 있지는 않다. case of 구성 방법은 여러분이 표현식의 값을 구하는 데 패턴 매칭 연산을 이용할 수 있게 해 준다. 여기 문법 구조가 있다.

`case e of`
$p_1 \rightarrow e_2$
$|\ldots$
$|p_n \rightarrow e_n$

예를 들면,

```
# case (3,5) of
       (0,_) => 0
      |(x,y) => x*y;;
- :  int = 15
```

연습 문제
1.7 논리값 a,b,c를 받아 $(a \vee b \vee c)$ 를 돌려주는 함수를 3가지의 케이스를 사용하는 형태로 정의하여 써 보라.

1.5 다형성

다형성(polymorphism)은 여러분이 먼저 타입 변수(type variable)의 개념을 알고 나면 쉽게 이해할 수 있는 중요한 개념이다.

1.5.1 타입 변수

잠시 함수 `fst`(2개짜리 묶음값의 첫번째 필드를 얻는 함수)가 무슨 타입을 가질지에 대해 생각해 보자. 표현식 `fst(1,true)` 의 타입을 결정하기 위해서는, 우리는 `fst` 가 `((int * bool) → int)` 타입을 가짐을 가정해야 한다. 반면 `fst(true,1)` 의 타입을 결정하기 위해서는 `fst` 가 `((bool * int) → bool)` 의 타입을 가짐을 가정해야 한다.

그러면 nML에게는, 함수 `fst`의 타입은 도대체 무엇이란 말인가?

답을 알기 위해서는, 여러분은 간단히 다음과 같이 물어보면 된다.

```
# fst;;
- :  'a * 'b -> 'a = <fun>
```

fst를 표시하는 과정에서, nML은 'a 와 'b 라는 두 개의 타입 변수를 사용했다. fst는 'a 와 'b를 임의의 타입으로 바꿈으로써 여러분이 얻을 수 있는 모든 타입을 가질 수 있다. 이와 같은 방법으로 fst와 같은 함수의 타입을 결정하는 것은 자연스럽다. 함수의 인자로 임의의 타입을 가지는 2개의 원소를 넣는 것이 허용되는 이상, 이 원소 묶음을 인자로 받는 함수가 모든 가능성을 고려해야 한다는 것은 당연하다. 타입 규칙(우리가 타입 시스템에 대해 다루기 전까지는 이렇게 부르겠다)이 타입 변수를 쓰도록 허용하는 경우 이를 다형성이 있다(be polymorphic)고 한다. 함수의 경우 그 타입에 다형성이 있을 경우 다형성이 있다고 말한다. [2]

다형성이 있는 함수를 사용하는 것은 쉽고 자연스럽다. 여기 약간의 예제가 있다.

사영

```
# fun fst(x,y) = x;;
val fst :  'a * 'b -> 'a = <fun>
# fun snd(x,y) = y;;
val snd :  'a * 'b -> 'b = <fun>
# fun proj_23(x,y,z) = y;;
val proj_23 :  'a * 'b * 'c -> 'b = <fun>
```

항등 함수

```
# val id = fn x => x;;
val id :  'a -> 'a = <fun>
# id(3);;
- :  int = 3
# (id(id)) (id(3),id(4));;
- :  int * int = (3, 4)
```

합성 함수

```
# fun compose(f,g) = fn x => f(g(x));;
val compose :  ('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>
# let fun square(x) = x*x
  in (compose(square,square))(3) end;;
- :  int = 81
```

여러분은 nML에게 nML이 기본적으로 골라 주는 타입보다 좀 더 좁은 범위의 타입을 선택하라고 요구할 수 있다. 그렇게 하기 위해서는, e과 같은 표현식의 경우 여러분이 이 표현식의 타입을 t로 하고 싶다면 (e:t) 와 같이 쓰면 된다. 이를 제한된 표현식(constraint expression)이라 한다. 이 방법은 함수의 인자에도 적용된다. 예를 들면, 여러분은 다음과 같이 쓸 수 있다.

[2] 현재 대부분의 프로그램 언어는, 물론 정적인 타입을 쓰지 않는 LISP 등을 제외하고, 다형성이 없다는 사실에 주의하라.

```
# fn ((x:int),(y:bool)) => (y,x);;
- :  int * bool -> bool * int = <fun>
```

또는 다음과 같이 쓸 수도 있다.

```
# fn (x,y) => ((y:bool),(x:int));;
- :  int * bool -> bool * int = <fun>
```

심지어 다음과 같이 쓰는 것도 가능하다.

```
# fn (x,y) => ((y:bool),(x:int));;
- :  int * bool -> bool * int = <fun>
```

또는 비슷하게 다음과 같이 쓸 수도 있다.

```
# ((fn (x,y) => (y,x)):int*bool -> bool*int);;
- :  int * bool -> bool * int = <fun>
```

물론, nML은 사용자가 요구한 제한 사항이 타입을 결정하는 규칙과 어긋나지 않는지를 체크한다. 그것은 표현식의 구조와 어긋나지 않고 사용자의 제한 사항과도 어긋나지 않는 가장 일반적인 타입을 선택한다.

```
# fn ((x:int),y) => (y,x);;
- :  int * 'a -> 'a * int = <fun>
```

1.5.2 타입의 형성

앞 절의 `compose` 함수에서 nML이 찾아낸 타입은 이 시스템이 사용자에 의해 입력된 표현식을 분석할 때의 "지능"을 빛내준다. nML은 f와 g를 합성하기 위해서는 g의 치역과 f의 정의역의 타입이 같아야 한다는 것을 발견해 냈다.

만일 우리가 간단한 표현식만을 살펴본다면, 타입을 형성하는 규칙 역시 간단하다. 표현식 (`fn (f,g) => fn x => f(g(x))`) 에서는, 변수 f, g, x 가 나타났다. 이들의 타입을 각각 φ, ψ, χ 라 하자.

g가 x를 정의역으로 하므로, 우리는 ψ가 $(\chi \to \alpha)$ 의 타입을 가짐을 추론할 수 있고, 따라서 표현식 `(g x)`의 타입은 α가 된다.

또한 f가 (g x)를 정의역으로 하므로, 우리는 φ가 $(\alpha \to \beta)$의 타입을 가짐을 추론할 수 있고, 따라서 표현식 `f (g x)`의 타입은 β가 된다.

결과적으로, 표현식 `(fn x => f (g x))`의 타입은 $(\chi \to \beta)$가 되고 표현식 `(fn (f,g) => fn x => f(g(x)))` 의 타입은 $((\alpha \to \beta) * (\chi \to \alpha)) \to (\chi \to \beta)$ 가 된다.

f, g, x에 다른 제한 조건이 없기 때문에, 더 이상 α, β, χ 의 타입을 자세하게 할 필요가 없다.

우리가 이런 방법으로 찾아낸 타입은 합성된 타입에 어떤 치환을 해서, 즉, 이 타입에서 어떤 타입 변수를 타입 표현으로 바꾸어서 얻을 수 있는 어떤 타입보다도 일반적인 것이다.

우리가 방금 타입을 형성하기 위해 썼던 방법은 본질적으로 유니피케이션(unification) 이라는 이름으로 알려진 알고리즘에 기반하는 것이다. 우리는 이 알고리즘에 대해 5.2.4절에서 다시 다룰 것이다.

이와 같은 타입 합성 방법은 우리가 타입을 결정하려는 표현 내의 각각의 변수들에 대해 유일한 타입 변수를 대응시키는 것이다. 따라서 모든 같은 변수의 타입은 같아야 한다. 예를 들면, f가 임의의 인자 하나를 받을 수 있는 함수일 때, 표현식 `(fn f => (f 1, f true))` 의 타입은 f가 2개의 다른 타입이 동시에 되어야 하기 때문에 결정될 수 없다.

그러나, nML은 하나의 변수가 여러 개의 타입을 가질 수 있게 해 주는데, 예를 들면, 표현식 `let val id = fn x => x in (id 1, id true) end` 에서와 같은 경우이다. 이런 종류의 변수들은 `let` 구성 방법에 의해 만들어질 수 있다. 이 구성 방법을 고려하기 위해서는, 타입 합성은 계층적으로 이루어져야 한다. 표현식 `let val x =` e_1 `in` e_2 `end`의 타입을 결정하기 위해서는, 우리는 먼저 e_1의 타입 t_1을 계산해야 하고, 그 다음에 x가 t_1의 타입을 가진다는 가정 하에 e_2의 타입을 계산해야 한다. 만일 t_1이 다형성을 가진다면, 즉, 타입 변수를 포함한다면, 우리는 e_2에서 x가 나타날 때마다 이 변수의 이름을 다르게 할 수 있다.

예를 들면, `let val id = fn x => x in (id 1, id true) end` 의 타입을 결정하려면, 우리는 먼저 `id`의 타입, 즉, `fn x => x`의 타입을 결정해야 한다. 이는 $\alpha \to \alpha$가 되고, 따라서 우리는 `(id 1, id true)`에서 나타나는 각각의 `id`에 대해, 서로 다른 타입인 $\alpha_1 \to \alpha_1$ 과 $\alpha_2 \to \alpha_2$ 를 줄 수 있다. 그리고 나서 우리는 $\alpha_1 = int, \alpha_2 = bool$임을 알게 되고, 우리가 고려하던 표현식의 타입은 `int * bool`이 된다.

이 방법은 표현식 `let val id = fn x => x in (id 1, id true) end` 을 `((fn x => x) 1, (fn x => x) true)` 로 바꾸는 것과 동등하다.

연습 문제

1.8 표현식 `(fn f => fn (x,y) => (f x, f y))` 과 표현식 `let val id = fn x => x in fn (x,y) => (id x, id y) end` 의 타입을 결정하라. 여러분의 답을 확인해 보라.

1.6 문법 구조에 대해

이제 우리는 문법 구조에서의 약간의 문제점에 대해 다시 살펴보고 이전 절에서 소개했던 해결책들에 대해 좀더 명확하게 짚고 넘어갈 것이다.

1.6.1 함수에 인자를 적용할 때의 표기법

함수에 인자를 적용할 경우, nML은 모든 함수를 인자 하나만을 취하는 함수로 취급하는 방식을 택했다. 이 하나의 인자가 여러 값의 묶음인 경우에도, 이 관점은 언어에 추가적인 수정을 할 필요가 없게 해 준다. 그러면 이 장에서는 무엇을 이야기하려는 것인가?

사실상, 다형성을 가진 언어에서 이 방식의 선택이 단 하나의 가능한 선택이라는 결론을 이끌어 내기는 쉽다. 예를 들어, 항등 함수 `id`(1.5절 참조)의 경우, 이 함수가 모든 타입에 대해 적용되어야 한다는 것은 명확하고, 특히, 데카르트곱에도 적용되어야 한다. 따라서, 우리는 `id(1), id(2,3)` 과 같이 쓸 수 있다. 따라서 `id`를 인자 하나만을 가지는 함수로 취급하고, 그 인자가 간단한 값, 값의 묶음, 또는 보다 복잡한 것이 될 수 있게 하는 방법 이외에 적절한 방법을 찾기 힘들다.

이 관점에서는 함수에 인자를 적용할 때의 문법 구조는 단순히 나란히 늘어놓는 것으로 표기할 수 있다. 우리가 `id(2,3)`을 쓸 때, 괄호의 유일한 목적은 `id`의 인자의 범위를 결정하는 것이다. 괄호는 우리가 `id(1)` 또는 `id(id)` 와 같이 쓸 때는 필요하지 않다. 즉, 우리는 `id 1` 또는 `id id`[3] 와 같이 쓸 수 있다.

이 간단한 표기법은 높은 차수의 함수들을 쓸 때도 잘 적용된다. 이에 대해 살펴보기로 하자.[4]

높은 차수의 함수 사용은 우리의 표현식에 전통적인 것보다 더 복잡한 구조를 가능하게 해 준다. 우리가 `(f e)` 형태의 적용을 나타내는 표현식을 쓸 때, e 뿐만이 아니라 f도 표현식이 될 수 있다. 표현식 `(h sin) 0.1` 에서, `(h sin)` 자체가 표현식 0.1에 적용되었다. 이런 표현식들을 쓰는 방법을 간단히 하고 쓰이는 괄호의 갯수를 줄이기 위해서, 우리는 다음과 같은 규약을 만들었다.

- 함수의 적용이 연속되어 있을 경우 기본적으로 왼쪽부터 묶여서 처리된다. 다시 말하면, 함수의 적용은 왼쪽 결합법칙을 따른다. 따라서 `h sin 0.1` 과 같은 표현식은 `(h sin) 0.1`과 동등하다.

- 일반화된 함수의 타입은 기본적으로 오른쪽부터 묶여서 처리된다. 따라서 함수 h의 타입은 $(real \to real) \to (real \to real)$ 또는 $(real \to real) \to real \to real$로 표기될 수 있다.

  그러나, 그것은 $real \to real \to real \to real$로 표기될 수는 없는데, 그 이유는 이는 $real \to (real \to (real \to real))$ 로 처리되기 때문이다.

1.6.2 nML 표현식의 문법 구조 분석

함수의 적용을 단순히 나란히 나열하는 것으로 표현하는 것은 분명히 문법 구조에서 모호함을 가져온다. 예를 들면, 우리가 `f 2,3` 이라고 쓰면, 이는 `f(2,3)` 으로 읽혀야 하는가 아니면 `(f 2),3` 으로 읽혀야 하는가?

이와 같은 모호함을 제거하기 위해, 우리는 산술 연산자 사이에 쓰였던 우선 순위의 개념을 다른 연산자들, 함수 적용이나 묶음 생성과 같은, 에도 적용했다. 함수의 적용은 최고의 우선 순위를 가진다. 결과적으로, 그것은 모든 구성 방법보다 높은 우선 순위를 가지고 특히 산술 연산자들보다 높은 우선 순위를 가진다. 한편, 쉼표는 가장 낮은 우선 순위를 가진다. 다음 예제는 이 규칙들이 어떻게 시스템에서 동작하는가를 보여준다.

```
# fun d x = (x,x);;
val d :  'a -> 'a * 'a = <fun>
# fun s x = x+1;;
val s :  int -> int = <fun>
# (d 2,3);;
- :  (int * int) * int = ((2, 2), 3)
# d (2,3);;
- :  (int * int) * (int * int) = ((2, 3), (2, 3))
```

[3] 그러나, `id` 와 그 인자 사이의 빈 칸은 반드시 필요하다. 그 이유는 `id1`과 `idid` 역시 식별자의 이름이 될 자격이 있기 때문이다.(자세한 내용에 대해서는, 1.1.2절의 식별자에 대한 내용을 참조하라.)

[4] 한편, 이 표기법은 λ-calculus 에서 온 것이다.

```
# s 2*3;;
- :  int = 9
# s (2*3);;
- :  int = 7
```

nML에서, 괄호는 수학에서의 산술 연산 표현에서와 같은 방법으로 사용된다. 그들은 표현식의 구조를 명확히 해 주고, 기본 결합 법칙과 다른 결합을 가능하게 해 준다. 한편, 우리는 꼭 필요하지 않은 곳에 괄호를 추가로 쓸 수 있다. 예를 들면 우리는 시스템이 어떤 우선 순위 규칙을 가지고 있는지 확신하지 못할 때 괄호를 쓴다. 이는 숙련된 nML 프로그래머들 사이에서도 자주 있는 일이다!

1.6.3 사용자에 의해 정의된 중위 연산자

1.6.4 식별자(이름)의 유효 범위

표현식에서 쓰인 변수들은 자유 변수일 수 없다. 표현식에서 이용되기 위해서는, 변수는 반드시 정의(val 구성 방법을 이용한) 또는 어떤 패턴(fn 또는 case 구성 방법을 이용한) 내에 나타나야 한다. 각 구성 방법은 그것들이 만들어 낸 변수가 쓰일 수 있는 범위를 정의한다. 예를 들면, `let val x =` e_1 `in` e_2 `end` 와 같은 경우, x의 유효 범위는 표현식 e_2 내에서만으로 한정된다.

프로그램의 내에서, 같은 식별자가 한 번 이상 변수의 이름으로 쓰일 수 있다. 결과적으로, 우리는 변수의 쓰임을 그 정의와 혼란 없이 이어 주어야 한다. nML에서 이를 위해 우리가 따르고 있는 규칙은, 수학에서와 마찬가지로, 기호가 '쓰인' 곳에서 가장 가까운 곳의 정의를 잠조하는 것이다. 예를 들면, `let val x =` e_1 `in (fn x => e) end` 와 같은 경우, 표현식 e내에서는 구성 방법 fn에 의해 정의된 x가 우선권을 가진다.

이 간단한 규약은 우리가 자유 변수를 가진 함수(함수의 인자에 있지 않은 변수를 가진 함수)에 특별한 주의를 기울여야 하도록 만든다. 예를 들어 다음의 표현식을 살펴보자.

```
# let val x = 7 in
      let fun f z = z*x in
          let val x = true in
              f 3
          end
      end
  end;;
- :  int = 21
```

만일 우리가 함수 f가 정의된 곳을 고려한다면 이 함수가 사용한 변수 x는 `let val x = 7`이 된다. 우리가 나중에 다른 x를 다시 정의했다는 사실은 함수 f의 정의에 영향을 주지 않는다.

정의의 유효 범위에 대해 nML이 따르고 있는 규칙은 정적인 유효 범위(static scope) 또는 사전적인 유효 범위(lexical scope) 또는 원문 기준의 유효 범위(textual scope) 라 불린다.

이 유효 범위에 대한 개념은 변수의 값이 외부의 정의에 의해 영향을 받지 않음을 의미한다. 그리고 이 특성은 표현식의 타입과 의미 구조를 완전히 규칙에 따라 정의할 수 있도록 해 준다. 이전의 예에서, 우리가 f의 타입을 결정하는 방법은 만일 f내의 x가 논리값 타입으로 재정의될 수 있었다면 제대로 성립하지 않았을 것이다.

1.7 함수 타입에 대한 더 많은 내용 : 동등한 함수 공간

nML과 같은 언어에서 함수를 정의하는 것은 효율적이다. 말하자면, 함수 정의 방법은 함수 호출을 포함하는 표현식을 기계적으로 값매김할 수 있도록 해 준다. 이 효율성의 댓가로 우리는 약간의 융통성을 잃어야 한다.

예를 들면, 만일 e가 x와 y라는 2개의 자유 변수를 가지고 있는 표현식이라면(즉, `2*x+3*y`와 같은 것이라면), 이로부터 추상화될 수 있는 nML 함수에는 총 4가지가 있다. 그들은 다음과 같다.

```
# val f1 = fn (x,y) => 2*x + 3*y;;
val f1 :  int * int -> int = <fun>
# val f2 = fn (y,x) => 2*x + 3*y;;
val f2 :  int * int -> int = <fun>
# val f3 = fn x y => 2*x + 3*y;;
val f3 :  int -> int -> int = <fun>
# val f4 = fn y x => 2*x + 3*y;;
val f4 :  int -> int -> int = <fun>
```

수학적 관점에서 보면, 이 함수들 간의 차이는 경미하고 함수 공간상에서 사소한 같은 형태(isomorphism)로 취급된다. 그럼에도 불구하고, 이 함수들은 다르고, 기계적 계산의 관점에서 보면, 한 종류의 함수에서 다른 종류의 함수를 얻는 것은 우리가 명쾌하게 해 놓아야 하는 일이다.

f_1에서 f_3을 얻는 것(그리고 f_2에서 f_3을 얻는 것)은 커링(currying)[5]이라 불린다. 반대 방향으로 가는 것은 언커링(uncurrying)이라 불린다.

nML에서 이와 같은 변환 함수를 만드는 것은 쉽다.

```
# fun curry f x y = f(x,y);;
val curry :  ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# fun uncurry f (x,y) = f x y;;
val uncurry :  ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

f_1을 f_2로 바꾸는 함수와 f_3을 f_4로 바꾸는 함수는 다음과 같이 할 수 있다.

```
# fun perm f (x,y) = f (y,x);;
val perm :  ('a * 'b -> 'c) -> 'b * 'a -> 'c = <fun>
# fun perm' f x y = f y x;;
val perm' :  ('a -> 'b -> 'c) -> 'b -> 'a -> 'c = <fun>
```

[5] 이는 $\lambda - calculus$와 비슷한 결합 논리 체계를 세웠던 논리학자 Haskell Curry의 이름을 딴 것이다.

1.7.1 미리 정의된 연산자들의 타입

nML의 미리 정의된 이항 연산자들은 커링된 타입을 가지고 있다.

1.8 예제 : 표현력

이 장에서, 우리는 여러분이 함수 중심적 프로그래밍이 얼마나 위력적인지를 느낄 수 있도록 하기 위해 몇 개의 함수를 짜는 방법을 보여 줄 것이다. 이 예제들은 특히 함수를 인자로 받는 것이 어떤 잠재적 가능성을 가지고 있는지에 중점을 둘 것이다.

우리가 다룰 예제들은 대부분 수치 계산에 관한 것인데, 그 이유는 다른 종류의 예제들은 다음 장에서 소개될 타입 선언을 필요로 하는 것들이 대부분이기 때문이다. 그러나, 여기서 쓰이는 일반화된 함수들은 본질적으로 다형성을 가지며, 다른 경우에 재활용될 수 있다.

많은 수치 계산들은 함수 내에서의 반복 연산에 의존한다. 예를 들면, 연속적인 반복으로 방정식을 풀어내는 근사 해법의 경우 모두가 반복 연산을 사용한다. 이런 경우들에서, 우리는 반복이 몇 번이나 이루어질지 미리 알 수가 없고, 다만 결과에 얼마나 근접했는지를 알아내는 검사에 의존할 뿐이다. 다른 경우, 예를 들면, 수열의 9번째 원소를 알아낸다던가 하는 일의 경우는, 역시 반복이 이루어지지만, 반복의 횟수는 시작할 때 미리 알 수 있다.

이 두 종류의 반복 연산과 이와 관련한 덧셈의 개념은 우리의 급수 계산에 대한 함수 중심 프로그래밍 예제와 대응한다.

1.8.1 한정된 반복 연산

nML에서 함수 `iter`(곧 정의될)는 한정된 반복 연산과 대응한다. 그것은 2개의 인자를 가진다.

- n은 반복의 횟수이다.
- f는 반복할 함수이다.

`iter n f` 는 따라서 함수 f^n이다. 여기 정의가 있다.

```
# fun iter n f = if n=0 then id else compose f (iter (n-1) f);;
val iter :  int -> ('a -> 'a) -> 'a -> 'a = <fun>
# iter 4 (fn x => x*x) 2;;
- :  int = 65536
```

또한 우리는 반복의 시작값에 대응하는 인자 x를 사용하여 이 함수를 정의할 수도 있다.

```
# fun iter n f x = if n=0 then x else f(iter (n-1) f x);;
val iter :  int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

이 방법으로, 우리는 바로 전의 함수와 완전히 같은 함수를 정의했다. 이 함수의 타입을 보면, 인자 f의 타입이 다형성을 가진다는 것을 알 수 있고, 따라서

그것은 숫자 계산만을 하는 함수일 필요는 없다. 예를 들면, 우리는 숫자 2개의 묶음을 인자로 받고 또한 결과로 돌려주는 함수 f에 iter를 적용할 수 있다. 이 방법으로, 우리는 피보나치 수열을 계산하는 효과적인 방법을 얻을 수 있다.

실제로, 피보나치 수열의 공식은 다음과 같다.

$$u_{n+2} = u_{n+1} + u_n$$

두 개의 연속되는 피보나치 숫자를 포함하는 숫자 2개의 묶음을 사용하면, 우리는 다음을 얻을 수 있다.

$$(u_{n+2}, u_{n+1}) = f(u_{n+1}, u_n)$$

여기서 f는 (x+y,x)를 (x,y)에 대응시키는 함수이다.

```
# fun fib n = fst(iter n (fn (x,y) => (x+y,x)) (1,0));;
val fib :  int -> int = <fun> # fib 50;;
- :  int = 1037658242
```

이 함수는 각각의 피보나치 수를 단 한번만 계산하기 때문에 우리가 예전에 20페이지에서 만들었던 함수보다 훨씬 효율적이다. 결과적으로, 계산 시간은 n의 1차 함수에 비례한다.

1.8.2 한정되지 않은 반복 연산

두 번째 종류의 반복 연산은 반복할 횟수를 나타내는 숫자 n을 쓰지 않는다. 그 대신, 이 연산은 현재까지의 계산으로 얻어진 값이 만족스러운지 아니면 반복이 더 이루어져야 하는지를 알아보는 테스트를 이용한다.

```
# fun loop p f x = if (p x) then x else loop p f (f x);;
val loop :  ('a -> bool) -> ('a -> 'a) -> 'a -> 'a = <fun>
```

이 loop함수는 iter함수보다 더욱 일반적이다. 사실, 우리는 다음과 같이 loop함수에서 iter함수를 만들 수 있다.

```
# fun iter n f x = snd(loop (fn (p,x) => n=p)
              (fn (p,x) => (p+1,f x))
              (0,x));;
val iter :  int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

1.8.5절에서, 우리는 loop와 iter함수의 기능 차이에 대해 더 논의해 볼 것이다.

loop함수는 우리가 함수의 근을 찾는 방법과 같은 많은 전통적인 연속 근사법을 사용할 수 있게 해 준다. 우리는 이들 중 이분법과 뉴턴 근사법에 대해 알아볼 것이다.

1.8.3 이분법

연속이고 단조증가하는 함수 f와 f(a),f(b)가 서로 반대 부호를 가지는 구간 [a,b]에 대해, 우리는 구간 [a,b]를 둘로 나누고 이중 근을 포함하는 구간에 대해 이 방법을 계속 적용하여 f의 해를 구할 수 있다는 사실을 알고 있다. 이 방법은 구간 분할법이라고도 불리운다.

만일 ϵ이 우리가 원하는 오차의 크기라면, 해를 찾는 데 필요한 반복 연산의 수는 $\log(\frac{|b-a|}{\epsilon})$이 될 것이다.

함수 dicho는 함수 f와 숫자 2개 묶음 (a,b), 그리고 우리가 구한 근이 실제 근에 어느 정도 가까이 있는지를 정의하는 양인 ϵ을 인자로 받는다.

dicho(f,a,b,epsilon)은 loop is_ok do_better (a,b)로 나타낼 수 있으며, 이 경우 is_ok는 멈출지를 테스트하는 함수이고 do_better는 각 반복 연산의 단계를 정의한 함수이다.

```
# fun dicho (f,a,b,epsilon) =
      let fun is_ok(a,b) = abs_float(b---a)<epsilon
      and do_better(a,b) =
          let val c = (a+++b)///2.0 in
              if f(a) *** f(c) > 0.0 then (c,b) else (a,c)
          end
      in
          loop is_ok do_better (a,b)
      end;;
val dicho :  (real -> real) * real * real * real -> real * real = <fun>
```

여기 $\cos(\frac{x}{2})$의 해를 찾아 π의 근사값을 구하는 예제가 있다.

```
# dicho((fn x => cos(x /// 2.0)), 3.1, 3.2, 1e-10);;
- :  real * real = (3.14159265356, 3.14159265365)
```

1.8.4 뉴턴 근사법

먼저, 뉴턴 근사법에 대해 다시 살펴보자. f가 미분 가능한 함수라고 하자. $(x, f(x))$가 f의 그래프 위의 한 점이라고 하자. 만일 $f'(x) \neq 0$이라면, $(x, f(x))$점에서 그래프의 기울기는 x좌표가 $x - \frac{f(x)}{f'(x)}$ 점에서 x축과 교차할 것이다. 뉴턴 방법에서, 우리는 x를 $x - \frac{f(x)}{f'(x)}$ 에 대응시키는 반복 연산을 정의한다. $f(a)$와 $f(b)$가 서로 반대의 부호를 가지는 구간 [a,b]상에서 생각할 때, 만일 f'와 f''가 부호가 일정한 0이 아닌 수라면, 이 구간 내의 어떤 점에서 시작하더라도, 이 반복 연산은 수렴한다는 것이 증명되어 있다.

우선 수치 계산으로 미분값을 구하는 함수를 정의하는 것에서 시작하자. 이 함수는 작은 값 dx를 사용하여 미분의 근사값을 계산한다.

```
# fun deriv(f,dx) x = (f(x+++dx)---f(x))///dx;;
val deriv :  (real -> real) * real -> real -> real = <fun>
```

뉴턴 근사법을 구현하는 함수는 그 인자들 중의 하나로 근을 구할 함수 f를 받는다. 그리고 초기값인 start, f'를 계산할 때 쓰이는 작은 구간인 dx, 허용되는 오차의 크기인 epsilon을 또한 인자로 받는다.

```
# fun newton(f,start,dx,epsilon) =
      let fun is_ok x = abs_float (f x) < epsilon
      and do_better x =
          let val f' = deriv (f,dx) in
              (x --- f x///f' x)
          end
      in
          loop is_ok do_better start
      end;;
val newton :  (real -> real) * real * real * real -> real = <fun>
```

예를 들어 1.5 근처에서의 코사인 함수의 근을 구하여 $\pi/2$의 근사값을 구하고, 이 값에 2를 곱하면 π의 근사값을 얻을 수 있다.

```
# newton(cos,1.5,1e-10,1e-10) * 2.0;;
- :  real = 3.14159265361
```

같은 방법으로, 우리는 함수 $\log(x) - 1$의 근을 2.7 근처에서 구하여 e의 근사값을 얻을 수 있다.

```
# newton((fn x => log x - 1.0),2.7,1e-10,1e-10);;
- :  real = 2.71828182846
```

1.8.5 반복 연산에 대하여

전통적인 프로그래밍 언어들에서는, 반복 연산 함수 iter와 loop에 대응하는 개념들은 반복문을 이용하여 프로그램되었다. 실제로, iter는 보통 **for**문을 이용하여 만들어졌고, loop은 보통 **while**문을 이용하여 만들어졌다.

이 장에서 배운 내용을 사용하여 여러분은 반복분의 본체에 해당하는 부분과 멈출지를 체크하는 부분을 인자로 만들 수 있다. 즉 각각의 프로그램마다 반복 부분을 다시 쓰지 않고, 위 부분에 해당하는 함수를 만들어 인자로 넘길 수 있는 것이다.

이론적인 관점에서 보면, 이 두 가지의 반복 연산 함수는 서로 다른 기능과 특성을 가지고 있다. 만일 우리가 양의 정수만을 고려하고 함수 f의 타입을 int→int로 한정하면, 우리가 iter를 이용하여 정의할 수 있는 함수들은 1차 자기 참조 함수들로 제한된다. 반면, loop를 이용하여 정의할 수 있는 함수들은 완전한 자기 참조 함수들의 집합을 구성한다. 각각의 집합들은 논리학의 자기 참조를 다루는 이론들에서 정의되고 연구되어 왔다. 두 집합간의 가장 큰 차이는 iter로 정의되는 함수들은 그 구성상 전체 함수[6]를 만들어 낸다. 즉, 계산은 항상 끝나는 것이 보장된다. 반면, loop이 만들어 내는 함수들은 어떤 경우 끝나지 않는 계산을 수행하게 된다.

[6] 전체 함수란 그 정의역상의 모든 원소에 대해 정의된 함수이다. 반면 부분 함수란 정의역의 일부에 대해서만 정의된 함수이다.

대부분의 응용 프로그램에서 모든 경우에 정의되는 함수들만을 쓰는 것은 좋은 생각이기 때문에, 어떤 사람은 프로그래머들이 iter만을 써야 한다고 생각할지 모른다. 운 없게도, 1차 자기 참조 함수가 아닌 전체 함수들이 있기 때문에 그렇게 할 수는 없다. 이는 어떤 함수들은 loop를 이용하면 만들 수 있지만 iter를 이용해서는 만들 수 없음을 의미한다. 이 끝나는 계산에 대한 문제에 대해서는 3.4절에서 다시 다룰 것이다.

1.8.6 합계 내기

함수를 인자로 사용하면, $\sum u_n$ 또는 $\prod u_n$과 같은 수학적 표기는 바로 프로그램으로 바꿀 수 있다. 여기서 u_n은 $u_{n+1} = f(u_n)$ 형태로 정의되는 수열이다.

예를 들면, 함수 sigma는 구간 [a,b]에서의 함수 f에 대해 $\sum_{n=a}^{n=b} f(n)$을 계산해 낸다.

```
# fun sigma f (a,b) =
      if a>b then 0
      else (f a) + sigma f (a+1,b);;
val sigma :  (int -> int) -> int * int -> int = <fun>
```

더 일반적으로, 우리는 임의의 2항 연산을 사용해서 함수의 정해진 구간 내의 값들을 결합할 수 있다.

그리고, 구간의 개념은 초기값, 값의 증가율, 그리고 종료 조건으로 대체될 수 있다.

함수 summation은 모든 합계 계산을 일반화한다. 이 함수의 첫번째 인자 묶음인 (incr,test)은 인자를 증가시키는 함수와 종료 조건을 검사하는 함수를 제공한다. 두번째 인자 묶음인 (op,e)는 합계를 내는 함수와 초기값을 제공한다.(보통, 초기값은 0과 같은 중성 원소이다.)

```
# fun summation (incr,test) (op,e) f a =
      if test a then e
      else op ( (f a), (summation (incr,test) (op,e) f (incr a)) );;
val summation :
  ('a -> 'a) * ('a -> bool)
  -> ('b * 'c -> 'c) * 'c -> ('a -> 'b) -> 'a -> 'c = <fun>
```

값 증가율을 dx, 구간을 실수값 [a,b]로 하여 f(x)의 합을 구하는 함수는 다음과 같이 쓰일 수 있다.

```
# fun sum (op,e) f a b dx =
      summation ((fn x => x+++dx),(fn x => x>b)) (op,e) f a;;
val sum :
  ('a * 'b -> 'b) * 'b
  -> (real -> 'a) -> real -> real -> real -> 'b = <fun>
```

이 함수를 이용하여, 우리는 수치 계산을 이용한 적분 함수를 만들 수 있다.

$$\int_a^b f(x)dx \sim \sum_{n=0}^{n=\lfloor \frac{b-a}{dx} \rfloor} f(a+ndx)$$

```
# fun integrate f a b dx =
sum ((+++), 0.0) (fn x => f(x) *** dx) a b dx;;
val integrate :  (real -> real)
  -> real -> real -> real -> real = <fun>
# integrate(fn x => 1.0///x) 1.0 2.0 0.001;;
- :  real = 0.69389724306
```

정수 구간 [a,b]에서 1씩 증가시켜 가며 합계를 구하는 함수는 다음과 같이 만들 수 있다.

```
# fun summation_int (op,e) f a b =
summation ((fn x => x+1), (fn x => x>b)) (op,e) f a;;
val summation_int :  ('a * 'b -> 'b) * 'b
  -> (int -> 'a) -> int -> int -> 'b = <fun>
```

이 함수로, 우리는 (int→float)의 타입을 가지는 함수 f에 대해 $\sum_{n=a}^{n=b} f(n)$과 $\prod_{n=a}^{n=b} f(n)$을 구할 수 있다.

```
# val sigma = summation_int((+++),0.0);;
val sigma :  (int -> real) -> int -> int -> real = <fun>
# val pi = summation_int(( *** ),1.0);;
val pi :  (int -> real) -> int -> int -> real = <fun>
```

우리는 $n!$을 정의하기 위해 pi를 쓸 수 있다.

```
# val fact = pi float_of_int 1;;
val fact :  int -> real = <fun>
# fact 10;;
- :  real = 3628800
```

마지막으로 $\sum \frac{1}{n!}$을 정의해 보자.

```
# sigma (fn n => 1.0///fact n) 0 20;;
- :  real = 2.71828182846
```

1.9 요약

우리는 nML의 핵심부에 대해 살펴보았다. 이는 프로그램은 값매김 가능한 표현식이며, 프로그램을 수행하는 것은 표현식을 값매김하는 것과 같다는 기본 개념을 가지고, 그 위에 만들어졌다.

변수 $x_1, \ldots, x_n$을 포함하는 표현식으로부터, 우리는 함수 (fun $(x_1, \ldots, x_n) \rightarrow e$)를 만들 수 있다. 이 구성 방법은 함수 추상화**(function abstraction)**라 불리워진다. 이의 존재는 함수 중심적 언어**(functional language)**의 큰 특징이다.

함수 추상화를 사용하면, 우리는 높은 차수의 함수**(higher order function)** 를 정의할 수 있다. 즉, 다른 함수를 인자로 받거나 결과로 돌려주는 함

수를 정의할 수 있다. 이 기능은 언어에 엄청난 표현력을 준다. 특히, 프로그램을 매개 변수화하는 데 이는 매우 유용하다.

nML 프로그램은 시스템에 의해 자동으로 타입을 가지게 된다. 많은 함수들의 타입은 다형성을 지닌다. 즉, 그것들은 타입 변수를 포함한다. 이런 함수들은 여러 타입의 인자를 받을 수 있다.

1.10 더 배울 내용들

함수 중심 언어의 개념은 수학적 논리에서 기원한다. 1930년대에, Alonzo Church는 단 두 가지의 구성 방법, 즉 함수 추상화(abstraction) [7]와 함수 적용(application)만으로 구성된, 함수를 정의하는 λ-Calculus([1]) 라는 시스템을 고안해 냈다. 이 시스템은 계산 가능한 함수를 정의하기 위한 다른 여러 시스템들과 동등하다는 것이 증명되었다.(예를 들면, 튜링 머신이나 Curry의 조합 논리 시스템과 동치이다.) 이 많은 시스템들 간의 동등성은 우리가 지금 **Church**의 정리라고 부르는 공식을 이끌어 내었다. 즉, 이 다양한 시스템들이 가정했던 것들은, 사실상, 정확하게 계산 가능한 함수에 대한 직관적인 개념을 잡아냈던 것이다.

컴퓨터가 이용되기 시작하면서, 계산 가능한 함수에 대한 개념은 매우 견고한 모습을 갖추게 되었고 프로그램에 의해 계산될 수 있는 함수에 대한 개념과 비슷한 의미를 가지게 되었다. 튜링 머신은 실제로 컴퓨터가 어떻게 동작할지를 결정해 내는 이론적인 모델이 되었고, 이에 대한 연구는 복잡도에 대한 일반적인 문제들을 이끌어 냈다. 그 후 견고한 이론적 기반을 토대로 새로운 언어를 만들고 프로그램의 의미 구조를 엄격하게 정의하고자 하는 전산학자들에 의해 60년대에 λ-Calculus가 만들어졌다.

타입의 개념은 수학적 논리로부터 기원한다. 타입에 대한 이론들은 주로 λ-Calculus에서 만들어져 나왔다. nML에서 쓰인 타입의 개념은 본질적으로 Curry에 의해 만들어진 것이다. 그럼에도 불구하고, 여러분은 전산학에서 쓰이는 타입의 개념이 논리학에서 쓰이는 타입의 개념보다 훨씬 제한적이라는 사실을 기억해야 한다. Church의 생각에 따르면, λ-Calculus는 수학의 기초적인 정확한 시스템을 의미했다. 그렇게 되었을 경우에, 이의 내에서 자기 참조 함수가 정의될 수 있다는 것은 너무 강력한 시스템을 만들어 냈다. 즉, 그것은 시스템에 논리적 역설을 가져왔고, 따라서 시스템에는 모순이 생길 수 있었다. λ-Calculus에 타입의 개념을 추가함으로써, 우리는 이 역설들을 피할 수 있지만, 그렇게 했을 경우, 우리는 자기 참조 함수를 쓸 수 없기 때문에 시스템의 표현력을 매우 많이 잃어버리게 된다. 증명 이론이라고 불리는 논리학의 한 계열에서는, 타입 시스템의 구현과 그 표현력에 대한 연구가 아직 주 활동으로 남아 있다.

전산학에서, λ-Calculus는 장, 단기적으로 관심의 대상이 되어 왔다. 우선, 그것은 다른 언어들의 의미 구조를 정의할 수 있도록 해 주기 때문에, 프로그래밍 언어에 대한 일종의 참고 자료로 생각할 수 있다. 이 관점에서 보면, 모든 계산 가능한 함수를 정의할 수 있다는 사실은 λ-Calculus의 필수 불가결한 특징이 된다.

타입 시스템을 사용할 경우에 나타나는 효과 중 하나는, 그것이 시스템의 표현력을 전체 함수들의 집합의 부분집합으로 한정해 버린다는 것이다. nML

[7] λ-Calculus에서는, 변수 x에 대한 표현식 e의 추상화는 λx.e와 같이 표기된다. 이 표기법의 이름이 시스템의 이름이 된 것이다.

과 같은 언어에서, 우리는 논리학의 관점에서 보면 매우 간단한 타입 시스템을 사용했지만, val rec 이라는 특별한 구성 방법을 도입해서 자기 참조 함수를 정의할 수 있게 하였다. 이 방법으로, 우리는 함수를 사용할 때 일관성을 유지할 수 있었지만, 프로그램이 반드시 끝나도록 하는 특성은 표현력을 증가시킨 댓가로 희생시킬 수밖에 없었다.

장기적으로 볼 때, 증명 이론과 같은 분야에서 연구되고 있는 더욱 강력한 타입 시스템은 프로그램 개발에서 완전히 세련된 방법을 만들어 낼 수 있을 것이다. 그 때 쓰일 타입들은 훨씬 복잡하고, 프로그램의 특성을 표현하는 논리식과 비슷하게 될 것이다. 그렇게 되면 프로그램이 완전히 타입을 가질 수 있다는 사실은 그 타입 시스템에 의해 지정된 내용들을 프로그램이 정확하게 가지고 있다는 것을 의미할 것이다.

여기서 쓰인 언어인 nML은 ML이라 불리는 언어 계열에 속해 있다. ML의 첫 정의는 1978년까지 거슬러 올라간다. [4] R.Milner가 J.Landin의 작업에 기반하여 그것을 디자인했다. ML은 원래 프로그램의 정확성을 증명하는 LCF라는 시스템의 명령 언어였다. 프로그램의 특성이 표현된 그 논리 언어에 비교하면, ML은 중간 언어(metalanguage)라고 불릴 수 있었을 것이다. 사실상, 그것이 ML이라는 이름의 유래이다. 이때부터, ML은 크게 발전했고, 다양한 방법으로 구현되었다. 이들에는 미국 벨 랩의 SML이 있고, 또 프랑스의 CAML이 있으며, 한국의 nML이 있다. [6], [10]

2 장

데이터 구조에 대하여

프로그램은 숫자, 글, 그림, 수식, 또는 다른 프로그램들과 같은 매우 다양한 대상들을 다루어야 한다. 이 대상들은 대부분 프로그래밍의 세계에서는 낯선 것들이고 꽤나 복잡한 시스템들과 관련되어 있다.

이 대상 범주들의 각각에 대해, 프로그래머들은 그들이 쓰는 언어에 이 대상들을 나타내는 방법을 정의해야 한다. 이 방법은 반드시 효율성(프로그래머가 알고리즘에 대해 배웠던 내용 등을 이용)과 명료성 등의 조건을 만족시켜야 한다. 즉, 데이터가 구조화되는 방법은 반드시 나타내어지는 대상의 구조를 반영해야 한다. 외부로부터 가져온 대상을 나타내는 방법은 애매모호한 암호처럼 되어서는 안 되고, 개념적이며, 원래의 구조가 명확하게 나타나야 한다.

nML과 같은 타입이 존재하는 언어에서는, 대상의 본질은 그 타입에 나타난다. 따라서 다루어지는 대상의 외부 구조가 내부의 나타내는 방법에서의 타입에 반영되어야 한다는 것은 당연한 요구이다. 이를 위해서, 프로그래머는 매우 다양한 타입을 정의하는 방법을 가지고 있어야 한다.

이 장에서는, 우리는 nML이 제공하는 두 개의 주된 타입 만들기 방법을 배운다. 이는 레코드를 이용하는 방법과 구성자(constructor)의 결합을 이용하는 방법이다. 여러분은 레코드를 이름 붙여진 것들의 곱집합이라고 생각하고 구성자의 결합을 이름 붙여진 것들의 합집합이라고 생각할 수 있다.

2.1 레코드 또는 이름 붙여진 데카르트곱

우리가 복소수를 다루는 프로그램을 짜야 한다고 가정하자. 우리는 복소수의 실수부와 허수부를 나타내기 위해 `(real * real)`타입을 가지는 숫자 묶음 `(r,i)`를 사용하는 방법을 생각할 수 있다.

이 경우 우리는 덧셈을 다음과 같이 정의할 것이다.

```
# fun add_complex (r1,i1) (r2,i2) = (r1+++r2,i1+++i2);;
val add_complex :  real * real -> real * real -> real * real = <fun>
```

그러나, 보다시피, 이 표기 방법은 불편하다. 우선, 숫자 두 개의 묶음으로 표현될 수 있는 대상은 복소수뿐만이 아니다. 일정 구간을 표현하는 방법으로도 우선 숫자 두 개의 묶음을 사용하는 것을 떠올릴 수 있다. 또한, 복소수를 극

좌표 형태로 표현할 때도 숫자 두 개의 묶음을 사용할 수 있다. 만일 우리가 이 표기 방법을 사용한다면, 시스템은 복소수를 써야 할 자리에 극좌표 형태의 복소수나 심지어 일정 구간을 나타내는 숫자 묶음을 집어 넣어도 에러를 내지 않을 것이다.

또한, 데카르트곱을 이용하는 방법도 다른 이유에서 불편하다. 만일 우리가 매우 많은 정보를 가진 구조, 예를 들어, 사람들에 대한 기록 – 성, 이름, 주소, 기타 세부 사항들을 포함하는 – 등을 다룰 때 데카르트곱을 이용한다면, 이를 이용하여 프로그램하는 것은 매우 골치 아플 것이다. 왜냐 하면, 세부 사항들의 순서를 바꾸어 넣기가 매우 쉽기 때문이다. 예를 들면, 성과 이름은 둘 다 `string`형태일 것이고, 이를 바꾸어 넣었을 때 타입 검사기는 전혀 에러를 내지 않을 것이다.

이런 종류의 대상을 다루기 위해, 우리는 레코드를 쓴다. 레코드는 일종의 데카르트 곱이지만, 내부의 필드들에 모두 식별자가 붙어 있고, 우리는 이름을 써서 그것들을 다룰 수 있다.

nML에서는, 각 레코드의 타입은 사용자가 레코드를 선언할 때 이름 붙여진다. 예를 들면, 복소수를 정의하기 위해서, 우리는 다음과 같이 쓸 수 있다.

```
# type complex = re:real, im:real;;
type complex =  re:  real, im:  real
```

이렇게 함으로써 우리는 실수 부분인 `re`와 허수 부분인 `im`을 가지는 레코드 타입으로 `complex`라는 타입을 정의했다. 레코드 내의 필드들은 중괄호 내에 쓰여졌고, 각 필드들간의 구분은 쉼표로 이루어졌다.

레코드타입을 가지는 대상들의 실제 표기법은 그 타입의 표기법과 매우 비슷하다. 다만 ":" 기호가 "="기호로 대체되고, 기호 뒤에 타입이 아니고 값이 들어간다는 점이 다르다. 대상의 필드를 쓰기 위해서는, 여러분은 그 대상을 나타내는 표현식 뒤에 점을 찍고 그 뒤에 쓰고자 하는 필드의 이름을 적으면 된다.

```
# val cx1 = re=1.0,im=0.0;;
val cx1 :  complex = re=1, im=0
# val cx2 = re=0.0,im=1.0;;
val cx2 :  complex = re=0, im=1
# cx1.re;;
- :  real = 1
```

레코드를 인자로 가지는 함수들은, 레코드의 구조가 함수 인자에서 나타나는 형태, 또는 함수 내부에서 점을 이용하여 레코드의 필드를 사용하는 형태의 두 가지 모습으로 쓰일 수 있다. 예를 들면, 복소수의 덧셈은 다음과 같이 쓰일 수 있다.

```
# fn re=r1,im=i1 re=r2,im=i2
      => re=r1+++r2,im=i1+++i2;;
- :  complex -> complex -> complex = <fun>
```

그리고 다음과 같이 쓰일 수도 있다.

```
# fn c1 c2 => re=c1.re+++c2.re, im=c1.im+++c2.im;;
- :  complex -> complex -> complex = <fun>
```

물론, 레코드의 원소들 역시 사용자가 정의한 타입을 가질 수 있다.

```
# type point = xcoord:real, ycoord:real
  type circle = center:point, radius:real
  type triangle = ptA:point, ptB:point, ptC:point;;
type point =  xcoord:  real, ycoord:  real
type circle =  center:  point, radius:  real
type triangle =  ptA: point, ptB: point, ptC: point
```

연습 문제
2.1 위의 point 타입을 사용하여 평행 이동, 회전, 축소·확대를 할 수 있는 함수들을 만들어라.
2.2 앞에서 정의된 complex 타입을 사용하여 기본적인 복소수 산술 연산을 할 수 있는 함수들을 만들어라.

2.2 구성자의 결합

우리는 종종 여러 다양한 타입을 가지는 값들을 하나의 타입으로 묶고 싶어지게 된다. 이 문제는 nML에서는 구성자의 결합을 이용하여 해결된다. 이 개념은 일종의 합집합과 같은데, 합집합의 각 요소가 구성자와 대응하게 된다.

2.2.1 상수 구성자

어떤 타입들은 값의 유한한 목록으로 정의될 수 있다.

```
# type suit = Club | Diamond | Heart | Spade;;
type suit = Club | Diamond | Heart | Spade
```

"Club", "Diamond" 와 같은 이름들은 사용자에 의해 임의로 선택된 이름들이다. 이들이 타입의 정의에 나타났다는 사실은 그들에게 특별한 상태를 부여한다. 우리는 그것들을 자료 구성자(data constructor)라고 부르고, 이 이름들은 더 이상 변수 이름으로 쓰일 수 없다. nML에서는, 구성자들의 이름은 항상 대문자 또는 _(밑줄)로 시작해야 한다. 이 규칙은 코드를 좀 더 이해하기 쉽게 해준다.

타입을 정의할 때 나온 구성자는, 이후부터는 숫자 상수나 논리값과 완전히 똑같이 사용될 수 있다.

```
# Club;;
- :  suit = Club
# fn Club => 1 | Diamond => 2 | Heart => 3 | Spade => 4;;
- :  suit -> int = <fun>
```

한편, nML에서는 사용자가 다시 정의할 수 없는 이름들이 있다. bool 과 같은 타입이 그 예이다.

그림 2.1: int와 real의 겹침 없는 합집합

```
# type bool = TrUe | FaLse;;
type bool = TrUe | FaLse
```

2.2.2 인자를 가진 구성자

구성자들은 또한 인자를 받을 수도 있다. 다음의 예에서, 우리는 정수와 실수의 합집합으로 정의된 타입인 num을 소개할 것이다.

```
# type num = Int of int | Real of real;;
type num = Int of int | Real of real
```

이 정의에서, of 는 구성자의 이름과 구성자의 인자로 올 인자의 타입 사이에 놓이는 예약어이다. Int와 Real은 num 타입의 값을 만들기 위해 int나 real타입의 인자에 적용될 수 있다. 이런 방법의 정의는 다음과 같은 의미를 가진다.

> "우리는 int타입의 대상에 Int 구성자를 취하거나 real타입의 대상에 Real 구성자를 취하여 num타입을 가지는 값을 얻을 수 있다."

```
# Int(3);;
- :  num = (Int 3)
# Real(4.0);;
- :  num = (Real 4)
```

딱딱하게 말하면, Int와 Real은 각각 int와 real 타입을 num타입에 사영시킨 것이고, num은 그들의 겹침 없는(disjoint) 합집합이다. 여러분은 이 개념을 그림 2.1에서 명확히 할 수 있다.

또한, 구성자들은 case문의 정의에서 쓰인 패턴(pattern)의 일종으로 쓰일 수 있다. 어떻게 쓰는지를 보여주기 위해, 우리는 num을 위한 덧셈 연산을 정의할 것이다.

```
# val add_num = fn
      (Int m,Int n) => Int(m+n)
      | (Int m,Real n) => Real((float_of_int m) +++ n)
      | (Real m,Int n) => Real(m +++ (float_of_int n))
      | (Real m,Real n) => Real(m +++ n);;
val add_num :  num * num -> num = <fun>
```

우리는 이런 방법으로 정수와 실수를 모두 다루는 일반적인 산술 연산을 정의할 수 있다.

우리가 구성자를 이용해서 패턴을 표시할 때, 우리는 자연스럽게 이전 장에서 사용했던 패턴 매칭의 개념을 확장하게 된다. 사실, 패턴 매칭을 이용하여 정의하는 함수들의 진정한 가치는 사용자에 의해 정의된 구성자를 사용하는 것에서 얻을 수 있다. 나중에 여러분은 이 말을 실감할 수 있을 것이다.

2.2.3 단 하나의 구성자를 가지는 타입과 간략화한 타입

이제까지, 우리는 여러 개의 구성자를 가지는 타입에 대해서만 살펴보았다. 때때로, 단 하나의 구성자만을 포함하는 타입도 정의하는 것이 유용할 때가 있다. 예를 들어 각을 다루는 기하학적 프로그램을 짤 때, 우리는 다음과 같은 것을 정의하고 싶을지도 모른다.

```
# type angle = Angle of real;;
```

만일 우리가 각을 `angle`이라는 타입을 가지는 대상으로 다룬다면, 우리는 프로그램 짜는 것에 약간의 노고를 더 들여야 하지만, 그 보상으로 프로그램의 무결성을 보장하는 데 타입의 합성을 이용할 수 있다. 에를 들면, 우리는 각과 길이를 더한다던가 하는 실수를 하는 것을 막을 수 있다.

이 종류의 정의는 타입 간략화와 혼동되어서는 안 된다. nML에서 지원하는 타입 간략화의 목적은 어떤 타입을 사용하는 것을 간략하게 하기 위한 것이며, 이는 새로운 타입을 더 만들어내지는 않는다.

예를 들면 다음과 같은 정의는 단순히 `intpair`타입을 `int * int`타입과 동의어로 만드는 역할을 한다.

```
# type intpair = int * int;;
```

반면 다음과 같은 정의는 새로운 타입을 만들어 내는 역할을 한다. [1]

```
# type intpair = Intpair of int * int;;
```

2.2.4 자기 참조 타입

값들과는 달리, 타입들은 지금 정의하는 타입의 이름이 그 타입의 정의 내에서 나올 경우 바로 자기 참조 형태로 간주된다. 구성자를 이용한 타입의 자기 참조 정의는 크게 두 가지 영역에서 응용된다. nML에서, 그것들은 알고리즘을 위한 전통적인 데이터 구조를 정의하는 데 쓰인다. 또한 그것들은 추상적 문법 구조(abstract syntax)를 정의한다.

예를 들면, 말단 부분에 숫자를 가지고 있는 이진 트리는 다음과 같은 방법으로 정의된다.

```
# type inttree = Leaf of int | Node of inttree * inttree;;
```

여기 이와 같은 이진 트리의 예제가 있다.

```
# Node(Leaf 3, Node(Leaf 4, Leaf 5));;
- :  inttree = (Node ((Leaf 3), (Node ((Leaf 4), (Leaf 5)))))
```

이 트리는 그림 2.2에 그림으로 나타나 있다.

여기 트리 내의 값들의 합을 구하는 함수가 있다.

```
# val rec total = fn
      (Leaf n) => n
      | (Node(t1,t2)) => total(t1)+total(t2);;
val total :  inttree -> int = <fun>
```

그림 2.2: 이진 트리

inttree타입의 정의는 이진 트리를 정의하는 구조화된 방법이다. 이는 트리를 사용하는 알고리즘에서 쓰기는 좋지만, 특별한 이진 트리를 만드는 데는 좋지 않다. 즉, 우리는 다음과 같이 쓰기보다는

```
# Node(Leaf 3, Node(Leaf 4, Leaf 5));;
```

다음과 같이 쓰고 싶어한다.

```
# (3,(4,5));;
```

전산학에서는, 추상적 문법 구조와 구체적 문법 구조의 개념을 구분한다. 추상적 문법 구조는 대상의 구조 묘사와 내부적인 표현에 대응한다. 구체적 문법 구조는 문자열의 형태로 이를 입,출력하는 데 사용된다. 구성자를 이용한 타입은 우리가 추상적 문법 구조를 우아하게 정의하는 것을 가능하게 해 준다. (구체적 문법 구조에 대한 논의거리는 2.2.8절을 참조하라.)

우리가 정해진 형태의 표현식들을 다룰 때, "추상적 문법 구조"는 그 의미를 완전히 드러낸다. 다음의 예는 nML에서 정수값에 대한 산술 표현들을 어떻게 나타낼 수 있는지를 보여준다.

```
# type exp =
      Constant of int
      | Variable of string
      | Addition of exp * exp
      | Multiplication of exp * exp;;
```

이 타입 정의에 따르면, 산술 표현 하나는 정수 상수, 또는 변수 또는 두 산술 표현의 합, 또는 두 산술 표현의 곱으로 나타내어진다. 이를 염두에 두고, 우리는 이 산술 표현의 값을 계산하기 위한 함수 eval을 만들 것이다. 어떤 면에서 보면, 이 함수는, nML 자체의 값매김 방법을 구현한 11장의 값매김 함수의 원형이 된다.

표현식들은 변수를 포함할 수 있기 때문에, 전체 표현식의 값을 계산하기 위해서는 반드시 이 변수들의 값을 알아야 한다. 전산학에서는, 변수들을 값들과 연결하는 방법들의 집합을 환경(environment)이라고 말한다. 이 시점에서는, 우리는 환경을 string에서 int로의 함수로 나타내기로 한다. 이 값매김 함수는 환경과 값매김할 표현식에 동시에 의존한다.

```
fun eval env expression =
      case expression of
          (Constant n) => n
          | (Variable x) => env x
          | (Addition(e1,e2)) => (eval env e1) + (eval env e2)
          | (Multiplication(e1,e2)) => (eval env e1) * (eval env e2)
val eval :  (string -> int) -> exp -> int = <fun>
```

[1]타입 간략화는 나중에 모듈의 접속 형태(interface)를 다룰 때 정말 유용하다. 그것은 또한 우리가 제한 조건을 표시하는 방법을 간략하게 해 준다.

함수 eval 이 차후의 값매김 수행 함수의 원형인 것과 마찬가지로, 다음의 함수 deriv 는 규칙에 따라 표현식을 계산하는 프로그램의 원형이다. deriv 함수는 규칙에 따라서 미분 형태를 계산해 낸다. 이를 수행하기 위해, 그것은 첫 번째 인자로, 미분할 대상을 어떤 변수에 대해 미분할지를 나타내는 문자열을 받는다.

```
fun deriv var expression =
      case expression of
          (Constant n) => Constant 0
          | (Variable x) => if x=var then Constant 1 else Constant 0
          | (Addition(e1,e2)) => Addition(deriv var e1,deriv var e2)
          | (Multiplication(e1,e2)) => Addition(
              Multiplication(e1,deriv var e2),
              Multiplication(deriv var e1,e2))
val deriv :  string -> exp -> exp = <fun>
```

이 함수는 결과 표현식을 간단히 만들지 않기 때문에, 분명히 조잡하고 초보적인 수준이다.

2.2.5 다형성을 지닌 타입

사용자에 의해 정의된 타입은 다형성을 지닐 수 있다. 말하자면, 여러분은 타입 정의시 그 인자로 타입 변수를 사용할 수 있다. 이 기능은 여러분이 전통적인 자료 구조들을 nML의 타입으로 만들 수 있게 해 준다. 예를 들면, 여러분은 말단 부분의 자료형을 분명히 정하지 않은 형태의 이진 트리를 정의할 수 있다.

```
# type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree;;
type 'a tree = Leaf of 'a | Node of ('a tree * 'a tree)
```

이전 장에서 정의된 inttree는, 우리가 방금 정의한 'a tree의 특별한 경우이다. 그것은 간단히 int tree로 표현된다.

```
# Node(Leaf 3, Node(Leaf 4, Leaf 5));;
- :  int tree = (Node ((Leaf 3), (Node ((Leaf 4), (Leaf 5)))))
```

정수를 말단 노드로 가지는 이진 트리 내의 숫자값들의 합을 계산하기 위한 함수는 이제 이전과 같은 방법으로 정의될 수 있다.

```
# val rec total = fn
      (Leaf n) => n
      | (Node(t1,t2)) => total(t1)+total(t2);;
val total :  int tree -> int = <fun>
```

이 형태의 다형성은 레코드 형태의 타입에도 사용할 수 있다. 예를 들면, 사전을 정의하는 타입(즉, **key**에 의해 참조될 수 있는 정보들을 저장하는 자료 구조)은 다음과 같이 쓸 수 있다.

```
# type ('a,'b) dict_entry = content:'a, key:'b
```

같은 방법으로, 타입 간략화를 할 때도 다형성은 쓰일 수 있다.

```
# type ('a,'b) dictionary = ('a,'b) dict_entry list;;
```

2.2.6 리스트

자료 구조를 나타내기 위해 우리가 정의할 수 있는 많은 다형성 타입들 중에서, 가장 빈번하게 쓰이는 타입은 `list` 이다. 이것은 다음과 같이 정의될 수 있다.(여기서 Nil은 예약어이므로 이 코드는 실제로는 에러를 내게 된다.)

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

따라서, 리스트는 구성자 `Nil`(즉, 빈 리스트)이거나, 첫번째 원소('a 타입을 가지는)와 나머지('a list 타입을 가지는) 를 포함하는 비지 않은 리스트가 된다.

사실상, `list` 타입은 nML에서 미리 정의되어 있고, 그 원소들은 특별한 문법 구조로 표기된다. 구성자 `Cons`는 `::`(즉, `+`나 `*`와 같은 중위 연산자)으로 표기된다. 다시 말하면, 그것은 인자 둘의 가운데에 써야 한다. 빈 리스트는 `[]`으로 표기된다.

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

부수적으로, $e_1, \cdot, e_n$ 들로 이루어진 리스트를 표현하기 위해서 e_1`::`$\cdot$`::`e_n`::[]` 말고도 [e_1`,`e_2`,`$\cdot$`,`e_n`]` 과 같은 표현을 사용할 수 있다.

두 리스트를 붙이기 위한 접합 연산이 제공되며, 이는 중위 연산자 @로 표기된다.

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

또한 차후에 다룰 List 라이브러리를 사용하면 다른 여러 가지 연산을 이용할 수 있다. 예를 들어 리스트의 처음 원소를 돌려주는 `List.hd` 함수, 그 뒷부분을 돌려주는 `List.tl` 함수들을 사용할 수 있다.

우리는 2.3절에서 리스트 구조를 사용하는 방법에 대해 상세히 다룰 것이다.

2.2.7 추상적인 타입

우리가 이제까지 다루었던 타입 구성 방법에 대비하여, 이른바 추상적인 타입(abstract type)은, 엄밀히 말하면 nML에는 존재하지 않는다. 다만 모듈(module)을 사용하여 이를 쓸 수 있을 뿐이다. 그럼에도 불구하고, 이 개념은 언어에 독립적으로 프로그래밍에서 이용될 수 있는 중요한 개념이다.

대략 다음과 같은 것이다. 우리가 계산하고자 하는 대상이 하나 이상의 자연스러운 표현 방법을 가지고 있다고 하자. 예를 들면, 우리는 복소수를 실수부와 허수부, 또는 절대값과 편각으로 나타낼 수 있다.각 방법은 이들을 이용하

는 방법에 따라 각각 장단점을 가지고 있다. 이런 상황에서 가장 현명한 방법은 표현 방법을 결정하는 것을 나중으로 미루고 이에 대해 어떤 것도 가정하지 않은 채로 프로그램을 짜는 것이다.

이렇게 하기 위해, 우리는 complex라는 타입과, 복소수 상수 0, 1, i 등을 가진 변수들, 그리고 이 타입에 대한 여러 연산들을 가지고 있다고 가정할 수 있으면 된다.

complex라는 타입은 추상적으로 다루어질 것이다. 즉, 우리는 그 표현 방법을 모르지만, 그것을 위해 정의된 상수나 함수들을 통해 그것을 사용할 수 있다.

nML에서, 모듈은 우리가 파일 하나당 프로그램의 한 부분을 담을 수 있게 해 줌으로써 큰 크기의 프로그램을 짤 수 있게 해 준다. 이 모듈들은 따로따로 구현되고 컴파일되며 디버깅될 수 있다. 보통 .n 로 끝나는 파일은 모듈을 가지고 있고, .ni 로 끝나는 파일은 그 모듈의 서명(signature)이라는 것을 가지게 된다. 이 서명을 통해, 우리는 주어진 모듈 중 어떤 부분이 외부에서 접근 가능한지를 나타낼 수 있다. 즉, 모듈의 외부에서 접근 가능한 타입, 예외 처리자, 값들을 정해줄 수 있다. 이외의 다른 부분은 모듈의 외부에서 접근할 수 없게 된다. .ni 파일은 모듈 자체가 구현되기 전에 만들어질 수 있고, 또한 다른 모듈에 의해 사용될 수 있다.

모듈의 서명은 외부에서 접근 가능한 값에 대해서는 타입을 부여해 준다. 또한 타입에 대해서는, 그것들의 완전한 정의 또는 단순히 그것의 이름만을 부여해 준다. 타입에 이름만이 부여된 상태에서는, 그 모듈을 참조하는 프로그램은 이 타입을 단지 그 타입을 가지는 값들을 통해서만 사용할 수 있다. 예를 들면, 여러분은 다음과 같은 정보를 가지는 복소수 연산 모듈의 서명에 대해 생각해 볼 수 있다.

```
type complex
val cx0 :  complex (* 복소수 0 *)
and cx1 :  complex (* 복소수 1 *)
and cxi :  complex (* 복소수 i *)
and add_complex :  complex -> complex -> complex (* 덧셈 *)
and mult_complex :  complex -> complex -> complex (* 곱셈 *)
;;
```

이 복소수 연산 모듈을 사용하는 프로그램은 각 상수값들과 함수에는 접근할 수 있지만, 복소수 타입의 표현 방법에는 접근할 수 없다.

2.2.8 자료 구조의 구체적 문법 구조

2.3 리스트에 대하여

리스트에 대해 정의된 함수들은 보통 빈 리스트와 그렇지 않은 리스트 두 경우에 대한 정의를 가지는 형식을 취하게 된다.

여기 몇 개의 예제가 있다.

리스트의 길이

```
# val rec length = fn
      [] => 0
      | (a::l) => 1 + length l;;
val length :  'a list -> int = <fun>
```

두 리스트의 결합

```
# fun append l1 l2 =
      case l1 of
          [] => l2
          | (a::l) => a::append l l2;;
val append :  'a list -> 'a list -> 'a list = <fun>
```

리스트 뒤집기

```
# val rec rev = fn
      [] => []
      | (a::l) => append (rev l) [a];;
val rev :  'a list -> 'a list = <fun>
```

정수 리스트의 원소들의 합과 곱

```
# val rec sigma = fn
      [] => 0
      | (a::l) => a + sigma l;;
val sigma :  int list -> int = <fun>
# val rec pi = fn
      [] => 1
      | (a::l) => a * pi l;;
val pi :  int list -> int = <fun>
```

리스트의 모든 원소에 대해 함수 f 적용하기

```
# fun map f1 =
      case l of
          [] => []
          | (a::l) => f(a)::map f l
val map :  ('a -> 'b) -> 'a list -> 'b list = <fun>
# map (fn x => x*x) [1,2,3,4,5];;
- :  int list = [1, 4, 9, 16, 25]
```

리스트의 리스트를 평평하게 하기

```
# val rec flat = fn
      [] => []
      | (l::ll) => append l (flat ll)
val flat :  'a list list -> 'a list = <fun>
```

2.3.1 리스트를 위한 일반적인 기능들

앞의 함수들로부터, 우리는 다음과 같은 구조를 명확히 했다.

- 리스트가 비어 있을 경우, 자기 참조 함수 f의 값은 f에 의존하지 않는 값으로 정의된다.
- 리스트가 (a::l)의 모양을 가지는 경우, f의 값은 a와 f(l)에 의존한다.

결과적으로, 이 정의들은 상수 하나와 인자 2개를 가지는 함수 하나를 매개변수로 가지는 하나의 함수로 일반화 가능하다.

```
# fun list_hom e f l =
      case l of
          [] => e
          | (a::l) => f a (list_hom e f l)
val list_hom :  'a -> ('b -> 'a -> 'a) -> 'b list -> 'a = <fun>
```

다음은 이 list_hom 함수를 사용하여 이전 절의 함수들을 재정의한 것이다.

```
# val length = list_hom 0 (fn _ n => n+1)
val length :  '_a list -> int = <fun>
# fun append l1 l2 = list_hom l2 cons l1
val append :  'a list -> 'a list -> 'a list = <fun>
# val rev = list_hom [] (fn a l => append l [a])
val rev :  '_a list -> '_a list = <fun>
# val sigma = list_hom 0 (fn a b => a+b)
val sigma :  int list -> int = <fun>
# val pi = list_hom 1 (fn a b => a*b)
val pi :  int list -> int = <fun>
# fun map f l = list_hom [] (fn x l => f(x)::l) l
val map :  ('a -> 'b) -> 'a list -> 'b list = <fun>
# val flat = list_hom [] append
val flat :  '_a list list -> '_a list = <fun>
```

(함수 cons는 (fn a l => a::l) 로 정의되어 있다.)

함수 list_hom와 같은 경우는 구성자를 사용하는 타입과 자연스럽게 연결될 수 있는 특별한 경우이다. 우리는 이런 것들을 동형성(homomorphism)이라고 부른다. 여러분은 이런 동형성을 가지는 것들의 예제를, 트리와 정형화된 것들에 대해 다루는 **??**장에서 볼 수 있을 것이다.

nML 라이브러리의 리스트 관련 함수 중에는, list_hom과 비슷한 역할을 하는 fold_left 와 fold_right 함수가 있다.

fold_right 함수는 인자의 순서만 list_hom과 다르다. 여기 정의가 있다.

```
# fun fold_right f l e =
      case l of
          [] => e
          | (a::l) => f a (fold_right f l e);;
val fold_right :  ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

fold_right 함수의 인자 순서는 f 와 fold_right 함수의 타입이 비슷하도록 선택된 것이다.

이 함수의 동작은 다음과 같다. 즉, 리스트의 원소를 오른쪽 것부터 하나씩 접어서 초기값이 e 인 결과로 f라는 방법을 이용해서 접어 넣는다고 생각할 수 있다.

```
fold_right f [a1,···,an] e = f a1 (f a2 (···(f an e)···))
= list_hom e f [a1,···,an]
```

fold_left 함수는 리스트의 원소들을 역순으로 결합하는 것에 대비해 만든, 즉 왼쪽부터 접어 결과로 넣는 함수이다.

이는 다음과 같이 정의된다.

```
# fun fold_left f e l =
      case l of
          [] => e
          | (a::l) => fold_left f (f e a) l;;
val fold_left :  ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

이 함수의 동작은 다음과 같다.

```
fold_left f e [a1,···,an] = f (···(f (f e a1) a2)···) an
```

이 두 함수는 모든 이진 연산자 f, 리스트 l, 값 x 에 대해 다음과 같은 등식으로 연결된다.

```
fold_left f x l = fold_right (fn x y = f y x) l x
```

그리고,

```
fold_right f l x = fold_left (fn x y -> f y x) x l
```

이다. 우리가 쓰고자 하는 이진 연산자에 대해서, 둘 중 한 형태가 다른 형태보다 편할 수 있다.

함수 f가 결합 법칙이 성립하는 연산자인 +이고, e가 중성 원소인 경우, 다음과 같은 특별한 식이 성립한다.

```
fold_left + e [a1,···,an]
= fold_right + [a1,···,an] e
= list_hom e + [a1,···,an]
= a1 + ··· + an
```

연습 문제

2.3 비교를 수행하는 ('a -> 'a -> bool) 타입의 함수와 'a 타입을 가지는 값들의 리스트를 받아서 주어진 비교 함수에 대해 리스트 내의 최대 원소를 돌려주는 함수를 fold_left 함수를 이용하여 작성하라.

2.4 최소 원소와 최대 원소를 동시에 돌려주도록 위 함수와 비슷한 함수를 만들어 보라.

2.3.2 리스트 나누기와 정렬하기

종종 리스트로부터 어떤 특정 성질을 만족하는 부분만을 가진 리스트를 추출해 내고 싶을 때가 있다. partition 함수는 리스트를 주어진 특성에 따라 두 개의 부분 리스트로 나누어 준다. 결과는 리스트 두 개의 묶음으로 주어지고, 이 중 뒤의 리스트는 주어진 특성을 만족하는 원소들을 포함하며 앞의 리스트는 리스트의 나머지 모든 원소를 포함한다.

```
# fun partition test l =
      let fun switch elem (l1,l2) =
          if test elem then (l1,elem::l2) else (elem::l1,l2)
      in fold_right switch l ([],[])
      end;;
val partition :  ('a -> bool) -> 'a list -> 'a list * 'a list = <fun>
```

지역적으로 정의된 함수 switch 는 'a -> ('a list * 'a list) -> ('a list * 'a list) 의 타입이다. 원소 x가 리스트 묶음 (l1,l2)에 대해, 그것은 x가 test 특성을 만족하는지의 여부에 따라 이를 l1 또는 l2 에 추가한다.

filter 함수는 주어진 리스트에서 조건을 만족하는 원소만을 골라서 리스트로 만들어 돌려준다.

```
# fun filter test = compose (snd,(partition test));;
 val filter :  ('a -> bool) -> 'a list -> 'a list = <fun>
# filter (fn x => (x mod 2) = 0) [2,3,5,8,9,12,15];;
- :  int list = [2, 8, 12]
```

예를 들어 함수 partition 은 quicksort 와 같은 알고리즘을 프로그램할 때 사용될 수 있다. 이 알고리즘은 리스트에서 한 원소 (예를 들어 첫번째 원소)를 축으로 선택해서, 이를 기준으로 리스트를 둘로 나눈 후, 각각에 대해 다시 quicksort를 자기 참조 호출하여 정렬되도록 하고, 결과적으로 정렬된 마지막 리스트를 얻어내는 알고리즘이다. 인자로는 리스트 원소에 대해 순위를 매기는 연산을 위의 comp 와 같은 형식의 'a -> 'a -> b bool 타입로 넘겨 주어야 한다.

```
# fun quicksort order list =
      case list of
          [] => []
          | [a] => [a]
          | (a::l) =>
              let val (l1,l2) = partition (order a) l
              in (quicksort order l1)@(a::(quicksort order l2))
              end;;
val quicksort :  ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
```

quicksort의 case 문의 두 번째 부분은 리스트가 단 하나의 원소를 가지고 있을 때에 대응한다. 여기서 패턴 [a]은 (a::[])와 동치이다.

quicksort 알고리즘은 리스트를 평균 $n\log(n)$ 개의 연산으로 정렬한다. 최악의 경우, 이는 n^2의 시간이 걸린다. 하지만, 이는 리스트에 적용될 경우, 실제

로는 배열에 적용된 것보다 훨씬 더 비효율적인데, 이는 이 알고리즘이 임시 리스트를 계속해서 만들어 내기 때문이다. 이렇게 자료 구조를 계속 만들어 낼 때 드는 비용은 12장에서 할당(allocation)에 대해 다룰 때 분명하게 알 수 있을 것이다. 배열에 대해 동작하는 quicksort의 버전은 4.3.1절에서 제공될 것이다.

quicksort는 여러분에게 다형성뿐만 아니라 함수를 인자로 전달하는 것의 이점을 보여준다. 원소간 순서를 결정하는 함수가 인자로 전달되었기 때문에, 이 함수는 모든 순서 있는 집합에 대해 적용될 수 있다.

```
# quicksort (fn x y => x<y) [6,3,9,1,2,7];;
- :  int list = [1, 2, 3, 6, 7, 9]
# quicksort (fn x y => x>y) [6,3,9,1,2,7];;
- :  int list = [9, 7, 6, 3, 2, 1]
# quicksort (fn x y => x<y) [0.25,0.125,0.1095,0.3];;
- :  real list = [0.1095, 0.125, 0.25, 0.3]
```

fold_right 함수는 삽입 정렬(insertion sort) 역시 프로그램하기 쉽게 해 준다. 그것은 이미 정렬된 리스트에 원소 하나를 삽입하는 삽입 함수를 리스트 전체에 적용해 준다. 여기 삽입 함수가 있다.

```
# fun insert order elem list =
      case list of
          [] => [elem]
          | (a::l) =>
              if order elem a then elem::a::l
              else a::(insert order elem l);;
val insert :  ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list = <fun>
```

그리고 나면 이제 삽입 정렬 함수를 얻기 위해서는 이를 리스트에 반복적으로 적용하면 된다.

```
# fun sort order = fold_right (insert order) [];;
val sort :  ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
```

연습 문제

2.5 숫자 두 개 묶음의 리스트를 정렬하는 함수를 짜라. 단, 순서는 숫자 묶음의 첫 번째 원소끼리 비교하여 결정한다.

2.6 집합 E_1과 E_2, 이에 대해 각각 정의된 순서 $<_1$과 $<_2$ 이 있을 때, 집합 $E_1 \times E_2$에 대해 $<_1$과 $<_2$를 이용하여 정의된 사전적 순서(lexicographic order)는 다음과 같다 : $(x, y) < (x', y') \Leftrightarrow x <_1 x'$ or $(x = x'$ and $y <_2 y')$. 두 개의 비교 함수를 인자로 받고 리스트 두 개의 묶음을 사전적 순서에 따라 정렬하는 함수를 짜라.

2.3.3 리스트로 집합을 나타내기

전산학과 자료 처리에서, 우리는 종종 대상들의 집합을 어떻게 나타낼지 고민하게 된다. 우리의 선택은 이 집합에 대해 우리가 수행하고자 하는 연산들에 의해 결정된다. 예를 들면, 선택은 우리가 단순히 집합에 추가, 참조, 제거등의 연

산만을 수행할 것인지, 아니면 집합의 합집합과 교집합을 구할지에 의해 결정된다.

우리는 리스트로 집합을 나타낼 수 있지만, 이는 상당히 비효율적이다. 이 경우, 원소 하나를 찾는 데 걸리는 시간이 리스트의 길이에 비례하게 되기 때문이다. 우리는 원소간의 순서를 정할 수 없는 작은 크기의 집합에 대해서만 이 표현 방법을 선택해야 한다. 이런 경우는 원소에 접근하는 데 걸리는 시간이 표기법에 관계없이 원소 수에 비례하기 때문이다.

이제부터는 리스트에 집합의 원소를 정렬해서 저장하는 것과 그렇지 않은 채로 저장하는 것의 차이에 대해 알아볼 것이다.

정렬되지 않은 리스트로 집합 나타내기

우리가 집합을 다루기 위해 정의하는 모든 함수들은 집합 내의 원소들간의 관계를 나타내는 함수 하나를 인자로 받는다. 이 함수는 `'a * 'a -> bool` 의 타입을 가지고, 원소끼리 동치인지를 비교하는 역할을 할 것이다.

함수 `member`는 원소 하나가 집합에 속하는지를 검사한다.

```
# fun member equiv e list =
      case list of
          [] => false
          | (a::l) => equiv(a,e) orelse member equiv e l;;
val member :  ('a * 'b -> bool) -> 'b -> 'a list -> bool = <fun>
```

함수 `rem_from_list`는 리스트로부터 한 원소를 제거한다. 만일 이 원소가 존재하지 않으면, 리스트에는 변화가 없게 된다. 만일 이 원소가 리스트에 두 번 이상 나오면, 모든 해당 원소들은 지워지게 된다. 이 함수는 리스트에서 중복된 원소를 제거하여 집합으로 만드는 `make_set` 함수에 사용된다.

```
# fun rem_from_list equiv e list =
      case list of
          [] => []
          | (a::l) =>
              let val l' = rem_from_list equiv e l
              in if equiv(a,e) then l' else a::l'
              end;;
val rem_from_list :  ('a * 'b -> bool) -> 'b -> 'a list
  -> 'a list = <fun>
# fun make_set equiv list =
      case list of
          [] => []
          | (a::l) => a::make_set equiv (rem_from_list equiv a l);;
val make_set :  ('a * 'a -> bool) -> 'a list
  -> 'a list = <fun>
```

함수 `rem_from_set`는 집합에서 원소 하나를 제거한다. `rem_from_list`와는 다르게, 이 함수는 같은 원소는 1개만 존재한다고 가정한다.

```
# fun rem_from_set equiv e list =
```

```
      case list of
          [] => []
          | (a::l) =>
              if equiv(a,e) then l
              else a::(rem_from_set equiv a l);;
val rem_from_set :  ('a * 'a -> bool) -> 'a -> 'a list
  -> 'a list = <fun>
```

함수 add_to_set은 집합에 원소 하나를 추가한다.

```
# fun add_to_set equiv e l =
      if member equiv e l then l else e::l;;
val add_to_set :  ('a * 'a -> bool) -> 'a -> 'a list
  -> 'a list = <fun>
```

함수 union은, 물론, 두 집합의 합집합을 구하고, intersect는 두 집합의 교집합을 구한다.

```
# fun union equiv (l1,l2) = fold_right (add_to_set equiv) l1 l2;;
val union :  ('a * 'a -> bool) -> 'a list * 'a list
  -> 'a list = <fun>
# fun inter equiv (l1,l2) = filter (fn x => member equiv x l2) l1;;
val inter :  ('a * 'b -> bool) -> 'b list * 'a list
  -> 'b list = <fun>
```

위 두 함수들이 집합의 크기의 제곱에 비례하는 수행 시간을 가진다는 것에 주의하라.

정렬된 리스트로 집합 나타내기

여러분이 기대하던 대로, 정렬된 리스트로 집합을 나타내는 방법은 순서 있는 집합을 표현하는 데 사용된다. 이 표현 방법은 추가, 탐색, 제거 연산에 걸리는 시간을 반으로 줄여 주고, 특히 합집합과 교집합을 구하는 데 걸리는 시간을 획기적으로 줄여 준다. 우리는 이 연산들을 집합의 크기에 비례하는 시간 안에 수행할 수 있다. 또한, 정렬되어 있을 경우 배열을 사용하면 우리는 이진 탐색 알고리즘을 사용하여 탐색 연산에 걸리는 시간을 로그 형태까지 줄일 수 있다. (이에 대해서는 4장을 보라.) 그러나, 그 경우는 배열은 자유롭게 길어질 수 없기 때문에 추가 연산이 쉽지 않게 된다. 6장에서, 여러분은 균형잡힌 트리를 이용하여 어떻게 순서 있는 집합을 효과적으로 표현할 수 있는지 볼 수 있을 것이다.

정렬된 리스트를 이용하여 나타내어진 집합에 적용되는 함수들은 인자로 (order,equiv)의 집합 내 원소간의 관계를 나타내는 함수 묶음을 받는다. 이 중 첫 번째 함수는 원소간 순서를 결정하고, 두 번째 함수는 원소간 동치 여부를 결정한다. 물론 우리는 3개의 값(작다,크다,같다)중 하나를 돌려주는 함수를 이용하여 이 두 함수를 하나로 합칠 수 있고, 이에 대해서는 6장에서 다룰 것이다.

member함수는 찾는 원소보다 큰 원소를 처음 만났을 때 멈추도록 다시 쓰여졌다.

```
# fun member (order,equiv) e list =
      case list of
          [] => false
          | (a::l) =>
              if order(a,e) then member (order,equiv) e l
              else equiv(a,e);;
val member :
('a * 'b -> bool) * ('a * 'b -> bool) -> 'b -> 'a list
  -> bool = <fun>
```

함수 add_to_set은 50쪽의 insert함수의 변형이며, 리스트에 이미 원소가 있을 경우를 처리하기 위해 변경되었다.

```
# fun add_to_set (order,equiv) elem list =
      case list of
          [] => [elem]
          | (a::l) =>
              if order(elem,a) then elem::a::l
              else if equiv(elem,a) then a::l
              else a::add_to_set (order,equiv) elem l;;
val add_to_set :
('a * 'a -> bool) * ('a * 'a -> bool) -> 'a -> 'a list
  -> 'a list = <fun>
```

함수 union과 inter는 리스트가 정렬되어 있기 때문에 각 리스트를 한번씩만 통과하면 된다.

```
# fun inter (order,equiv) = fn
      ([],_) => []
      | (_,[]) => []
      | ((l11 as a1::l1),(l12 as a2::l2)) =>
          if equiv(a1,a2) then a1::inter (order,equiv) (l1,l2)
          else if order(a1,a2) then inter (order,equiv) (l1,l12)
          else inter (order,equiv) (l11,l2)
val inter :
('a * 'b -> bool) * ('a * 'b -> bool) -> 'a list * 'b list
  -> 'a list = <fun>
# fun union (order,equiv) = fn
      ([],l2) => l2
      | (l1,[]) => l1
      | ((l11 as a1::l1),(l12 as a2::l2)) =>
          if equiv(a1,a2) then a1::union (order,equiv) (l1,l2)
          else if order(a1,a2) then a1::union (order,equiv) (l1,l12)
          else a2::union (order,equiv) (l11,l2)
val union :
('a * 'a -> bool) * ('a * 'a -> bool) -> 'a list * 'a list
  -> 'a list = <fun>
```

2.3.4 리스트에서 탐색하기

리스트에서 어떤 특징을 가진 원소 하나를 찾는 것은 매우 유용하다. 49쪽에서, 우리는 리스트에서 주어진 특성을 만족하는 원소들을 추출해 주는 `filter` 함수를 정의했다. 그러나, 많은 경우에, 우리는 주어진 특성을 만족하는 첫번째 원소 하나만을 찾는 것만으로 만족하게 된다. 예를 들면, 리스트 내에 그 특성을 갖는 원소가 하나밖에 없는 경우 등이다. 그러면 이제 문제는 주어진 특성을 만족하는 원소가 없을 경우 이 함수가 어떤 값을 돌려줄까 하는 것이다.

더 일반적으로 말하면, 문제는 부분 함수를 정의하는 방법을 아는 것이다. 부분 함수란, 정의역의 어떤 곳에 대해서 정의되지 않은 함수를 말한다. 이런 함수를 정의하는 방법 중의 하나는 다음과 같은 타입을 사용하는 것이다.

```
# type 'a option = None | Some of 'a;;
type 'a option = None | Some of 'a
```

그리고 `'a -> 'b option` 의 타입을 가지는 부분 함수를 만들면 된다. 여기 이 방법으로 만든 리스트 검색 함수가 있다.

```
# fun find prop list =
      case list of
          [] => None
          | (a::l) => if prop a then Some a else find prop l;;
val find :  ('a -> bool) -> 'a list -> 'a option = <fun>
# find (fn x => (x mod 2) = 0) [3,1,7,8,13,4];;
- :  int option = (Some 8)
# find (fn x => (x mod 2) = 0) [3,1,7,13];;
- :  int option = None
```

이와 같은 함수들을 사용하는 가장 일반적인 예는 우리가 관계 리스트(association list)를 사용할 때 일어난다. `'a -> 'b` 의 타입을 가지며 정의역이 유한한 함수는 `('a * 'b) list`의 타입을 가지는 리스트로 표현 가능하다. 이제 이 리스트에서 왼쪽 부분이 값 v, 타입 `'a` 인 원소를 검색하면, 우리는 이 변수 v와 대응하는 값을 얻어낼 수 있다.

```
# fun associate v l =
      case find (fn (x,y) => x=v) l of
          None => None
          | Some (x,y) => Some y;;
val associate :  'a -> ('a * 'b) list -> 'b option = <fun>
```

부분 함수를 처리하는 다른 방법은 에러 또는 예외 처리자(exception)의 개념을 언어에 추가하여 함수가 값을 돌려주지 않는 경우에 대비하는 것이다. 이 방법은 여러 개의 예외적 상황을 묶어 처리할 수 있다는 장점이 있다. 함수에서 예외 처리자는 계산 중간의 예외적 상황, 인터럽트 등을 처리할 수 있게 해 주고 특별한 경우에 대해 일반적인 루틴을 따라가지 않을 수 있게 해 준다. 우리는 4장에서 이 개념에 대해 다룰 것이다.

연습 문제

2.7 다항식은 그 계수의 리스트로 다음과 같이 나타낼 수 있다 : `[a0,a1,···,an]`.

이 방법으로 나타내어진 다항식을 더하는 함수와 곱하는 함수를 짜라. 또한 변수에 대한 값이 주어지면 다항식의 값을 구하는 함수를 짜라. 또한 다항식의 미분값과 적분값을 구하는 함수를 짜라.

2.8 위 문제에서의 다항식은 다항식의 크기가 0이 아닌 항들에 의해 정해지는 것이 아니고 다항식의 차수에 의해 정해진다는 점에서 불편하다. 0이 아닌 항들을 보관하지 않는 nML 다항식 타입을 정의하라. 새로운 다항식 표현에 대해 위 연산을 수행하는 함수들(더하기, 곱하기, 값매김, 미분, 적분)을 정의하라.

2.9 43쪽의 `'a tree`에 대해 이 타입의 트리에 어떤 값들의 집합을 대응시키는 함수를 만들어라.

2.10 우리는 `'a tree` 타입의 트리 내의 노드 또는 말단노드(leaf)들을 2개중 하나를 선택하는 동작들의 리스트와 각각 대응시킬 수 있다. 예를 들면, 만일 우리가 왼쪽을 글자 `l`로 표시하고 오른쪽을 글자 `r`로 표시하면, 트리 `Node(Leaf 3,Node(Leaf 4,Leaf 5))`에서, `rl`은 `Leaf 4`로 가는 길을 나타내게 된다. 트리 `t`와 이런 리스트 `u`가 주어지면, 우리는 `u`에 대응하는 부분 트리 $t_{/u}$를 생각할 수 있다. 만일 이 부분 트리가 존재하지 않으면, `u` 는 $u = u_1u_2$ 로 나누어질 수 있는데, 여기서 u_1은 $t_{/u1}$ 가 원 트리의 말단노드가 되도록 선택된다. `L`과 `R` 이라는 두 값을 가지는 타입 `direction`을 정의하고, 방향 리스트 `dl` 과 트리 `t` 를 받아 $t_{/dl}$ 을 제공하는 `direction list -> 'a tree ->'a tree option`의 타입을 가지는 함수를 정의하라.

2.11 만일 `'a tree`타입의 트리 내의 모든 값이 서로 다르다면, 트리는 그것이 포함하는 모든 값에 대해 2진 압축 코드를 제공한다. 예를 들면, 우리가 0이 왼쪽을 의미하고 1이 오른쪽을 의미한다고 정하면, 그림 2.3의 트리는 이진 코드 `(a -> 000), (b -> 001), (c -> 010), (d -> 011), (e -> 1)` 와 대응한다. 이 이진값들을 `'a tree` 타입의 이진 트리를 이용하여 해독해서 `'a`타입의 값들을 다시 얻으려면, 단순히 이진값들을 이전 문제에서의 `direction` 에 해당하는 값으로 생각해서 트리를 타고 내려가다가, 말단노드에서 `'a`타입의 값을 하나 얻은 후 다시 트리의 뿌리로 돌아가서 이 작업을 반복해 나가면 된다. 우리의 예제에서 "0011010" 은 "bec"로 해독된다. 이제 `direction list` 의 타입을 가지는 이진값들의 집합을 트리를 이용하여 해독하는, `direction list -> 'a tree -> 'a list option` 타입을 가지는 함수를 짜라.

2.12 우리가 이진값으로 압축하고자 하는 한 값들의 집합에 대해서, 기본적으로 수많은 압축 방법에 대응하는 수많은 이진 트리가 존재할 수 있다. 우리는 압축된 메세지의 길이를 비교해서, 서로 다른 압축 방법을 비교할 수 있다. Huffman에 의해 제시된 다음 방법은 이 관점에서 볼 때 최적의 이진 코드를 찾아낸다. 우선 우리가 압축할 $(x_1, \cdots, x_n)$ 라는 기호들을 가지고 있다고 하자. 그리고 이 값들이 나타날 확률은 각각 $(p_1, \cdots, p_n)$ 이라고 하자. 그러면 우리는 $[(Leafx_1, p_1), \cdots, (Leafx_n, p_n)]$ 와 같은 리스트를 만들 수 있다. 그리고 나서 우리는 이 리스트에 다음과 같은 동작을 반복해서 수행할 수 있다. 우선, 리스트에서 p_a와 p_b가 가장 작은 두 값묶음 $(a, p_a), (b, p_b)$를 고른 후, 이를 $(Node(a, b), p_a + p_b)$ 로 대체한다. 이를 반복하다 리스트가 결국 단 하나의 값묶음을 가지게 되면, 이 값묶음의 앞쪽 원소가 원 집합에 대한 최적의 암호화 트리이다. 이 알고리즘을 구현하는 `('a * real) list -> 'a tree` 타입의 함수를 짜라.

그림 2.3: 암호화($a- > 000, b- > 001, c- > 010, d- > 011, e- > 1$)를 위한 트리

2.4 요약

우리는 이제 nML의 주 자료 구조에 대한 소개를 마쳤다. 즉, 이름 붙여진 데카르트곱(즉, 레코드)과, 이름 붙여진 것들의 합집합(즉, 구성자를 이용한 타입)에 대해 알아보았다.

사용자가 정의한 타입들은 이름 붙여지고, 정의된 후에는 기본으로 정의된 타입들과 같은 기능을 가진다.

데카르트곱과 합집합 형태의 타입들은 인자를 받을 수 있다. 사용자가 쓸 수 있는 타입 환경에서, 인자를 받지 않는 타입은 상수 타입 구성자로 생각할 수 있고, 인자를 받는 타입은 변수를 가지는 구성자로 생각할 수 있다. 모든 타입 정의는 타입 환경에 추가된다.

또한, 모든 타입 정의는 함수를 케이스별로 정의할 때 사용할 수 있는 패턴의 문법 구조에도 추가된다. 이 패턴 매칭에서의 유연성은 자료 구조에 따라 다양한 경우를 가지는 함수를 만드는 것을 자연스럽게 해 준다. 그것은 또한 언어가 심볼릭 프로그래밍에 좀더 잘 적용될 수 있게 해 준다.

2.5 더 배울 내용들

아마도 여러분은 파스칼(1971) 언어에서 이름 붙여진, 필드를 가진 데카르트곱을 레코드로 부르는 데 익숙해졌을 것이다. 하지만 그것은 PL/1(1965), SIMULA(1967), 심지어 코볼(1960) 에서도 등장했던 개념이다. 그리고 이 개념은 이제 수많은 언어에 퍼져 있게 되었다. 그렇지만, 이 타입들에 대해 인자를 사용할 수 있게 한 것은, ML류 언어에서 지원하는 다형성으로 인해 가능한 특별한 기능이다.

구성자를 이용해서 타입을 만드는 개념은 80년대 초반 HOPE라는 언어에서 소개되었고, ML을 비롯한 다른 함수 중심 언어에서 다시 사용하게 되었다.

ML은 타입 정의를 생성적(generative)으로 만들기 위해 많은 근본적인 선택을 하였다. 타입 정의가 생성적이라는 말은, 각 정의가 이전에 존재하던 타입들과는 다른 새로운 타입을 만들어 낸다는 것을 의미한다. 심지어 새로운 타입 정의가 이전에 존재하던 타입 중 하나와 완전히 같은 구조를 가지더라도 이는 참이다. 이 선택은 타입을 구조적(structural) 관점에서 다루는 것과 대비되는데, 이 방식은 타입이 이름 붙여지지 않고 순전히 그 구조로만 정의되며, 따라서 구조적으로 같은 타입은 같게 취급하게 된다.

ML이 한 선택은 타입을 정의한다는 것은 기본적으로 이 대상을 어떤 특정한 목적으로 사용할 의도가 있다는 생각에 기반하고 있다. 각 타입은 어떤 개념과 대응하고, 그 대상들은 다른 대상들과 구분 가능한 특별한 기능과 특징을 가지고 있다. 타입에 이름을 붙임으로써, 우리는 이 명확히 나타나지 않은 특징들을 모아서 요약하게 된다고 할 수 있다.

이에 반해, 구조적 관점은 타입의 목적이 완전히 그 구조 내에 포함되어 있다는 생각에 기반한다. 즉, 그 타입을 구성하는 필드들의 타입에 정보가 모두 들어있다고 생각하는 것이다. 그럼에도 불구하고, 우리는 타입의 이름에 대해

다루었던 절에서 볼 수 있었던 일종의 모호함이 구성자나 레코드 필드의 이름에 대해 다룰 때도 나타날 수 있다는 사실에 주의해야 한다.

이 두 가지 입장을 조율하기 위해, 우리는 기본적으로 논리적 관점을 채택하기로 했다. 이 관점에서, 타입은 정형화되어 표현된 특징들의 집합으로 정의된다. 따라서 어떤 대상은 이 특징들을 만족해야만 주어진 타입에 속할 수 있다. 그러나, 이 접근 방법은 타입 합성이 자동 증명을 수행하기 위한 몇 가지 방법을 가정하고 있다는 사실과 맞지 않는다. 이는 오늘날 우리가 가지고 있는 지식을 훨씬 벗어나는 것이다.

생성적인 타입(constructive type)에 대한 수많은 연구가 현재 진행되고 있다. 아마도 가까운 장래에, 그 연구들은 프로그래밍에서 쓰이는 타입이 좀 더 많은 논리적 내용들을 포함할 수 있게 해 줄 것이다.

3 장

의미 구조(semantics)에 대하여

이 장에서는 우리가 이전의 장들에서 설명했던 함수 중심 언어의 의미 구조에 대해 다룬다. 이 언어의 의미 구조라는 것은, 이 언어에서 쓰인 표현식들의 뜻을 정의하는 것이다. 다시 말하면, 각 표현식들의 값을 정확하게 정의하는 것이다. 표현식과 그 값 사이의 관계는 다시 쓰기(rewrite) 규칙에 의해 만들어진다. 즉, 표현식의 의미를 바꾸지 않고 그 겉모양만을 변환하는 규칙에 의해 관계가 정해진다. 우리는 이 규칙들에 대해 3.1절에서 살펴보고, 논의할 것이다.

이 다시 쓰기 규칙들은 비결정적(non-deterministic)이다. 다시 말하면, 일반적으로, 고려 대상이 되는 어떤 표현식에 대해서도, 적용할 수 있는 여러 개의 규칙들이 있다. 이 규칙들의 일관성은 이 규칙들이 수렴하는(convergent) 시스템을 이룬다는 사실에 기반한다. 즉, 우리가 어떤 단계에서 어떤 규칙을 선택하건 간에, 항상 서로 다른 계산이 같은 표현식으로 수렴하게 할 수 있다. 이 특성은 무한한 계산이 이루어질 가능성을 배제하지는 못한다. 그러나 이는 한 표현식이 두 개의 서로 다른 값을 가질 가능성을 배제해 준다. 표현식의 값(만일 존재한다면)은, 따라서 유일하다. 하지만 우리는 여기서 이 수렴 특성을 증명하지는 않을 것이다. 이의 증명에 대한 참고 문헌은 이 장의 마지막에서 찾을 수 있다.

실제로, 언어의 값매김 장치를 구현하려면, 우리는 각 단계에 대해 어떤 다시 쓰기 규칙을 고를 것인가를 결정하는 전략을 정의해야 한다. 즉, 우리는 모든 가능한 다시 쓰기 형태 중 한 가지를 선택해야 한다. 3.2절에서 우리는 여러 전략을 제시하고 이들을 비교할 것이다. 같은 표현식이 무한한 계산을 수행할 수도 있고 유한한 계산을 수행할 수도 있기 때문에, 우리는 반드시, 표현식의 값이 존재할 경우 이를 구해내는 완전한 전략과, 어떤 경우에 유한한 계산이 존재하는데도 불구하고 무한한 계산을 수행하는 불완전한 전략을 구분해야 한다. nML에서의 전략 – 값에 의한 호출 – 은 불완전한 전략이고, 우리는 왜 이런 선택을 했는지를 설명해야만 한다.

3.3절에서는 표현식의 값을 정형화하여 정의하기 위해 무엇을 미리 해야 하는지를 다룬다.

의미 구조를 정형화하여 정의하고 나면, 우리는 프로그램의 특성들을 증명하기 위해 논리적으로 생각할 수 있게 된다. 3.4절에서, 우리는 프로그램의 특성에 대한 증명을 하기 위한 방법을 고안해 낸다. 이 방법은 자기 참조 원리와 동치만들기 논법(equational reasoning)에 기반한다.

마지막으로, 3.5절에서는, 우리는 타입 합성에 대한 내용을 다룬다. 타입 합성 규칙들은 표현식과 대응될 수 있는 어떤 타입의 집합들을 정형적으로 정의한다. 또한 우리는 타입과 타입 합성에 대한 일반적인 특성들에 대해서도 다룬다.(즉, 주 타입(principal type)의 존재 여부와 타입과 의미 구조간의 호환성에 대해 다룬다.)

3.1 값매김에 대하여

값매김은 표현식에서 시작하여 값에서 끝나는 일련의 변환들의 연속으로 정의될 수 있다. 이 변환들은 **다시 쓰기**(rewrite)라고 불리고, 다시 쓰기들의 연속은 **줄이기**(reduction)라고 불린다.

3.1.1 값매김, 표현식을 다시 쓰는 것

예를 들면, 만일 우리가 순수 산술 표현식들에만 관심이 있다면, 다시 쓰기는 하위 표현식들 중 연산자 하나와 숫자 상수 2개로 이루어진 것을 그 연산의 결과로 바꾸어 넣는 것으로 이루어진다.

$$(2+3)*(4+5) \rightarrow (2+3)*(9) \rightarrow 5*9 \rightarrow 45$$

여기서 우리는 하위 표현식 여러 개 중에서 줄이고자 하는 것을 선택할 수 있는 자유를 가진다. 즉, 우리는 위 식을 왼쪽부터 시작하여 오른쪽으로 가면서 계산할 수도 있다.

$$(2+3)*(4+5) \rightarrow (5)*(4+5) \rightarrow 5*9 \rightarrow 45$$

이와 같은 줄이기 방법이 다음과 같은 특성을 가진다는 것은 명백하다.

- **유한성**(Finiteness) : 모든 줄이기는 끝난다.
- **일관성**(Consistency) : 모든 줄이기는 같은 결과를 가진다.

일관성은 **수렴성**(convergence)이라는 좀 더 일반적인 특성에 기인한다. 다시 쓰기 규칙들의 집합은, 만일 표현식 e가 e_1과 e_2로 다시 쓰일 수 있는 모든 상황에서 다시 e_1과 e_2가 어떤 표현식 e_3으로 다시 쓰일 수 있다면, 수렴성을 가지게 된다. 이 특성은 그림 3.1에 잘 나타나 있다. 그림 3.2와 같이 표현식 $(2+3)*(4+5)$ 의 모든 가능한 다시 쓰기들의 집합을 표현하게 되면, 이 특성이 무엇을 의미하는지 명확히 알 수 있다.

우리가 산술 표현에서 자기 참조 함수의 다시 쓰기를 지원하는 함수 중심 언어로 옮겨오면, 모든 줄이기의 유한성은 보장되지 않는다. 그럼에도 불구하

그림 3.1: 수렴성

그림 3.2: 다시 쓰기(줄이기)에 대한 그래프

고, 수렴성은 보장된다. 그러나, 무한한 줄이기가 존재할 경우, 수렴성은 일관성을 보장하지는 않는다. 즉, 한 표현식에 대한 서로 다른 줄이기 방법이 항상 같은 결과를 가져오지는 않는다. 대신, 우리는 좀더 약해진 형태의 일관성을 정의하게 된다.

> **약한 일관성**(weak consistency) : 끝나는 모든 줄이기는 같은 결과를 가진다.

분명히, 무한한 줄이기를 지원하는 다시 쓰기 시스템에서는, 무한한 줄이기와 유한한 줄이기를 동시에 가지는 표현식이 존재할 수 있다. 이와 같은 시스템은 약한 일관성만을 가질 수밖에 없다.

이 약한 일관성은 함수 중심 언어에서만 보장된다. 이는 프로그램의 증명에 필수적이다.

간단한 산술 표현이 아닌 nML과 같은 언어의 경우, 우리는 언어의 모든 구성 방법(construction)에 대해 다시 쓰기 규칙을 정의해야 한다. 우선은, 우리는 함수의 추상화(abstraction)와 적용(application)에 집중할 것이다. 그것만으로도 여러분은 값매김에서의 문제점이 무엇인지를 충분히 파악할 수 있다.

처음에, 우리는 단순한 함수 추상화, 즉, `(fn x => e)`의 꼴을 가지는 변수 하나짜리 함수 추상화만을 다룰 것이다. 이와 같은 함수의 적용과 관련된 다시 쓰기 규칙은 **치환**(substitution)만을 이용한다. 치환이란, 어떤 표현식 내의 변수 하나가 다른 하나의 표현식으로 대체되는 것이다.

표현식 e_1내의 변수 x를 표현식 e_2로 치환하는 것은 e_1내의 모든 자유 변수 x를 e_2로 대체하는 것으로 이루어진다. 결과는 $e_1[\mathrm{x} \leftarrow e_2]$ 로 표시된다. x가 자유 변수라는 말은, 변수 x가 변수 x에 대한 함수 추상화 안에 있지 않음을 의미한다. 좀 더 정확하게는, 치환은 다음과 같이 정의된다.

- `x[x ←  e] = e`
- `y[x ←  e] = y`
- $(e_1\ e_2)$`[x ←  e] =` $(e_1$ `[x ←  e])(`e_2 `[x ←  e])`
- `(fn x ⇒ `e_1`)[x ←  `e_2`] = (fn x ⇒ `e_1`)`
- `(fn y ⇒ `e_1`)[x ←  `e_2`] = (fn y ⇒ `e_1`[x ←  `e_2`])`

그리고 여기 두 개의 예제가 있다.

```
(y + x y) [x ← (fn x ⇒ x)] = (y + (fn x ⇒ x) y)

(x + (fn x ⇒ x) y)[x ← 2*y] = (2*y + (fn x ⇒ x) y)
```

두 번째 예에서, 두 번째 x(즉, 하위 표현식 `(fn x ⇒ x)`의 안에 나타난 x)는 자유 변수가 아니기 때문에 치환에 의해 영향을 받지 않았다. 우리는 `(fn x ⇒ x)`를 동치인 표현식 `(fn z ⇒ z)`로 대체해서, 이 하위 표현식이 치환에 영향을 받지 않는다는 사실을 강조해 줄 수 있다.

우리의 치환에 대한 정의는 완전하지 않은데, 그 이유는, 변수 y를 인자로 가지는 함수 내로 자유 변수 y를 가지는 표현식이 지환되어 들어갔을 경우 일어날 수 있는 이름 충돌을 고려하지 않았기 때문이다. 예를 들면, 치환 `((fn y ⇒ x*y) x)[x ← 2*y]` 는 `((fn y ⇒ (2*y)*y)(2*y))` 가 되어서는 안 된다.

이 문제를 피하기 위해서, 우리는 이전에 `(fn x ⇒ x)`를 `(fn z ⇒ z)` 로 대체했던 것과 같이 변수 이름을 바꿀 수 있다는 사실을 이용할 수 있다. 따라서,

```
((fn y ⇒ x*y) x)[x ← 2*y]
```

를 줄이기 위해, 우리는 치환을 수행하기 전에 이를

```
((fn z ⇒ x*z) x)[x ← 2*y]
```

와 같이 바꾼다. 그렇게 하는 것은 다음과 같은 결과를 낸다.

```
((fn z ⇒ (2*y)*z) (2*y))
```

함수 추상화의 적용을 위한 줄이기 규칙은 다음과 같다.

- `(fn x ⇒` e_1`)` e_2 ⇒ e_1`[x ←` e_2`]`

예를 들면, 우리는 표현식 `(fn x ⇒ x*x) (2+3)`을 다음과 같이 줄일 수 있다.

```
     (fn x ⇒ x*x) (2+3)
 ⇒   (fn x ⇒ x*x) (5)
 ⇒   5*5
 ⇒   25
```

이 규칙은 고차의 표현식을 값매김하는 것을 쉽게 해 준다. 여기 함수 `sq`((fn x ⇒ x*x)) 와 `double`((fn f ⇒ fn x ⇒ f (f x))) 에 대한 예제가 있다.

`double sq 5` 의 값매김은 다음과 같다.

그림 3.3: 줄이기에 대한 그래프

```
    double sq 5
=   (fn f ⇒ fn x ⇒ f (f x)) sq 5
⇒   (fn x ⇒ sq (sq x)) 5
⇒   sq (sq 5)
⇒   (fn x ⇒ x*x)((fn x ⇒ x*x) 5)
⇒   (fn x ⇒ x*x)(5*5)
⇒   (fn x ⇒ x*x) 25
⇒   25*25
⇒   625
```

3.1.2 값매김 전략

줄이기 과정이 유일하게 정의되지 않는 한, 표현식을 값매김하는 장치는 선택을 해야 한다. 줄이기의 모든 과정에서, 우리는 모든 줄이기 가능한 하위 표현식들 중 줄일 것을 골라야 한다. 우리는 이와 같은 선택을 하기 위한 모든 일관된 절차를 **값매김 전략**(evaluation strategy)이라고 부른다. 값매김 전략이 줄이기의 길이에 영향을 미치지 않던 단순 산술 표현의 값매김과는 달리, 프로그램의 값매김은 값매김 전략이 줄이기의 길이에 영향을 미치는 경우가 있다.

전략을 정의하는 데 있어서 핵심은 함수의 적용을 어떻게 다룰 것인가 하는 것이다. 이전의 예제들에서 `(fn x ⇒ `e_1`)` e_2를 값매김할 필요가 있었을 때, 우리는 함수가 e_2에 적용되기 이전에 e_2를 줄이는 것을 선택했다. 여기서 다른 방법으로 할 수도 있었다. 예를 들면, 우리는 표현식 `(fn x ⇒ x*x) (2+3)`을 다음과 같이 값매김할 수도 있다.

```
    (fn x ⇒ x*x) (2+3)
⇒   (2+3)*(2+3)
⇒   (2+3)*5
⇒   5*5
⇒   25
```

그림 3.3은 표현식 `(fn x ⇒ x*x) (2+3)` 에 대해 수행할 수 있는 모든 다시 쓰기의 그래프를 보여준다.

같은 견지에서, `double sq 5`의 또 다른 가능한 줄이기 방법은 다음과 같다.

```
    double sq 5
 =  (fn f ⇒ fn x ⇒ f (f x)) sq 5
⇒   (fn x ⇒ sq (sq x)) 5
⇒   sq (sq 5)
 =  (fn x ⇒ x*x) (sq 5)
⇒   (sq 5)*(sq 5)
 =  (sq 5)*((fn x ⇒ x*x) 5)
⇒   (Sq 5)*(5*5)
⇒   (sq 5)*25
 =  ((fn x ⇒ x*x) 5)*25
⇒   (5*5)*25
⇒   25*25
⇒   625
```

여기서 우리는 인자에 함수를 적용할 때 이를 완전히 값매김하지 않았고, 따라서 중복된 줄이기를 수행해야 했다. 그럼에도 불구하고, 우리는 항상 인자를 먼저 값매김하는 것이 옳다고 단정할 수 없는데, 그 이유는 어떤 경우 함수의 인자가 결과에 포함되지 않고 따라서 이를 줄이는 것이 쓸데없는 일일 수도 있기 때문이다.

```
    (fn x ⇒ 1) (fact 1000)
⇒   1
```

여기서 우리가 (fact 1000)자리에 줄이기가 끝나지 않는 표현식을 넣을 수도 있었다는 사실에 주의하라.

같은 선상에서, 다음과 같은 조건문을 값매김하려면

if e_1 then e_2 else e_3

우리는 e_1은 반드시 값매김해야 하지만, e_1이 true로 값매김되지 않는 이상 e_2를 값매김할 필요는 없고, 또한 e_1이 false로 값매김되지 않는 이상 e_3를 값매김할 필요가 없다. 이 구성 방법과 대응하는 다시 쓰기 규칙은 다음과 같다.

- if true then x else y ⇒ x
- if false then x else y ⇒ y

이와 같이 함수의 인자를 값매김하지 않는 전략은 어떤 함수 적용을 처리하는 데 있어서는 최고의 방법이 될 수 있다. 특히, 데이터 구조의 구성자들의 경우에는, 이 전략은 어떤 구조 전체를 값매김하지 않도록 해 준다. 예를 들면, 우리가 (e_1,e_2) 의 값묶음을 값매김하려 할 경우, 하위 표현식 e_1과 e_2를 즉시 값매김하지 않는 것은 상당히 그럴 듯한 일이다. 다시 말하면, 함수 fst, snd, 또는 패턴 매칭 등에 의해 e_1 이나 e_2를 꺼낼 일이 없을 때까지는 값매김하지 않는 것이다.(우리는 이 생각에 대해 나중에 다시 다룰 것이다.)

다시 쓰기를 이용한 값매김의 일반적인 특성은 끝나는 모든 줄이기는 같은 결과에 이른다는 것이다.[1] 이 줄이기에 있어서의 (약한) 일관성의 특성은 함수

[1]만일 우리가 케이스별로 정의된 함수들을 다룰 경우, 이는 오직 패턴 매칭이 결정적(determinisitc) 일 경우만 참이 된다.

중심적 프로그램의 특성이다. 즉, 줄이기의 순서가 바뀌더라도 그 결과는 영향을 받지 않는다. 하지만, 이 특성은, 우리가 언어에 함수 중심적이지 않은 구성 방법을 도입할 경우 더 이상 참이 아니다. 예를 들면, 할당과 같은 구성 방법이다.(이 문제에 대해서는 4장을 참조하라.)

따라서 우리는 어떤 표현식의 값을, 그 표현식에 대한 모든 가능한 줄이기가 결과적으로 도달해서 끝나는, 마지막 표현식으로 정의할 수 있다. 만일 어떤 값매김 전략이, 결과가 존재할 경우 이를 항상 같은 결과에 이르게 해 준다면 우리는 이를 **완전한**(complete) 전략이라고 부르고, 그렇지 않은 경우, 어떤 다른 줄이기 방법이 제대로 된 결과를 내 주는데도 불구하고 이 전략에 따랐을 경우는 에러가 나거나 무한한 줄이기를 하게 된다면, 이를 **불완전한**(incomplete) 전략이라고 부른다.

3.1.3 자기 참조를 다루는 방법

우리가 nML에서 구성 방법 `val rec`을 사용하여 자기 참조를 정의한 것은 완전히 만족스럽지는 않다. 사실, 평범한 함수들이 그냥 표현식으로 표기될 수 있는데도 불구하고, 자기 참조 함수들은 이름을 주지 않으면 만들 수가 없다.

이제 우리가 보게 될 값매김에서, `double` 또는 `sq`와 같은 함수들은 한번, 그리고 영원히 그것들의 "내용" 으로 대체된다. 하지만 자기 참조 함수들은 그 자체에 함수 자신의 이름을 포함하고 있기 때문에 이와 같은 방법으로 처리할 수가 없다. 따라서 우리가 이들을 우리의 값매김에 포함시키고 싶다면 우리는 자기 참조 함수들을 위한 새로운 표기법을 만들어야 한다.

가능한 하나의 해결책은 `Rec` 이라는, `(('a → 'a) → 'a)` 의 타입을 가지는 상수를 만들고, `val rec f = fn x => e`와 같이 정의된 함수를 단순히 `Rec (fn f x => e)`와 같이 표기하는 것이다.

이와 같은 함수의 값매김 규칙은 다음과 같다.

- `Rec f` ⇒ `f (Rec f)`

직관적으로, `Rec`은 `(t → t)`의 타입을 가진 고차 함수 `F`를 `Rec F`로 표시되는 타입 `t`의 고정점과 대응시키는 고정점 연산자이다. 여기서 고정점이란, `Rec F = F (Rec F)`의 특성을 갖는 값이다.

만일 우리가 nML에 새로운 상수 `Rec`을 추가하면,

```
# val rec fact = fn n => (if n = 0 then 1 else n*fact(n-1));;
```

와 같은 정의는 다음과 같이 쓰일 수 있다.

```
# val ffact = fn f n => if n = 0 then 1 else n*f(n-1);;
# val fact = Rec ffact;;
```

따라서 우리는 `fact(3)` 을 다음과 같이 줄일 수 있다.

```
    fact 3
 =  Rec ffact 3
⇒   ffact (Rec ffact) 3
 =  (fn f n => if n = 0 then 1 else n*f(n-1)) (Rec ffact) 3
⇒   (fn n => if n = 0 then 1 else n*Rec ffact (n-1)) 3
⇒   if 3=0 then 1 else 3*Rec ffact (3-1)
⇒   if false then 1 else 3*Rec ffact (3-1)
⇒   3*Rec ffact (3-1)
⇒   3*Rec ffact 2
...
⇒   3*(2*Rec ffact 1)
...
⇒   3*(2*(1*Rec ffact 0))
...
⇒   3*(2*(1*1))
⇒   3*(2*1)
⇒   3*2
⇒   6
```

이렇게 자기 참조를 다루는 방법은 λ-Calculus에서 채택되었던 방법이고, 여기서 고정점 연산자는 전통적으로 `Y`로 표기되어 왔다. 타입이 없는 λ-Calculus에서는 `Y`가 정의될 수 있고, 따라서 우리는 자기 참조 함수를 다루기 위한 특별한 구성 방법을 만들지 않고 모든 함수를 계산할 수 있다. 하지만 타입이 존재하는 λ-Calculus와 nML에서는 `Y`는 정의될 수 없고, 이는 우리가 특별한 구성 방법인 `val rec`을 만들게 된 이유가 된다.

3.1.4 자기 참조를 다루는 또 다른 방법

우리는 자기 참조를 완전히 반대의 관점에서 다룰 수 있다. 우리는 함수의 이름을 체계적으로 제거하는 것과 반대의 방법으로, 모든 함수 표현식에 체계적으로 이름을 붙이고 각 함수의 이름에 대해 정해진 규칙을 만들 수 있다.

이 방법을 쓸 경우, 평범한 함수와 재귀 함수간의 구분은 완전히 없어진다. `double`, `sq`, `fact`는 각각 다음과 같이 정의된다.

```
double f x  ⇒  f (f x)
sq x        ⇒  x*x
fact n      ⇒  if n=0 then 1 else n*fact(n-1)
```

이 방법은 매우 직관적이 된다는 장점을 가지고 있다. 또한 이 방법은 자연스럽게 함수가 케이스별로 정의되도록 확장할 수 있다. 이 경우 각 케이스는 서로 다른 규칙들에 대응한다.[2] 여기 `fact`함수를 케이스별로 정의한 것이 있다.

```
# val rec fact = fn 0 => 1 | n => n*fact(n-1);;
```

이는 두 가지 규칙을 만들어 낸다.

[2] 케이스들이 서로 겹치지 않는 범위 내에서만 그렇다.

```
fact 0  ⇒  1
fact n  ⇒  n*fact(n-1)
```

물론 첫째 규칙이 적용될 수 없을 때만 둘째 규칙이 적용된다. 즉 둘째 규칙의 적용은 (n≠0)이라는 조건에 의해 제한된다.

그렇다 하더라도, 이 직관적인 접근법은 단점이 있는데, 그것은 어떤 함수들은 자유 변수를 가지고 있기 때문에, 이렇게 다룰 경우 문제가 될 수 있기 때문이다.

예를 들면, 만일 우리가

```
# val f = fn x => (compose ((fn y => x*y),(fn y => x*(y+1))))
```

와 같이 정의된 함수를 다룰 경우, 이 정의를 단순히 다음과 같이 바꾸는 것은 불가능하다.

```
# val f1 = fn y => x*y;;
# val f2 = fn y => x*(y+1);;
# val f = fn x => compose (f1,f2);;
```

그 이유는 f1와 f2에서 나타나는 x의 정의가 제 위치를 벗어났고, 따라서 이는 말이 안되기 때문이다.

결과적으로, 우리는 x를 f1과 f2의 새로운 인자로 만들어 주어야 한다. 따라서 우리는 다음과 같은 결과를 얻는다.

```
# val f1 = fn x y => x*y;;
# val f2 = fn x y => x*(y+1);;
# val f = fn x => compose ((f1 x),(f2 x));;
```

우리는 위와 같은 조작이 항상 가능하다는 것을 증명할 수 있다. 이런 종류의 변환은 어떤 함수 중심 언어의 구현에 쓰이고 있고, λ-**리프팅**(λ-lifting) 이라고 불리워진다.

3.1.5 값매김 과정

만일 우리가 몇몇 중요하지 않는 단계들을 빼고 치환을 이용하여 팩토리얼 함수를 계산하는 것을 다시 살펴보면, 우리는 과정의 "기하학적인" 형태가 어떻게 되는지를 명확히 볼 수 있다. 표현식의 크기는 함수의 연속적인 호출이 이루어지는 동안 점점 늘어나다가, 이 호출들을 통해 만들어진 단순한 산술 표현식들이 값매김되면서 점점 줄어든다.

```
    fact(3)
⇒   3*fact(2)
⇒   3*2*fact(1)
⇒   3*2*1*fact(0)
⇒   3*2*1*1
⇒   3*2*1
⇒   3*2
⇒   6
```

함수 fact에 대한 모든 호출은 두 번째 인자가 알려지기 전까지는 수행될 수 없는 곱셈을 만들어 보류해 둔다. 이것이 표현식의 크기가 커지는 이유이다.

팩토리얼 함수를 다르게 정의해서 계산이 근본적으로 다르게 수행되도록 할 수 있다. 이 새로운 정의는 facti라는 인자 2개를 받는 함수를 추가로 정의하여 사용한다.

```
# val rec facti = fn n r => (if n=0 then r else facti (n-1) (r*n));;
val facti :  int -> int -> int = <fun>
# val fact = fn n => facti n 1;;
val fact :  int -> int = <fun>
```

fact 3의 계산은 다음과 같이 된다.

```
     fact 3
 ⇒   facti 3 1
 ⇒   facti 2 3
 ⇒   facti 1 6
 ⇒   facti 0 6
 ⇒   6
```

여기서, 각 단계에서 나타나는 표현식의 크기는 일정하고, 계산은 마지막 facti 함수의 호출이 끝나면 종료된다. 2.3절의 fold_left와 fold_right 함수에서도 같은 차이를 볼 수 있다.

물론 이는 실제로 컴퓨터 내에서 일어나는 일을 비슷하게 묘사한 것에 불과하지만, 이 두 계산 사이의 차이는 실제로 일어나는 두 계산 방법 사이의 차이를 반영한다. 일반적으로, 표현식이 다시 쓰일 때 그 길이가 늘어나는 것은 실제 계산을 수행할 때는 메모리가 쓰이는 양과 대응한다. 이 메모리는 자기 차례를 기다리는 계산들을 저장해 놓기 위해 쓰이는 스택의 형태로 소비된다. (이에 대한 자세한 것은 컴파일에 대해 다루는 12장을 참조하라.) 이렇게 메모리를 소비하는 것은 또한 계산 시간의 증가를 가져온다.

facti와 같은 함수는 **반복적**(iterative)으로 동작한다. 전문 용어로 말하면, 이와 같은 함수들은 **꼬리만 자기 참조적**(tail recursive)이다. 이들은 좀 더 효과적인 형태로 변환될 수 있다. 반대로, fact의 첫 번째 버전과 같은 함수들, 진짜로 자기 참조적인 함수들은, 그것들이 의미하는 수학적 함수들을 매우 직접적이고 자연스럽게 표현한다.

실제로는, 이 두 형태의 자기 참조가 얼마나 효과적인지는 어떤 컴파일 기술이 쓰이느냐에 크게 의존한다.

3.2 값매김 전략 정의하기

우리가 65쪽에서 정의한 완전성의 의미에 따른다면, 완전한 값매김 전략은 존재한다. 인자를 값매김하지 않고 함수의 적용을 수행하는 값매김 전략들은 기본적으로 완전하다. 그리고 이와 같은 완전한 값매김 전략에 기반하여 함수 중심 언어를 구현하는 것 역시 가능하다.

그러나, 실제로는, 불완전한 전략에 기반한 구현이 훨씬 이치에 맞다. 가장 중요한 이유는, 모든 상황에 대해 함수 중심적 프로그래밍 스타일을 적용하는 것이 어렵기 때문이다. 만일 프로그램이 외부와 상호 작용을 한다면, 그것은 상

호 작용의 상대가 되는 외부 세계의 제약을 반드시 고려해야 한다. 즉, 사건의 순차성(사건들이 일어나는 순서에 대한 제약)과 같은 것을 고려해야 한다. 예를 들면, 우리가 사용자와 대화를 하는 형식의 프로그램을 짤 때, 질문은 반드시 그에 해당하는 답변 이전에 일어나야 한다. 따라서 프로그래머는 프로그램의 여러 부분의 값매김 순서를 모두 알고 있어야 하게 된다. 완전한 값매김 전략에서는, 값매김의 순서는 값매김 장치 – 프로그래머가 아닌 – 에 의해 계산에서의 필요의 순서에 따라 결정되고, 따라서 값매김은 프로그래머의 제어를 벗어난다.

값에 의한 값매김(evaluation by value)라 불리는 값매김 전략, 다시 말하면, 함수 적용의 인자가 적용이 일어나기 전에 값매김되는 전략에서는, 프로그래머가 값매김 순서를 모두 아는 것이 가능하다. 이는 nML에서 이 전략을 선택한 이유가 된다. 따라서 nML은 어떤 응용 프로그램들을 구현하기 위해 필요한 함수 중심적이지 않은 구성 요소들을 가지도록 확장될 수 있다.

이론적인 관점에서 보면, 불완전한 전략을 선택한 것은 프로그램의 정확성(correctness)에 대한 어떤 증명도 포기한 것이 아니다. 요컨대, 끝나는 계산만 고려하게 되면 모든 불완전한 값매김 전략은 완전하게 되고, 특히 "값에 의한" 전략 역시 완전하다. 결국 값에 의한 값매김을 선택한 댓가로 우리가 치루어야 하는 것은, 프로그램의 정확성을 증명하기 위해서 프로그램이 끝난다는 것을 증명해야 하게 되었다는 사실이다.[3]

우리는 nML에서 쓰이고 있는 것부터 시작해서, 다양한 가능한 값매김 전략을 정의해 나갈 것이다. 이 전략들은 연역적인 규칙들에 의해 시스템에서 정확하게 정의된다.

우리는 이 규칙들이 실제 언어가 기계의 수준에서 구현되는 것과 단지 비슷하게 대응하기만 한다는 것을 기억해야 한다. 구현에서의 실제 문제들은 언어 해석(interpretation)에 대해 다루는 11장과 언어의 기계어 변환(compilation)에 대해 다루는 12장에서 논의할 것이다.

3.2.1 값에 의한 값매김, nML에서의 전략

우선 우리의 논의를 몇몇 직관적인 규칙에 대한 묘사에서 시작해 보자.

값매김 규칙들

- 값묶음을 값매김하기 위해서는, 각 원소들을 오른쪽에서 왼쪽으로 연속하여 값매김한다.
- `if t then` e_1 `else` e_2 표현식을 값매김하기 위해서는, `t`부터 값매김을 시작한다. 만일 그 값이 `true` 이면, 전체 표현식을 e_1으로 대체한다. 만일 그 값이 `false`이면, 전체 표현식을 e_2로 대체한다.
- 함수 표현식 `(fn x ⇒ e)` 또는 `(fn (`x_1`,...,`x_n`) ⇒ e)` 은 그것 자체가 값매김의 결과이다. 그것은 인자에 적용될 때까지 같은 모습으로 남는다.
- 함수의 적용을 값매김하기 위해서는, 우선 인자를 값매김하고 다음에 연산자를 값매김한다. 그리고 나서 연산자를 인자에 적용한다. 연산자는

[3] 우리는 지금까지는 단지 각 구성 요소에 대한 증명을 통해, 어떤 프로그램들은 끝나는 특성을 가진다는 것만을 증명할 수 있었다.

덧셈과 같은 기본 연산이 될 수 있고, 이 경우 우리는 바로 결과를 얻는다. 다른 경우, 만일 적용이 (fn x ⇒ e)(v) 의 꼴이라면, 우리는 표현식 e[x←e] 를 값매김하는 것으로 계산을 계속한다. 그리고 만일 적용이 (fn (x_1,...,x_n) ⇒ e) (v_1,...,v_n) 의 꼴이라면, 우리는 표현식 e[(x_1,...,x_n) ← (v_1,...,v_n)] 를 값매김하는 것으로 계산을 계속한다.

- let val x=e_1 in e_2 end 표현식을 값매김하기 위해서는, 우리는 먼저 e_1 을 값매김하고, 결과가 v_1 이라 하면, 다음에 e_2[x ← v_1] 을 값매김한다.[4]
- let val rec f = fn x ⇒ e_1 in e_2 end 표현식을 값매김하기 위해서는, e_2[f ← Rec(fn f x ⇒ e_1)] 을 값매김한다.

위 규칙을 완성하기 위해서는, 우리는 Rec f 형태의 적용에 대한 값매김 방법을 명시해야 한다. 만일 우리가 단순히 Rec f 를 f (Rec f) 꼴로 대체한다면, 바로 무한한 계산에 빠지게 되는데, 이유는 f (Rec f) 를 값매김하기 위해서는, 다시 한번 Rec f 를 값매김해야 하기 때문이다. 따라서 이 방법은 nML 에는 잘 맞지 않는다.

그럼에도 불구하고, 우리는 다음과 같은 새로운 규칙을 이용하여 이를 해결할 수 있다.

```
Rec f x  ⇒  f (Rec f) x
```

(Rec f) x 를 값매김하여 나온 Rec e 는 적당한 인자가 적용되었을 경우 단 한번만 값매김되고, 고유의 값을 가지게 된다. 이 방법으로, 우리는 무한한 계산을 피할 수 있다.

연역 규칙들로 정의된 값매김 규칙

값매김 규칙들은 또한 다음과 같이 표시되는 관계를 정의하는 연역 규칙들로 간주될 수 있다.

⊢ e ⇒ v

위 식의 뜻은 다음과 같다.

표현식 e는 값 v를 가진다.

여기서, 우리가 값이라고 부르는 것은 단순히 최초의 표현식을 다시 써서 얻어낸 표현식이다. 고려 대상이 된 표현식이 기본 타입이거나 또는 오직 기본 타입과 데카르트곱만을 써서 만든 타입일 경우, 그 값은 전통적인 의미에서의 값의 모습을 가진다. 그에 반하여, 함수 화살표를 내부에 가지는 표현식의 경우, 그 값은 단순히 함수 표현식을 가지고 있는 표현식이 된다. 이런 표현식들를 값으로 취급하게 되면 값으로 취급되는 표현식들이 줄일 수 있는 하위 표현식을 가질 수도 있음에 주의하자. 예를 들면, 표현식 fn x ⇒ x ∗ (2 + 3)은 내부에 (2+3)이라는 줄이기 가능한 하위 표현식을 가지는데도 불구하고 값으로 취급된다.

---

[4] 따라서 표현식 let val x=e_1 in e_2 end은 (fn x ⇒ e_2) e_1 과 완전히 동치인 것으로 취급된다.

언어의 모든 구성 방법에 대해, 우리는 다음과 같은 형식의 연역 규칙을 정의할 것이다.

$$\frac{\text{가정들의 리스트}}{\text{결론}}(\text{이름})$$

여기 그 규칙들이 있다.

$$\frac{\vdash \mathtt{e}_1 \Rightarrow \mathtt{v}_1 \ldots \vdash \mathtt{e}_n \Rightarrow \mathtt{v}_n}{\vdash (\mathtt{e}_1, \ldots, \mathtt{e}_n) \Rightarrow (\mathtt{v}_1, \ldots, \mathtt{v}_n)}(\text{값묶음})$$

$$\frac{\vdash \mathtt{e}_1 \Rightarrow \mathtt{true} \quad \vdash \mathtt{e}_2 \Rightarrow \mathtt{v}}{\vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \Rightarrow \mathtt{v}}(\text{조건1})$$

$$\frac{\vdash \mathtt{e}_1 \Rightarrow \mathtt{false} \quad \vdash \mathtt{e}_3 \Rightarrow \mathtt{v}}{\vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \Rightarrow \mathtt{v}}(\text{조건2})$$

$$\frac{}{\vdash (\mathtt{fn}\ \mathtt{x} \Rightarrow \mathtt{e}) \Rightarrow (\mathtt{fn}\ \mathtt{x} \Rightarrow \mathtt{e})}(\text{함수1})$$

$$\frac{}{\vdash (\mathtt{fn}\ (\mathtt{x}_1, \ldots, \mathtt{x}_n) \Rightarrow \mathtt{e}) \Rightarrow (\mathtt{fn}\ (\mathtt{x}_1, \ldots, \mathtt{x}_n) \Rightarrow \mathtt{e})}(\text{함수2})$$

$$\frac{\vdash \mathtt{e} \Rightarrow \mathtt{op} \quad \vdash \mathtt{e}_i \Rightarrow \mathtt{v}_i \quad \vdash \mathtt{op}\ \mathtt{v}_1 \ldots \mathtt{v}_n \Rightarrow \mathtt{v}}{\vdash \mathtt{e}\ \mathtt{e}_1 \ldots \mathtt{e}_n \Rightarrow \mathtt{v}}(\text{기본연산자 적용})$$

$$\frac{\vdash \mathtt{e}_1 \Rightarrow (\mathtt{fn}\ \mathtt{x} \Rightarrow \mathtt{e}) \quad \vdash \mathtt{e}_2 \Rightarrow \mathtt{v}_2 \quad \vdash \mathtt{e}[\mathtt{x} \leftarrow \mathtt{v}_2] \Rightarrow \mathtt{v}}{\vdash \mathtt{e}_1\ \mathtt{e}_2 \Rightarrow \mathtt{v}}(\text{적용 1})$$

$$\frac{\vdash \mathtt{e}_1 \Rightarrow (\mathtt{fn}\ (\mathtt{x}_1 \ldots \mathtt{x}_n) \Rightarrow \mathtt{e}) \quad \vdash \mathtt{e}_2 \Rightarrow (\mathtt{v}_1 \ldots \mathtt{v}_n) \quad \vdash \mathtt{e}[\mathtt{x}_i \leftarrow \mathtt{v}_i] \Rightarrow \mathtt{v}}{\vdash \mathtt{e}_1\ \mathtt{e}_2 \Rightarrow \mathtt{v}}(\text{적용 2})$$

$$\frac{\vdash \mathtt{e}_1 \Rightarrow \mathtt{v}_1 \quad \vdash \mathtt{e}_2[\mathtt{x} \leftarrow \mathtt{v}_1] \Rightarrow \mathtt{v}}{\vdash \mathtt{let}\ \mathtt{val}\ \mathtt{x}\ =\ \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2\ \mathtt{end} \Rightarrow \mathtt{v}}(\text{Let 1})$$

$$\frac{\vdash \mathtt{f}\ (\mathtt{Rec}\ \mathtt{f})\ \mathtt{x} \Rightarrow \mathtt{v}}{\vdash \mathtt{Rec}\ \mathtt{f}\ \mathtt{x} \Rightarrow \mathtt{v}}(\text{Rec})$$

$$\frac{\vdash \mathtt{e}_2[\mathtt{f} \leftarrow \mathtt{Rec}(\mathtt{fn}\ \mathtt{f}\ \mathtt{x}\ \Rightarrow\ \mathtt{e}_1)] \Rightarrow \mathtt{v}}{\vdash \mathtt{let}\ \mathtt{val}\ \mathtt{rec}\ \mathtt{f}\ =\ \mathtt{fn}\ \mathtt{x} \Rightarrow \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2\ \mathtt{end} \Rightarrow \mathtt{v}}(\text{Letrec})$$

3.2.2 또 다른 전략 : 미루어 값매김하기

만일 우리가 nML의 전략에서 시작하여 완전한 전략을 얻고 싶다면, 값묶음과 함수 적용, Rec에 대한 몇 가지 규칙들만 고치면 된다.

값묶음에 대한 새로운 규칙은 다음과 같다.

$$\frac{}{\vdash (\mathtt{e}_1, \ldots, \mathtt{e}_n) \Rightarrow (\mathtt{e}_1, \ldots, \mathtt{e}_n)} (\text{값묶음})$$

함수 적용에 대한 규칙은 그것들이 인자를 값매김하지 않도록 고쳐야 한다. 그럼에도 불구하고, 하나 이상의 변수를 가진 함수, 즉 $(\mathtt{fn}\ (\mathtt{x}_1, \ldots, \mathtt{x}_n) \Rightarrow \mathtt{e})$ 의 경우, 여기에 인자가 적용되기 위해서 인자는 반드시 값묶음이어야 하므로, 우리는 다음과 같이 규칙을 만들어야 한다.

$$\frac{\vdash \mathtt{e}_1 \Rightarrow \mathtt{fn}\ \mathtt{x} \Rightarrow \mathtt{e} \quad \vdash \mathtt{e}[\mathtt{x} \leftarrow \mathtt{e}_2] \Rightarrow \mathtt{v}}{\vdash \mathtt{e}_1\ \mathtt{e}_2 \Rightarrow \mathtt{v}} (\text{적용 1})$$

$$\frac{\vdash \mathtt{e}_1 \Rightarrow \mathtt{fn}\ (\mathtt{x}_1, \ldots, \mathtt{x}_n) \Rightarrow \mathtt{e} \quad \vdash \mathtt{e}_2 \Rightarrow (\mathtt{v}_1, \ldots, \mathtt{v}_n) \quad \vdash \mathtt{e}[\mathtt{x}_i \leftarrow \mathtt{v}_i] \Rightarrow \mathtt{v}}{\vdash \mathtt{e}_1\ \mathtt{e}_2 \Rightarrow \mathtt{v}} (\text{적용 2})$$

마지막으로, 구성 방법 Rec에 대한 규칙은 다음과 같이 처음의 모양을 사용하면 된다.

$$\frac{}{\vdash \mathtt{Rec\ f} \Rightarrow \mathtt{f}\ (\mathtt{Rec\ f})} (\text{Rec})$$

우리는 이 과정을 미루어진 값매김(delayed evaluation)이라고 부른다.

미루어진 값매김을 이용하여 얻은 값들은 값에 의한 값매김을 이용하여 얻어낸 값들과 같을 필요는 없다. 예를 들면, 미루어진 값매김에서는 값묶음의 인자들이 값매김되지 않기 때문에, (2+3, 7)은 값이다. 하지만 값메 의한 값매김에서는 (5, 7)이라는 값을 얻기 위해 계산을 수행해야 한다. 그러나, 간단한 타입에 속하는 표현식들의 경우, 값에 의한 값매김과 미루어진 값매김으로 얻는 결과는 같다. 즉, 해당하는 타입에 속하는 상수를 값으로 얻을 수 있다.

우리가 이와 같은 방법으로 정의한 젼략을 이해하기 위해서, 다음과 같은 표현식을 고려해 보자.

$$(\mathtt{fn}\ (\mathtt{x}, \mathtt{y}) \Rightarrow \mathtt{x})\ ((\mathtt{fn}\ \mathtt{x} \Rightarrow (\mathtt{x}, \mathtt{fact}\ 1000))\ 3)$$

미루어진 값매김으로 위 표현식의 값을 찾아내기 위해서는, 표현식이 인자가 값묶음인 함수를 가진 함수 적용이므로, 우리는 (적용 2) 에 관한 규칙을 사용해야 한다. 이 규칙을 적용하기 위하서는, 인자의 값묶음 구조가 겉으로 드러나야 하고, 따라서 우리는 반드시 인자, 즉 $((\mathtt{fn}\ \mathtt{x} \Rightarrow (\mathtt{x}, \mathtt{fact}\ 1000))\ 3)$ 를 값매김해야만 한다.

이를 값매김하기 위해서는, 간단한 함수 적용에 대한 규칙인 (적용 1) 을 사용하면 된다. 적용 결과는 (3, fact 1000) 이 된다.

이제 이 값묶음에 함수 $\mathtt{fn}\ (\mathtt{x}, \mathtt{y}) \Rightarrow \mathtt{x}$ 를 적용하면, 우리는 fact 1000 을 값매김할 필요 없이 바로 결과를 얻을 수 있다.

따라서 미루어진 값매김에 의해 위 표현식을 계산하는 과정은 다음과 같다.

```
    (fn (x,y) ⇒ x) ((fn x ⇒ (x,fact 1000)) 3)
⇒   (fn (x,y) ⇒ x) (3, fact 1000)
⇒   3
```

3.2.3 미루어진 값매김을 자료 구조에까지 확장하기

우리는 미루어진 값매김에서 값묶음에 대해 사용했던 방법을 다른 어떤 자료 구조나 구성 방법에도 적용할 수 있다. 예를 들면, 리스트의 구성 방법인 Cons(::)를 사용할 때도, 그 인자를 값매김할 필요는 없다.

패턴 매칭의 경우, 우리의 적용 규칙이 너무 복잡해지지 않도록, 제한된 형태의 패턴 매칭에 대해서만 생각해 보자. 자료 구성자 C_1, C_2, ..., C_n 를 포함하는 타입이 있다고 하면, 이 타입에 대해 케이스별로 정의된 함수는 항상 $\mathtt{fn}\ \mathtt{p_1} \Rightarrow \mathtt{e_1}\ |\ \ldots\ |\ \mathtt{p_n} \Rightarrow \mathtt{e_n}$ 의 형태를 가지고, p_i 는 $C_i(x_1, \ldots, x_{n_i})$ 의 형태이며, n_i 는 자료 구성자 C_i 의 인자의 갯수를 나타낸다.

예를 들면, 다음과 같은 리스트의 정의에서는

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

Nil 과 Cons(x_1, x_2) 형태의 패턴을 사용하는 것만이 허용된다.

자료 구성자들에 대한 새로운 규칙은 다음과 같다.

$$\frac{}{\vdash \mathtt{C(e_1,\ldots,e_n)} \Rightarrow \mathtt{C(e_1,\ldots,e_n)}}(\text{구성자})$$

$$\frac{\begin{array}{c}\vdash \mathtt{e} \Rightarrow (\mathtt{fn}\ \ldots\ |\ \mathtt{C_i(x_1,\ldots,x_{n_i})} \Rightarrow \mathtt{e_i}\ |\ \ldots) \\ \vdash \mathtt{e'} \Rightarrow \mathtt{C(e_1,\ldots,e_n)} \quad \vdash \mathtt{e_i[x_j \leftarrow e_j]} \Rightarrow \mathtt{v}\end{array}}{\vdash \mathtt{e\ e'} \Rightarrow \mathtt{v}}(\text{케이스 적용})$$

이 규칙들은 어떤 프로그램들의 복잡도에 중요한 영향을 끼칠 수 있다. 또한, 이 규칙들은 프로그래머에게 새로운 가능성을 제공한다.

예를 들면, 수열이라는 수학적 개념은 다음과 같은 간단한 타입을 가지는 무한 리스트로 직접 표현될 수 있다.

```
# type 'a seq = Cons of 'a * 'a seq;;
```

우리는 Cons를 리스트 타입에 대한 중위 연산자 :: 로 표기하고, 따라서 두 수열의 합은 다음과 같이 간단히 쓸 수 있다.

```
# val rec sum =
      fn (a1::l1, a2::l2) => (a1+a2)::sum(l1,l2);;
```

따라서 피보나치 수열은 다음과 같이 간단히 표현된다.

```
# val rec fibs = 1::fibs1
      and fibs1 = 1::sum(fibs, fibs1);;
```

이 정의들은 그것들 자체만으로는 아무런 계산을 수행하지 않는다. 반면, 이 수열에서 원소 하나를 얻어내는 일은 이 원소를 구하는 계산을 시작하게 하고, 이 원소를 계산하려면, 우리는 이 원소 앞의 모든 원소를 계산해야 한다. 원소 하나를 얻어내려면 우리는 nth라는 이름의 함수를 사용한다.

```
# val rec nth =
      fn n (a::l) => if n=0 then a else (n-1) l ;;
```

여기 4번째 피보나치 수를 얻어내는 계산이 있다.

3.3 프로그램의 의미 구조

프로그램의 의미 구조를 이해하기 위해서, 우리는 우선 다시 쓰기를 이용한 의미 구조(rewrite semantics)에 대해 살펴보고, 또한 수학적 표기법을 이용한 의미 구조(denotational semantics)에 대해서도 간단히 알아야 한다. 이를 알아보는 과정에서, 여러분은 왜 우리가 수학적 표기법을 이용한 의미 구조를 사용하는 대신 다시 쓰기를 이용한 의미 구조를 사용했는지를 알 수 있을 것이다.

3.3.1 다시 쓰기를 이용한 의미 구조

우리는 연속적인 과정에 의해서 표현식을 값으로 변환하는 다시 쓰기 규칙들을 정의했다. 이 경우 값은 단순히 더 이상 다시 쓸 수 없는 표현식이다. 우리는 이런 표현식들을 현재 고려중인 다시 쓰기 규칙에 관해 **정규 형태**(normal form) 에 있다고 말한다. 표현식이 단순한 타입(정수, 실수, 논리값 등) 을 가질 경우, 정규 형태는 일반적으로 이야기하는 표현식의 값과 같은 형태를 가진다.

모든 끝나는 계산은 같은 값을 결과로 가지기 때문에, 다시 쓰기 규칙은 표현식의 값을 모호하지 않게 정의할 수 있게 해 준다. Val(E)로 표기되는 표현식 E의 값은, 끝나는 계산이 존재할 경우, E의 정규 형태, 즉 모든 끝나는 계산이 도달하는 공통의 표현식으로 정의된다. 그런 계산이 존재하지 않을 경우, E의 정규 형태는 정의되지 않는다. 그러면 함수 Val은 언어의 의미 구조를 정의하는 것이 된다. 즉, Val은 언어의 모든 표현식에 대해 값을 연결해 주는 역할(비록 정의되지 않은 값이라 할지라도)을 한다.

한편 우리는 다시 쓰기가 이루어지는 순서에 제한을 두어서 값매김 전략을 정의하는 방법에 대해 살펴보았다. 주어진 값매김 전략 S에 대해, S의 규칙에 따른 계산 절차를 따라서 표현식 E를 값매김하여 얻어진 값 E'는, 만일 끝나는 계산이라면 Val(E') = Val(E) 를 만족한다. 게다가, 만일 E가 간단한 타입이라면, E' = Val(E) 가 성립한다.

우리는 값매김 전략 S와 연관된 값매김 함수를 Val_S 라고 표기할 것이다. 만일 E가 간단한 타입이고, Val_S(E)가 정의된다면(즉, 끝나는 계산이라면), 우리는 Val(E) 역시 정의되고 Val_S(E) = Val(E) 가 성립함을 확신할 수 있다. 만일 E가 간단한 타입이 아니라면, 우리는 단지 Val(Val_S(E)) = Val(E) 임을 알 수 있다.

3.3.2 수학 표기를 이용한 의미 구조

이 언어가 함수 중심적이라는 사실은 수학 표기를 이용한 의미 구조를 살펴볼 가치가 있게 만든다. 구체적으로 만일 우리가 이 언어가 다루는 모든 값들의 집

합을 D라 정의한다면(즉, D라는 수학적 기호로 표기한다면), 함수들 역시 값이므로, D 는 함수 공간 D → D 을 포함해야만 하며, 이는 D → D 가 항상 D 보다 많은 원소를 가지고 있기 때문에 보통의 수학에서는 비현실적이다. 따라서 우리는 전체 함수 공간보다 작도록 D → D 를 잘 해석해야 한다. 우리는 이를 D 에 대한 위상을 정의하고 D → D 를 D 위에서의 연속함수들의 공간으로 해석함으로써 이를 해결할 수 있다. 이와 같이, 수학 표기를 이용한 의미 구조는 매우 흥미로운 주제이지만, 이 책에서 다루기는 너무 복잡한 주제이다. 이 주제에 대한 좋은 입문 자료를 위해서는 [3] 를 참조하라.

다시 쓰기를 이용하여 정의한 의미 구조는 직관적으로 이해하기 쉽다는 장점을 가지고 있다. 또한 그것은 다음 절에서 소개할 프로그램의 정확성 증명을 정당화하는 기교로도 사용된다.

3.4 프로그램의 정확성 증명하기

함수 중심적 프로그램의 특성에 대한 증명들은 동치만들기 논법(equational reasoning), 즉 어떤 표현식을 같은 값을 가지는 다른 표현식으로 대체하는 방법과, 자료 구조에 대한 수학적 귀납법의 적용으로 이루어진다.

3.4.1 동치만들기 논법

프로그램을 값매김하기 위한 다시 쓰기 규칙들은 또한 등식들이나 동치 관계로 해석될 수도 있다. 우리가 다시 쓰기 규칙 하나를 적용하여 표현식 E로부터 E'를 얻었다고 가정하자. 이는 E ⇒ E'로 표현된다. 만일 E'가 잘 정의된 값이라면, E 역시 이과 똑같은 값을 가진다. 그 이유는 우리는 E ⇒ E' 라는 사실을 이용하여 E의 값을 계산하는 과정을 항상 E'의 값을 계산하는 과정과 연관시킬 수 있기 때문이다. 역으로, 만일 E가 잘 정의된 값을 가진다면, E'역시 수렴성에 의해서 E와 똑같은 값을 가진다.

요약하면, E ⇒ E' 라는 사실은 Val(E) = Val(E')라는 사실을 의미한다. 따라서 우리는 다시 쓰기 규칙을 표현식들간의 등식이나 동치 관계로 사용할 수 있다.

덧붙여서, 다음을 증명하는 것이 가능하다.

> Val(E) = Val(E') 는 Val(E[x ← E"]) = Val(E'[x ← E"]) 를 의미한다.

결과적으로, 동치만들기 논법은 치환에 대해서도 유효한 동치 관계를 성립시킨다.

예를 들어, 다음과 같은 함수들에 대해 생각해 보자.

```
# val double = fn f x => f (f x);;
# val compose = fn f g x => f (g x);;
```

그러면 동치만들기 논법은 모든 함수 h와 값들 x에 대해

```
double h x = h (h x) = compose h h x
```

가 성립함을 알 수 있게 해 준다.

따라서 우리는 표현식 `double h x` 와 `compose h h x` 가 모든 h와 x에 대해 같은 값을 가진다는 사실을 증명했다.

그러나 만일 특정한 값매김 전략 S에 대해서만 생각한다면, 위와 같은 동치 만들기 논법은 값매김 전략 S만을 사용했을 때도 위의 두 표현식이 같다는 것을 증명해 주지는 않는다. 예를 들어, 다음과 같은 자기 참조 함수에 대해 생각해 보자.

```
# val f = fn x => f (f x);;
```

이 경우 두 표현식 (`fn x ⇒ 0`) (`f 1`) 과 0 은 같지만, 첫번째 표현식은 nML에서 사용하는 전략만을 사용할 경우 값을 가지지 않는다.

불완전한 전략 S만을 사용할 경우 E = E' 를 해석하는 유일한 방법은, 만일 Val_S(E)와 Val_S(E') 가 둘 다 정의된다면, Val(Val_S(E)) = Val(Val_S(E')) 가 된다는 것이다. 만일, 이 표현식들이 기본 타입에 속한다면, Val_S(E) 와 Val_S(E') 는 동일하다.

사실, 우리가 정확성을 증명하고 싶은 대부분의 프로그램들은 무한한 계산을 포함하지 않는다. 이 경우, 어떤 값매김 전략 S에 대해서도, Val_S는 모든 표현식에 대해 정의되고, 따라서 동치만들기 논법은 명확히 유효하다. 하지만 이를 사용하기 위해서는, 우리는 우리의 프로그램이 무한한 계산을 포함하지 않는다는 사실을 증명해야 한다. 즉, 우리는 계산이 끝남을 증명해야만 한다.

3.4.2 함수를 값으로 취급하는 것을 고려하기

두 함수 표현식이 같은지의 여부를 증명하는 것은 대부분의 경우 불가능하다. 예를 들면, 두 표현식 `fn x ⇒ x` 와 `fn y ⇒ y` 는 문법적으로 다르기 때문에 이들이 서로 같다고 직접 말하기는 어렵다.

이런 문제를 해결하기 위해서, 우리는 만일 두 함수가 같은 인자에 적용되었을 때 같은 결과를 낸다면, 두 함수가 같다고 생각한다. 즉,

$$(\forall x, \mathtt{f}(x) = \mathtt{g}(x)) \Longrightarrow \mathtt{f} = \mathtt{g}$$

이다. 만일 f와 g가 함수 표현식이라면, f= g임을 보이기 위해서는, f와 g내에 나타나지 않는 변수 z에 대해 fz= gz임을 보이면 된다.

예를 들어, `fn x ⇒ x = fn y ⇒ y` 임을 보이기 위해서는, 모든 z에 대해 (`fn x ⇒ x`) z= (`fn y ⇒ y`) z임을 보이는 것으로 충분하다.

같은 이유로, 우리가 이전 절에서 보였던 것을 이용하면, 우리는 모든 함수 h에 대해 다음이 성립함을 알 수 있다.

`double h` = `compose h h`

그러나, 우리는 fz= gz로부터 f= g를 이끌어 내기 위해서는 변수 z가 f와 g내에 나타나지 않아야 한다는 사실에 주의해야 한다.

`double h = compose h h` 로부터 `double = compose h` 를 이끌어 낼 수 있는 방법은 없다.

3.4.3 케이스별로 생각하기

어떤 변수의 타입이 정해져서 그 변수가 단지 유한한 갯수의 값만을 가질 수 있다고 하면, 우리는 그 변수의 값에 대해 케이스별로 생각하여 증명하는 방법을 사용할 수 있다.

예를 들면, 우리가 다음과 같은 등식을 증명하고 싶다고 하자.

(`if c then f else g`) `x` = `if c then f x else g x`

`c`가 다만 `true`와 tt false만을 가질 수 있기 때문에, 우리는 다음 두 개의 식만 증명하면 된다.

(`if true then f else g`) `x` = `if true then f x else g x`
(`if false then f else g`) `x` = `if false then f x else g x`

3.4.4 수학적 귀납법 이용하기

우리가 사용할 귀납법은 정수에 대한 귀납법의 확장형이다. 우리는 자료 구조들을 집합으로 생각하여 여기에 귀납법을 적용할 것이다.

(음 아닌) 정수에 대해서는, $\forall n\ P(n)$ 이라는 특성을 증명하기 위해서 우리는 다음 두 가지를 증명했다.

- $P(0)$
- $\forall n\ (P(n) \Longrightarrow P(n+1))$

만일 우리가 정수를, Z(0을 나타냄)와 S(주어진 정수의 다음 정수를 돌려주는 함수) 만으로 유일하게 나타낸다면, 귀납 원리는 다음과 같이 나타낼 수 있다.

- $P(Z)$
- $\forall n\ (P(n) \Longrightarrow P(S(n)))$

이 표기법에서, Z와 S는 다음과 같이 정의되는 정수 타입의 자료 구성자로 생각할 수도 있다.

```
# type nat = Z | S of nat;;
```

비슷하게, 다음과 같은 타입에 대해서도 생각해 보자.

```
# type btree = Empty | Bin of btree * btree;;
```

$\forall t\ P(t)$ 형태의 특성을 증명하기 위해서는, 우리는 다음을 증명해야 한다.

- $P(\texttt{Empty})$
- $\forall t_1, t_2\ P(t_1)$ `and` $P(t_2) \Longrightarrow P(\texttt{Bin}(t_1, t_2))$

이 증명은 `btree` 타입의 물체의 크기에 대한 귀납법 적용으로 생각할 수도 있다. 하지만 이 방법은 물체의 크기에 대한 직접 참조 없이, 각 자료 구성자에 대해 단 한 가지 경우만 따지면 되기 때문에 더 우아하다고 할 수 있다. 이 방법은 자료 구성자를 이용하여 정의된 자료 구조들의 집합 모두에 대해 적용 가능하다.

3.4.5 계산이 끝남을 증명하기

어떤 함수가 모든 영역에서 정의되었는지 그렇지 않은지는 **결정할 수 없는**(undecidable) 특성이다. 즉, 어떤 함수의 정의를 받아서 그것이 모든 입력 데이터에 대해 정의되어 있는지를 알려 주는 알고리즘은 존재하지 않는다. 또한, 어떤 함수의 정의와 특정한 값 하나를 받아서 주어진 값이 주어진 함수에 속하는지를 알려 주는 알고리즘 역시 존재하지 않는다.

이 비결정성의 문제는 계산 가능한 함수들의 집합을 정의하는 어떤 정형화된 시스템 – 튜링 머신, λ-Calculus, 그리고 여타 프로그래밍 언어들 – 에서도 나타난다. 결과적으로 어떤 함수가 끝나는지를 증명하는 것은 무한히 복잡할 수 있다.

그러나, 이와 같은 증명이 꽤나 간단하게 끝나는 많은 상황들이 존재한다. 함수 `f`가 기본적인 자료 구성자들과, 우리가 이미 모든 곳에서 정의되어 있음을 알고 있는 기본적인 함수들만을 사용해서 만들어진 자기 참조 함수라고 하자. 이 `f`가 모든 곳에서 정의되어 있음을 증명하기 위해서는, `f`의 내용에서 나타나는 `f`의 자기 참조 호출이 가지는 인자가 `f`가 가지는 인자보다 더 '간단한' 형태이고, 기본적인 케이스들이 잘 정의되면 된다.

예를 들어, 다음과 같은 팩토리얼 함수에 대해 생각해 보자.

```
fact(0) = 1
fact(n) = n*fact(n-1)
```

우리는 즉시 함수 `fact`가 모든 양의 정수에 대해 정의되어 있음을 귀납법으로 보일 수 있다.

- `fact(0)`은 정의되어 있고 그 값은 1이다.
- 모든 정수 `n`에 대해, 만일 `fact(n)`이 정의되었다고 가정하면 `fact(n+1)` 또한 정의되고, 그 값은 `(n+1)*fact(n)`이다.

자료 구성자를 사용하는 타입에 대해 정의되고 자기 참조 호출이 함수의 초기 인자의 하위 구조에 대해 이루어지는 모든 함수에 대해 같은 방법을 적용할 수 있다. 예를 들어 다음과 같은 함수에 대해 생각해 보자.

```
# val rec mir = fn
      Empty => Empty
      (Bin(a1,a2)) => Bin(mir a2, mir a1);;
```

우리는 `btree` 타입의 구조에 대한 귀납법을 이용하여 바로 함수 `mir` 가 모든 곳에서 정의되어 있음을 알 수 있다.

- `mir(Empty)` 는 잘 정의되어 있고 그 값은 Empty이다.
- 임의의 트리 두 개의 묶음 (t_1,t_2)에 대해, 만일 `mir`(t_1)과 `mir`(t_2)가 정의되면 `mir(Bin(`t_1`,`t_2`))` 의 값 역시 정의된다.

이런 종류의 정의는 기본 자기참조 정의(recursive primitive definition)들에 속한다. 그것들이 끝나는지의 여부는 그것들의 구조로부터 명백히 알 수 있다.

다른 상황의 경우, 우리는 함수의 인자들 중 하나의 감소에 촛점을 맞추어서 함수 호출이 끝남을 증명할 수도 있다. 이 때 해당 인자의 감소는 무한히 감소하는 수열을 포함하지 않는 순서 관계(ordered relation)로 특징지워질 수 있어야 한다. 그와 같은 순서 관계는 올바르게 기초하였다(well founded)고 불리워진다.

예를 들면, 다음과 같은 m^n을 계산하는 프로그램에서, 지수는 매 호출마다 반으로 나뉘어지기 때문에 정수 내에서 단조감소한다.

```
# val rec exp = fn (m,n) => case n of
      0 => 1
      |n => exp(m*m,n/2) * (if (n mod 2) = 1 then m else 1);;
```

위 사실과, n=0 일 때 함수값이 1로 정의되었다는 사실을 결합하여, 위 함수가 모든 양수 n에 대해 정의되었음을 증명할 수 있다.

끝남에 대한 특성을 증명할 때 유용한 올바르게 기초한 순서 관계들 중에서, 우리는 데카르트곱에 대해 정의된 사전적 순서 관계(lexicographic order)에 대해 주의를 기울일 필요가 있다. 두 집합 E_1과 E_2에 대해 각각 정의된 순서 관계 $<_1$, $<_2$에 대해서, $E_1 \times E_2$ 에 대해 정의되는 사전적 순서 관계 $<$는 다음과 같이 $<_1$과 $<_2$로 나타내어진다.

$$(\text{x,y}) < (\text{x',y'}) \Longleftrightarrow \text{x} <_1 \text{x' or } (\text{x} = \text{x' and y} <_2 \text{y'})$$

$<_1$과 $<_2$가 잘 기초하였을 경우 $<$ 역시 잘 기초한 순서 관계가 된다는 사실을 보이는 것은 쉽다.

우리는 Ackerman의 함수라고 알려진 다음 함수가 끝남을 증명하기 위해 자연수의 묶음에 이 사전적 순서를 사용할 수 있다.

```
# val rec ack = fn
      (0,n) => n+1
      (m,0) => ack(m-1,1)
      (m,n) => ack(m-1, ack(m,n-1));;
```

증명을 위한 주 개념은 다음과 같다. 집합 E에 대해 정의된 잘 기초한 관계 연산 $<$에 대해, $\forall x\ P(x)$ 가 성립함을 보이려면, 다음을 보이는 것으로 충분하다.

$$(\forall y\ <\ x\ P(y)) \Longrightarrow P(x)$$

결과적으로, ack(m,n)이 자연수 묶음 (m,n)에 대해 끝남을 증명하려면, (m,n)보다 작은 모든 자연수 묶음 (m',n')에 대해 ack(m',n')가 성립한다고 가정한 후, ack(m,n)을 증명하면 된다. m과 n이 0이 아닌 경우, 우리는 (m,n-1)이 (m,n)보다 작다는 사실로부터 ack(m,n-1)이 끝난다고 결론내릴 수 있다. 만일 우리가 이의 값을 p라고 부른다면, (m-1,p)는 분명히 (m,n)보다 작기 때문에, ack(m-1,p)역시 끝난다고 할 수 있다. 또한 ack의 정의로부터 ack(m,n) = ack(m-1,p)이기 때문에, 우리는 결과적으로 ack(m,n) 이 끝남을 알 수 있다. 따라서 우리는 함수 ack가 자연수 영역 내에서 전체 함수임을 증명하였다.

이제부터는 프로그램의 특성을 증명하는 것에 대해 살펴보도록 하자.

3.4.6 첫 번째 예 : 트리의 뒤집은 모양 구하기

우선 다음과 같은 nML 선언을 했다고 가정하자.

```
# type btree = Empty | Bin of btree * btree;;
# val rec mir = fn
      Empty => Empty
      |(Bin(a1,a2)) => Bin(mir a2, mir a1);;
```

다음의 두 등식이 위 프로그램과 대응한다.

- mir(Empty) = Empty
- mir(Bin(a_1,a_2)) = Bin(mir(a_2),mir(a_1))

위 등식들은 차후 증명에서 쓰일 것이다.

$\forall a\ P(a), P(a) \equiv (\texttt{mir}(\texttt{mir}(a)) = a)$ 인 특성에 대해 생각해 보자.
위 특성이 성립함을 증명하기 위해서는, 다음을 보이는 것으로 충분하다.

1. P(Empty)
2. $\forall a_1, a_2 \;\; P(a_1)\&P(a_2) \Longrightarrow P(\texttt{Bin}(a_1, a_2))$

P(Empty)의 증명:

$\texttt{mir}(\texttt{mir}(\texttt{Empty})) = \texttt{mir}(\texttt{Empty}) = \texttt{Empty}$

$\forall a_1, a_2 \;\; P(a_1)\&P(a_2) \Longrightarrow P(\texttt{Bin}(a_1, a_2))$ 의 증명:

$$
\begin{aligned}
&\texttt{mir}(\texttt{mir}(\texttt{Bin}(\texttt{t}_1, \texttt{t}_2))) \\
&= \texttt{mir}(\texttt{Bin}(\texttt{mir}(\texttt{a}_2), \texttt{mir}(\texttt{a}_1))) && \text{(mir의 정의)} \\
&= \texttt{Bin}(\texttt{mir}(\texttt{mir}(\texttt{a}_1)), \texttt{mir}(\texttt{mir}(\texttt{a}_2))) && \text{(mir의 정의)} \\
&= \texttt{Bin}(\texttt{a}_1, \texttt{a}_2) && \text{(자기 참조 함수에 대한 가정)}
\end{aligned}
$$

타입의 형태가 인자를 추가하기 위해 바뀌더라도, 위 증명은 본질적으로 변하지 않음에 주의하라.

```
# type ('a,'b) btree = Leaf of 'a | Node of 'b * btree * btree;;
# val rec mir = fn
      Leaf a => a
      |(Node(b,a1,a2)) => Node(b,mir a2,mir a1);;
```

증명해야 할 식은 단순히 다음과 같이 된다.

$\forall x\ P(\texttt{Leaf}(x))\ \&\ \forall a_1, a_2\ (P(a_1)\&P(a_2) \Longrightarrow \forall y\ P(\texttt{Node}(y, a_1, a_2)))$

3.4.7 두 번째 예 : append의 결합법칙 증명

```
# val rec append = fn
      ([],l) => l
      |(a::l,l') => a::append(l,l');;
```

이제 우리는 함수 append의 결합법칙을 증명하고자 한다. 즉, 우리는 다음과 같은 특성을 증명하려 한다.

$$\forall l_1, l_2, l_3 \; P(l_1, l_2, l_3)$$
$$P(l_1, l_2, l_3) \equiv \texttt{append}(l_1, \texttt{append}(l_2, l_3)) = \texttt{append}(\texttt{append}(l_1, l_2), l_3)$$

위를 증명하기 위해서, 우리는 다음을 보여야 한다.

$$\forall l_2, l_3 \; (P([], l_2, l_3) \,\&\, \forall l (P(l, l_2, lthr) \Longrightarrow \forall \; P(a :: l, l_2, l_3)))$$

P([],l_2,l_3)의 증명:

$$\begin{array}{lll} & \texttt{append}([], \texttt{append}(l_2, l_3)) & \\ = & \texttt{append}(l_2, l_3) & (\texttt{append}\text{의 정의}) \\ = & \texttt{append}(\texttt{append}([], l_2), l_3) & (\texttt{append}\text{의 정의}) \end{array}$$

$\forall l (P(l, l_2, lthr) \Longrightarrow \forall \; P(a :: l, l_2, l_3)))$ 의 증명:

$$\begin{array}{lll} & \texttt{append}(a :: l, \texttt{append}(l_2, l_3)) & \\ = & a :: \texttt{append}(l, \texttt{append}(l_2, l_3)) & (\texttt{append}\text{의 정의}) \\ = & a :: \texttt{append}(append(l, l_2), l_3) & (\text{가정에 의함}) \\ = & \texttt{append}(a :: \texttt{append}(l, l_2), l_3) & (\texttt{append}\text{의 정의}) \\ = & \texttt{append}(\texttt{append}(a :: l, l_2), l_3) & (\texttt{append}\text{의 정의}) \end{array}$$

3.4.8 예제 : 일반화 사용하기

다음 예제는, 어떤 경우에는 우리가 원하는 것보다 더 일반적인 특성을 증명해야 할 필요가 있다는 사실을 보여준다.

다음의 두 함수에 대해 살펴보자.

```
# val rec fact = fn
      0 => 1
      n => n*fact(n-1);;
# val rec facti = fn
      (0,r) => r
      (n,r) => facti(n-1,n*r);;
```

우리는 $\forall n$ fact(n) = facti(n,1) 라는 특성을 보이고 싶다. 만일 우리가 이를 정수 n에 대한 귀납법을 이용하여 직접 증명하려 하면, 다음을 얻는다.

- fact(0) = 1 = facti(0,1)
- fact(n+1) = (n+1)*fact(n) = (n+1)*facti(n,1) = ?

사실상, 위를 증명하기 위해서는 더 일반적인 다음 특성을 증명해야 한다.

$\forall n\ \forall r\ r{*}\texttt{fact}(n) = \texttt{facti}(n, r)$

증명:

- `r*fact(0) = r*1 = r = facti(0,r)`
- `r*fact(n+1) = (r*(n+1))*fact(n) = facti(n,r*(n+1)) = facti(n+1,r)`

3.4.9 예제 : 보조 정리 사용하기

다음 예제는, 어떤 경우에는 보조 정리를 사용할 필요가 있다는 것을 보여준다.
다음 함수에 대해 생각해 보자.

```
# val rec rev = fn
      [] => []
      |(a::l) => append(rev l,[a]);;
```

우리는 $\forall l\ \texttt{rev}(\texttt{rev}(l)) = l$ 임을 증명하려 한다. 빈 리스트의 경우 $\texttt{rev}(\texttt{rev}([])=[]$ 은 명백하다. 반면,

$\forall l\ (\texttt{rev}(\texttt{rev}(l)) = l \Longrightarrow \forall a\ \texttt{rev}(\texttt{rev}(a :: l)) = a :: l)$

를 직접 증명하려 하면, 다음을 얻는다. 우리는 다음 식이 a::l과 같음을 증명해야 한다.

$\texttt{rev}(\texttt{rev}(a :: l)) = \texttt{rev}(\texttt{append}(l, [a])) = ?$

결과적으로, 여기서 증명을 더 진행하기 위해서는, 우선 두 함수 rev와 append를 결합하는 특성 하나를 찾아 내어서 이를 증명해야 한다. 즉, 다음과 같은 특성을 증명하면 된다.

$$\forall l_1, l_2\ \ Q(l_1, l_2)$$
$$Q(l_1, l_2) \equiv \texttt{rev}(\texttt{append}(l_1, l_2)) = \texttt{append}(\texttt{rev}(l_2), \texttt{rev}(l_1))$$

이를 위해서 우리는 다음 두 특성을 연속하여 증명한다.

$\texttt{rev}(\texttt{append}([], l_2)) = \texttt{append}(\texttt{rev}(l_2), \texttt{rev}([]))$

그리고

$$(\texttt{rev}(\texttt{append}(l, l_2)) = \texttt{append}(\texttt{rev}(l_2), \texttt{rev}(l))) \Longrightarrow \forall a\ (\texttt{rev}(\texttt{append}(a :: l, l_2)) = \texttt{append}(\texttt{rev}(l_2), \texttt{rev}(a :: l)))$$

$$\begin{aligned} &\texttt{rev}(\texttt{append}([], l_2)) \\ &\quad = \quad \texttt{rev}(l_2) && (\texttt{append}\text{의 정의}) \\ &\quad = \quad \texttt{append}([], \texttt{rev}(l_2)) && (\texttt{append}\text{의 정의}) \\ &\quad = \quad \texttt{append}(\texttt{rev}([]), \texttt{rev}(l_2)) && (\texttt{rev}\text{의 정의}) \end{aligned}$$

$$
\begin{array}{lll}
& \texttt{rev}(\texttt{append}(a :: l, l_2)) & \\
= & \texttt{rev}(a :: \texttt{append}(l, l_2)) & (\texttt{append}\text{의 정의}) \\
= & \texttt{append}(\texttt{rev}(\texttt{append}(l, l_2)), [a]) & (\texttt{rev}\text{의 정의}) \\
= & \texttt{append}(\texttt{append}(\texttt{rev}(l_2), \texttt{rev}(l))), [a]) & (\text{귀납법의 가정에 의함}) \\
= & \texttt{append}(\texttt{rev}(l_2), \texttt{append}(\texttt{rev}(l), [a])) & (\texttt{append}\text{의 결합법칙}) \\
= & \texttt{append}(\texttt{rev}(l_2), \texttt{rev}(a :: l)) & (\texttt{rev}\text{의 정의})
\end{array}
$$

이제 $\forall l\ \texttt{rev}(\texttt{rev}(l)) = l$ 을 증명하기 위해서 위에서 증명한 보조 정리들을 사용할 수 있다.

$$
\begin{array}{lll}
& \texttt{rev}(\texttt{rev}(a :: l)) & \\
= & \texttt{rev}(\texttt{append}(\texttt{rev}(l), [a])) & (\texttt{rev}\text{의 정의}) \\
= & \texttt{append}(\texttt{rev}([a]), \texttt{rev}(\texttt{rev}(l))) & (\text{특성 Q에 의함}) \\
= & \texttt{append}(\texttt{rev}([a]), l) & (\text{귀납법의 가정에 의함}) \\
= & \texttt{append}(\texttt{append}(\texttt{rev}([]), [a]), l) & (\texttt{rev}\text{의 정의}) \\
= & \texttt{append}(\texttt{append}([], [a]), l) & (\texttt{rev}\text{의 정의}) \\
= & \texttt{append}([a], l) & (\texttt{append}\text{의 정의}) \\
= & a :: \texttt{append}([], l) & (\texttt{append}\text{의 정의}) \\
= & a :: l & (\texttt{append}\text{의 정의})
\end{array}
$$

3.4.10 예제 : 가정 하에서의, 또는 가정을 이용한 증명

리스트에 대한 함수들인 `fold_left`와 `fold_right`에 대해 생각해 보자. 이들은 다음과 같이 정의될 수 있다.

```
# fun fold_left f x = fn
      [] => x
      |(a::l) => fold_left f (f x a) l;;
# fun fold_right f l x = case l of
      [] => x
      |(a::l) => f a (fold_right f l x);;
```

이 함수들은 커리화한(currified) 형태이고, 각각은 함수 형태의 인자 하나를 받는다. 하지만 우리는 이전에 사용했던 증명 방법을 그대로 사용할 수 있다.

우리는 인자로 넘겨지는 함수 `f`가 결합 법칙을 만족하는 연산이고, 중성 원소(항등원) `e`를 가질 때, 다음이 성립함을 증명할 것이다.

$\forall l\ \texttt{fold\_left}\ f\ e\ l = \texttt{fold\_right}\ f\ l\ e$

우선 우리는 다음 사실을 쉽게 알 수 있다.

$$
\begin{array}{lll}
& \texttt{fold\_left}\ f\ e\ (a :: l) & \\
= & \texttt{fold\_left}\ f\ (f\ e\ a)\ l & (\texttt{fold\_left}\text{의 정의}) \\
= & \texttt{fold\_left}\ f\ a\ l & (\text{항등원의 성질})
\end{array}
$$

그리고

$$
\begin{array}{lll}
& \texttt{fold\_right}\ f\ (a :: l)\ e & \\
= & f\ a\ (\texttt{fold\_left}\ f\ l\ e) & (\texttt{fold\_right}\text{의 정의})
\end{array}
$$

이제 만일 우리가 다음을 증명할 수 있다면, 원 특성 역시 증명되게 된다.

$\forall x \, \forall l \, fx(\texttt{fold\_right}\ f\ l\ e) = \texttt{fold\_left}\ f\ x\ l$

이를 증명하기 위해서는 다음 두 가지를 증명하면 된다.

$$
\begin{array}{lll}
f\ x\ (\texttt{fold\_right}\ f\ []\ e) & & \\
\quad = \quad f\ x\ e & (\texttt{fold\_right}\text{의 정의}) \\
\quad = \quad x & (\text{항등원의 성질}) \\
\quad = \quad \texttt{fold\_left}\ f\ x\ [] & (\texttt{fold\_left}\text{의 정의})
\end{array}
$$

그리고

$$
\begin{array}{lll}
f\ x\ (\texttt{fold\_right}\ f\ e\ (a :: l)) & & \\
\quad = \quad f\ x\ (f\ a\ (\texttt{fold\_right}\ f\ l\ e)) & (\texttt{fold\_right}\text{의 정의}) \\
\quad = \quad f\ (f\ x\ a)\ (\texttt{fold\_right}\ f\ l\ e) & (\text{함수 f의 결합법칙 성립}) \\
\quad = \quad \texttt{fold\_left}\ f\ (f\ x\ a)\ l & (\text{귀납법의 가정에 의함}) \\
\quad = \quad \texttt{fold\_left}\ f\ x\ (a :: l) & (\texttt{fold\_left}\text{의 정의})
\end{array}
$$

연습 문제

3.1 다음 두 함수에 대해 생각해 보자.

```
# fun iter1 n f =
      if n=0 then id else compose f (iter1 (n-1) f);;
# fun iter2 n f x =
      if n=0 then x else f(iter2 (n-1) f x);;
```

이제 다음과 같이 정의된 함수 exp가 a^n을 계산해 냄을 보여라.

```
# fun exp a n =
      if n=0 then 1.0
      else (if (n mod 2) = 0 then 1.0 else a) *** exp (a***a) (n/2);;
```

3.2 2.3쪽에서 정의된 함수 map과 2.3.1쪽에서 list_hom을 이용하여 정의된 함수 map이 서로 같음을 보여라.

3.3 2.3.2쪽에서 정의된 함수 quicksort가 올바르게 동작함을 보여라.

3.5 표현식에 타입을 매기기

예전에 다시 쓰기 규칙들을 이용해 계산을 정의했을 때, 우리는 다시 쓰기 규칙들이 적용되었던 표현식들이 타입을 가지고 있다는 암묵적인 가정을 하고 있었다. 좀 더 정확히 말하면, 우리는 사실상 다음의 두 가지를 가정한 상태였다.

- 값매김할 표현식들은 타입을 가진다.
- 만일 표현식 E가 타입을 가지고 E ⇒E' 이면, E' 역시 타입을 가진다.

이 가정들은 결과를 가진다. 특히, 계산 과정에서 나타난 표현식이 기본 연산자의 적용이고, 그 인자들이 유효한 타입을 가질 경우, 이 연산자 적용은 수행될 수 있다. 즉, 값매김 과정은 타입 에러를 내지 않을 것이다.

우리는 이 특성에 대해 여기서 정형화된 증명을 하지는 않을 것이다. 하지만, 어떻게 그러한 증명이 가능한지를 대략 설명하고 지나갈 것이다. 먼저, 우리는 어떻게 표현식들에 타입을 매기는지를 정확히 정의해야 한다. 이는 타입 규칙 시스템에 의해 이루어진다.

타입 규칙 시스템은 다음과 같은 3항 연산(관계)를 사용한다.

$E \vdash e : t$

이는 다음과 같은 뜻을 가진다.

"표현식 e는 타입 환경 E에서 타입 t를 가진다."

타입 환경은 프로그램의 변수들과 그 타입을 연결해 주는 함수 또는 테이블이다. 여기 이를 위한 규칙들이 있다.

$$\frac{\mathtt{E(x)} = \mathtt{t}}{\mathtt{E} \vdash \mathtt{x} : \mathtt{t}} (\text{변수})$$

$$\frac{\mathtt{E} \vdash \mathtt{e_1} : \mathtt{t_1} \quad \ldots \quad \mathtt{E} \vdash \mathtt{e_n} : \mathtt{t_n}}{\mathtt{E} \vdash (\mathtt{e_1}, \ldots, \mathtt{e_n}) : (\mathtt{t_1} * \ldots * \mathtt{t_n})} (\text{N-묶음})$$

$$\frac{\mathtt{E} \vdash \mathtt{e_1} : \mathtt{bool} \quad \mathtt{E} \vdash \mathtt{e_2} : \mathtt{t} \quad \mathtt{E} \vdash \mathtt{e_3} : \mathtt{t}}{\mathtt{E} \vdash \mathtt{if\ e_1\ then\ e_2\ else\ e_3} : \mathtt{t}} (\text{조건})$$

$$\frac{(\mathtt{x} : \mathtt{t_1}) :: \mathtt{E} \vdash \mathtt{e} : \mathtt{t_2}}{\mathtt{E} \vdash \mathtt{fn\ x} \Rightarrow \mathtt{e} : \mathtt{t_1} \rightarrow \mathtt{t_2}} (\text{함수})$$

$$\frac{\mathtt{E} \vdash \mathtt{e_1} : \mathtt{t} \rightarrow \mathtt{t'} \quad \mathtt{E} \vdash \mathtt{e_2} : \mathtt{t}}{\mathtt{E} \vdash (\mathtt{e1\ e2}) : \mathtt{t'}} (\text{적용})$$

`let` 의 경우, 같은 변수를 두 개의 다른 타입으로 사용하는 것이 가능하게 하기 때문에 다루는 데 문제가 있다. (예를 들면, `let val id = fn x ⇒ x in id id end` 과 같은 경우이다.)

지금 당장은, 우리는 이 현상을 `let val id = fn x ⇒ x in id id end` 를 `fn x ⇒ x` `fn x ⇒ x` 로 대체함으로써 해결한다. 이는 두 하위 표현식 `fn x ⇒ x` 에 대해 적당한 타입을 가질 수 있도록 해 준다. 13장에서 우리는 이를 다루는 더욱 효과적인 방법에 대해 알아볼 것이다.

$$\frac{\mathtt{E} \vdash \mathtt{e_2}[\mathtt{x} \leftarrow \mathtt{e_1}] : \mathtt{t}}{\mathtt{E} \vdash \mathtt{let\ val\ x\ =\ e_1\ in\ e_2\ end} : \mathtt{t}} (\text{Let})$$

또한, 우리는 상수와 기본 연산자들에 대한 타입 규칙 역시 정의해야 한다. 이들은 매우 간단하므로 명시하지 않는다.

다음은 표현식 `let val id = fn x ⇒ x in id id end` 의 타입을 밝히는 과정을 보여 준다.

$$\frac{\dfrac{\dfrac{[\mathtt{x} : \alpha \rightarrow \alpha] \vdash \mathtt{x} : \alpha \rightarrow \alpha}{[] \vdash (\mathtt{fn\ x} \Rightarrow \mathtt{x}) : (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)} \quad \dfrac{[\mathtt{x} : \alpha] \vdash \mathtt{x} : \alpha}{[] \vdash (\mathtt{fn\ x} \Rightarrow \mathtt{x}) : (\alpha \rightarrow \alpha)}}{[] \vdash \mathtt{fn\ x} \Rightarrow \mathtt{x\ fn\ x} \Rightarrow \mathtt{x} : \alpha \rightarrow \alpha}}{[] \vdash \mathtt{let\ val\ id\ =\ fn\ x} \Rightarrow \mathtt{x\ in\ id\ id\ end} : \alpha \rightarrow \alpha}$$

위에서의 정형화된 정의는 다음 특성을 증명하는 데 기반이 된다.

(E ⊢ e : t) 이고 (e ⇒e') 이면 (E ⊢ e' : t) 이다.

3.6 요약

표현식에서 시작하여 값에서 끝나는 **계산**(computation)은 이전 표현식에 **다시 쓰기 규칙**(rewrite rule)을 적용하여 생성된 결과 표현식들의 연속으로 정의된다. 더 이상 다시쓸 수 없는 표현식은 **정규 형태**(normal form)라고 부른다. 값은 정규 형태에 있는 표현식이다.

프로그램과 관련된 다시 쓰기 규칙들의 집합은 3.1.1쪽에 정의된 **수렴하는**(convergent) 시스템을 이룬다. 특히, 이 특성은 표현식과 관련된 모든 끝나는 계산은 같은 정규 형태에서 끝난다는 사실을 내포하며, 우리는 이를 **값**(value)이라고 부른다. 이 방법으로, 우리는 우리가 사용하는 언어의 **의미 구조**(semantics)를 정의한다. 즉, 우리는 언어 내의 모든 표현식(정의되지 않는 것 포함)에 대해서 값을 연결하는 정형화된 방법을 가지게 된 것이다.

값매김 전략(computation strategy)은 값매김의 각 단계에서 적용 가능한 수많은 다시 쓰기 규칙들 중 어떤 것을 선택할지에 대한 일관되고 결정적인 규칙이다. 이 전략들은 **연역 법칙**(deduction rule)의 형태를 이용하여 정형화시켜 정의할 수 있다.

우리는 어떤 값매김 전략이 값이 존재하는 모든 표현식에 대해 항상 그 값을 계산해 낼 수 있을 경우 이를 **완전한**(complete) 전략이라고 말한다. nML에서의 전략 – **값에 의한 값매김**(evaluation by value) – 은 완전하지 않은 전략이다. 다른 전략, 예를 들어 **미루어진 값매김**(delayed evaluation)은 완전한 전략이다.

우리는 정형화된 등식 형태로 표현된 다시 쓰기 규칙과 자료 구조에 대한 귀납법 적용을 이용하여 프로그램의 특성에 대한 증명을 할 수 있다. 어떤 증명들은 부분적인 정확성만을 성립하게 해 주고, 이 경우 전체의 정확성을 보장하기 위해서 프로그램이 끝남을 증명하는 것이 필요하다.

타입에 대한 정형화된 정의는 모든 잘 정의된 타입을 가지는 표현식을 값매김하는 과정에서는 잘 정의된 표현식만이 생성된다는 것을 증명할 수 있게 해 준다. 따라서 정적인 타입 합성(static type synthesis)은 실행 도중의 타입 에러가 없음을 보장해 준다.

3.7 더 배울 내용들

다시 쓰여진 표현식의 연속으로 계산을 정의하는 것과 수렴성이라는 개념은 λ-Calculus에서 처음 사용되었다. [1] 수렴성의 개념은 또한 1차 등식 논리(first order equational logic) 에 기반한 자동 증명 시스템에서 중요한 역할을 한다.

우리가 이 장에서 살펴본 의미 구조들은 **조작적 의미 구조**(operational semantics) 로 분류될 수 있다. 그것들은 주로 문법적인 개념을 이용한다. 특히, 함수들은 수학적 의미에서의 진짜 함수가 아니라 단지 기계적으로 함수 적용 문법 내에 사용할 수 있는 표현식으로 다루어진다.

반면, **수학적 표기법을 이용한 의미 구조**(denotational semantics)는 진실한 수학적 실체를 프로그램에서의 표현식과 연관시켜 준다. 수학자 Dana Scott 는 이 종류의 의미 구조를 창시했다. 그는 **완전히 부분 순서를 가지는 집합**(complete partially ordered set), 또는 **CPO**의 개념에 기반하여 λ-Calculus의 표현식들에 대한 모델을 제시하고, 그 집합들 위에서의 연속 함수의 모델 역시 제시한 첫 번째 사람이다. 수학적 표기법을 이용한 의미 구조에 대해 가장 접근하기 쉽게 서술한 책은 [3] 이다. 여러분은 또한 [8] 과 [9] 에 대해서도 조사해 볼 수 있다.

프로그램의 정확성을 증명하기 위해서 많은 양의 연구가 수행되었다. 함수 중심적 프로그래밍에 대한 정확성 증명에 있어서는, 우리는 반드시, 존경할 만한 여러 연구들 중에서도, Milner의 `LCF` 시스템([4], [7])과 Boyer와 Moore의 시스템([2])에 대해 언급해야 한다.

현재의 연구는 생성적인 논리와 프로그램과 증명 사이의 유사성에 기반한 시스템의 개발에 집중되어 있다. 이 둘은 모두 끝나는 프로그램의 생성만을 허용하는 타입이 있는 고차 λ-Calculus로 표현될 수 있다. 이와 같은 접근에 기반한 시스템의 예로는 `Coq`([5])가 있다.

값매김 전략에 대해서는, 여러분은 현존하는 언어에서 두 가지의 다른 접근법을 살펴볼 수 있다. `ML`류의 언어들이나 스킴(`SCHEME`)과 같은 언어들은 함수적이지 않은 구성 방법들의 사용을 허가하는 방법으로 값에 의한 값매김을 선택했다. 다른 언어들은 미루어진 값매김 – 자주 "게으른" 값매김이라고 불리는 – 을 선택했다. 예를 들면 `MIRANDA` 또는 `HASKELL` 등이다.

4 장

기계 중심적인 측면들

완전히 함수 중심적인 관점에서 설명될 수 없는 nML의 기능들은 다음과 같은 이유로 기계 중심적인 특성이라고 불린다.

- 특별한 값매김 전략에서만 의미를 가지기 때문에(즉, 실행 순서와 관계되기 때문에)
- 기계 입장에서의 자료 구조의 표현과 관계되기 때문에

첫 번째 종류의 기계 중심적 측면에는 '예외' 들과 '입출력' 이 속한다.

두 번째 종류의 기계 중심적 측면에는, 우리가 **파괴적인**(destructive) 연산이라고 부르는 할당 연산과 같은 것이 포함된다. 그와 같은 연산의 효과는 자료 구조의 실제 구현에 대한 설명 또는 정형화된 의미 구조와 함께 설명해야만 완전하게 알 수 있다. (우리는 12장에서 이 개념에 대해 다룬다.) 그러나, 이 장에서 우리는 예제에 기반해서 이치에 맞는 설명 정도는 할 수 있다.

4.1 예외

2.3.4절에서, 우리는 부분 함수가 필요한 문제들에 대해 살펴보았다. 그것을 처리하기 위해, 우리는 다음과 같은 타입을 만들었었다.

```
#type 'a option = None | Some of 'a;;
```

하지만 이 해결책은 부분 함수를 필요로 하는 모든 상황을 고려해 주지는 않는다. 예를 들면, 나눗셈은 부분 연산(0으로 나누는 것이 정의되지 않으므로)이지만, `int`와 `real` 타입을 모든 수치 연산에서 `int option`과 `real option` 타입으로 대체하는 것은, 나눗셈이 아닌 모든 연산에 대해서도 그들의 인자가 정의되지 않을 경우에 대비하는 꼴이 되기 때문에 낭비이고, 따라서 실용적이지 않을 것이다. 이 방안의 가장 큰 문제점은 단순히 수행하기 너무 비효과적이기 때문에 수치 연산을 비실용적으로 만드는 것이다. 부수적인 이야기로, 만일 계산 도중에 0으로 나누는 일이 일어났다면, 그것은 분명히 제대로 된 결과가 아니기 때문에 더 이상 계산을 수행할 필요가 없다.

따라서, 우리는 계산 도중에 일어나는 어떤 예외적인 상황이건 고려할 수 있도록 **예외**(exception)이라는 개념을 사용한다. 이는 계산 외부적인 요인(메모리 부족 등)으로 일어나는 것들도 포함한다.

4.1.1 예외에 대한 예제

nML에서 예외는 많은 상황에서 일어난다. 예를 들면, 0으로 나누는 경우 일어난다.

```
#2+(1/0)*3;;
Uncaught exception:  Division_by_zero
```

또 빈 리스트의 첫 번째 원소를 사용하려고 했을 때도 일어난다.

```
#List.hd [];;
Uncaught exception:  Failure("hd")
```

시스템이 이 예외들로부터 복구하지 못할 경우(복구는 조금 후에 다룬다), 그 예외들은 대화 루프(또는 최상위 레벨)에 그 예외의 이름을 포함하는 메세지의 형태(`Division_by_zero` 또는 `Failure`)와 관련된 값(두번째 예제에서 "`hd`")으로 모습을 나타낸다.

4.1.2 예외 정의하기

시스템에서 특별한 순간에 정의된 예외들의 집합은 특별한 타입인 `exn` 을 생성한다. 이것은 자료 구성자로 각 예외들의 이름을 취한다. 상세히 하면, nML에 의해 제공되는 `exn` 타입은 다음 자료 구성자들을 포함한다.

```
type exn =
      ...
      | Division_by_zero
      | Failure of string
      | ...
```

평범한 다른 타입들과 달리, `exn` 타입은 확장될 수 있다. 사용자들은 그것에 구성자를 추가할 수 있고, 다음과 같이 쓰면 된다.

```
#exception Int_exception of int;; exception Int_exception of int
```

이는 타입 `exn` 의 정의에 다음과 같이 케이스를 추가한 것과 같은 효과를 지닌다.

```
type exn =
      ...
      | Division_by_zero
      | Failure of string
      | Int_exception of int
      | ...
```

4.1.3 예외 값을 만들어내기

평범한 방법을 사용하면, exn 타입의 값들은 평범한 방법으로 행동한다.

```
#Failure "oops";;
- :  exn = (Failure("oops"))
```

그것들은 단지 우리가 함수 raise를 그것들에 적용했을 때만 "예외적으로" 행동한다.

```
#raise(Failure "oops");;
Uncaught exception:  Failure("oops")
```

예를 들어 시스템의 리스트 관련 라이브러리에서 제공하는 hd 함수와 같은 역할을 하는 함수는 다음과 같이 정의할 수 있다.

```
#val hd = fn
      [] => raise(Failure "hd")
      |(a::l) => a;;
val hd :  'a list -> 'a = <fun>
```

예외가 구성 방법 raise를 사용하여 일어나고, 다음 장에서 설명할 구조가 그것을 처리하지 못할 경우, 예외는 최상위 레벨에 나타나게 된다.

```
#hd [];;
Uncaught exception:  Failure("hd")
```

4.1.4 예외 값을 처리하기

사용자 입장에서, 여러분은 raise에 의해 일어난 예외들을 처리할 수 있다. 그렇게 하기 위해서, 여러분은 구성 방법 handle 을 사용할 수 있다. 그것의 완전한 문법은 다음과 같다.

```
e handle p1 => e1
          ...
          pn => en
```

위와 같은 표현식의 값은 먼저 표현식 e 를 값매김하여 계산된다. 만일 그 값매김이 보통처럼 값 v 를 만들어 내면, 전체 표현식의 결과는 v가 된다. 만일 표현식 e 가 예외 값을 만들어 내면, 예외값을 처리하기 위해 패턴 집합 $p_1, \ldots, p_n$(이들은 모두 exn 타입을 가져야 한다)이 사용된다. 만일 p_i가 이 값에 매치하는 첫 번째 패턴이라면, 전체 표현식의 결과는 e_i가 된다. 만일 이 예외값에 어느 패턴도 매치하지 않으면, 이 예외는 handle 표현식 덩어리의 바깥으로 전달되게 된다.

예를 들어, 우리는 다음과 같은 방법으로 숫자 연산 에러에 의해 생성된 예외 Division_by_zero 를 처리할 수 있다.

```
#1/0 handle Division_by_zero => 0;;
- :  int = 0
```

표현식 $1/0$ 이 값매김될 때, 계산은 중지되고, 결과는 예외 값인 Division_by_zero 가 된다. 이 값은 그리고 나서 Division_by_zero 패턴에 의해 처리되고, 따라서 결과는 0이 된다.

4.1.5 예외 사용하기

물론, 예외는 어떤 함수 내에서 실패의 경우를 사전에 알아보기 위해 쓰인다.

```
#fun find p = fn
      [] => raise(Failure "find")
      |(a::l) => if p a then a else find p l;;
val find :  ('a -> bool) -> 'a list -> 'a = <fun>
#find (fn x => x mod 2 = 0) [3,1,7,8,13,14];;
- :  int = 8
#find (fn x => x mod 2 = 0) [3,1,7,13];;
Uncaught exception:  Failure("find")
```

또한 예외는 우리가 나중에 더 보게 될 많은 상황들에서 효과적으로 쓰일 수 있다.

미리 정의된 함수인 `failwith` 는 다음과 같이 정의할 수도 있다는 사실에 주의하라.

```
#fun failwith msg =
      raise(Failure msg);;
val failwith :  string -> 'a = <fun>
```

연습 문제

4.1 2.2.5쪽에 정의했던 타입 `'a tree` 를 다시 살펴보자. 타입 `(('a -> 'a -> bool) -> 'a tree -> 'a)` 을 가지는, 주어진 특성을 만족하는 원소를 트리에서 찾는 함수를 만들어라. 먼저 이 함수를 예외를 사용하지 않고 만들고, 다음에 그것을 대상 원소가 발견되었을 때 예외를 일으키도록 다시 써라. 만일 여러분이 이 함수를 handle 한다면, 여러분은 대상 원소의 값조차도 알아낼 수 있다.

4.2 명령 연속과 unit 타입

4.2.1 명령 연속

nML에서, 명령들의 연속은 대부분의 프로그래밍 언어에서 그러하듯이 세미콜론(;)으로 구분된다. 표현식 $(e_1;e_2)$ 의 의미 구조는 다음 규칙에 의해 정의된다.

$$\frac{\vdash \mathtt{e}_1 \Rightarrow \mathtt{v}_1 \quad \vdash \mathtt{e}_2 \Rightarrow \mathtt{v}_2}{\vdash (\mathtt{e}_1; \mathtt{e}_2) \Rightarrow \mathtt{v}_2}(\text{명령 연속})$$

즉, $(e_1;e_2)$ 의 값은 e_2의 값과 같다. 순수하게 함수적인 관점에서 보면, 이 구성 방법은 별 의미를 지니지 않는다.

반면, 앞 절에서 살펴본 예외를 고려하게 되면, 상황은 달라진다. 예외 고려 시, 이 구성 방법은 다음의 두 규칙에 따라 다른 행동을 보여주게 된다.

$$\frac{\vdash \mathtt{e}_1 \Rightarrow (\mathtt{exn}\mathtt{v}_1\mathtt{raised})}{\vdash (\mathtt{e}_1; \mathtt{e}_2) \Rightarrow (\mathtt{exv}\mathtt{v}_1\mathtt{raised})}(\text{연속 예외 1})$$

$$\frac{\vdash \mathtt{e_1} \Rightarrow \mathtt{v_1} \quad \vdash \mathtt{e_2} \Rightarrow (\mathtt{exn v_2 raised})}{\vdash (\mathtt{e_1; e_2}) \Rightarrow (\mathtt{exv v_2 raised})} \text{(연속 예외 2)}$$

결국 (e_1;e_2)의 값매김 과정에서 e_1에서의 예외의 발생은 중요한 역할을 하게 된다. 게다가, 이 구성 방법은 표현식 e_1과 e_2가 다른 효과, 즉 입출력 함수를 포함하는 경우에 진정한 존재 의미를 가진다.

4.2.2 unit 타입

우리가 기계 중심적인 기능들에 대해 다루게 되면, 인자가 없거나 돌려주는 값이 없는 함수들의 타입을 어떻게 할지에 대한 문제를 해결해야 한다.

nML에서는 이런 경우를, ()로 표기되는 값 하나만을 가지는 타입 `unit`을 이용하여 해결한다. 다음 절에서 이를 사용하는 예제를 볼 수 있다.

4.3 변경 가능한 자료 구조

nML은 여러분이 배열, 참조와 같은 자료 구조를 변경하는 것을 허락한다. 이 절에서는 이 기능이 여러분이 할 수 있는 일에 어떤 영향을 미치는지에 대해 다룬다.

4.3.1 배열

배열은 값묶음이나 리스트와 비슷하게 보이는 자료 구조이다.

- 배열은 `'a array` 라는 타입을 가지는 물체이다. 리스트와 같이, 그것들은 같은 타입을 가지는 물체들만을 포함할 수 있다. 그것들은 어떤 길이도 될 수 있다.
- `'a array` 타입은 리스트의 `cons`(::)와 닮은 어떤 종류의 연산도 제공하지 않는다. 값묶음과 같이, 배열은 단지 한 번의 연산으로 만들어지고, 그 이후에는 확장될 수 없다. e_1,...,e_n 값들을 포함하는 배열은 `[|`e_1`,...,`e_n`|]`와 같이 표기된다.
- 우리는 배열의 원소에 **인덱스**를 이용하여 접근한다. 길이 n인 배열 내의 원소의 인덱스는 반드시 0 이상 n-1 이하의 값을 가진다. 배열 `a`와 인덱스 `i`가 주어졌을 때, 우리는 그 인덱스에 해당하는 배열의 원소를 `a.[i]`와 같이 표기한다. 배열의 원소 하나에 접근하는 데는 상수 시간이 걸리며 이는 매우 짧은 시간인데, 그 이유는 배열의 원소들은 메모리에 순차적으로 저장되어 있기 때문이다. 따라서 각 원소의 주소는 단순히 첫 번째 원소의 주소에 그 원소의 인덱스를 더한 값이 된다.
- 배열의 각 원소들은 변경 가능하다.

배열의 경우 리스트와는 달리 어떤 원소가 그 배열에 속하는지를 로그에 비례하는 시간 내로 알 수 있다. 그렇게 하기 위해서 우리는 단순히 다음과 같이 이진 검색을 하면 된다.

```
#fun mem_array c e a =
      let fun find m n =
          if m > n then false
          else let val p = (m+n)/2 in
              if a.[p] = e then true
              else if c (a.[p],e) then find (p+1) n
              else find m (p-1)
          end
      in find 0 (Array.length a) end;;
val mem_array :  ('a * 'a -> bool) -> 'a -> 'a array -> bool = <fun>
#mem_array (<) 7 [|2,3,5,6,7,10,12,15|];;
- :  bool = true
```

배열의 원소는 할당 연산을 통해 변경될 수 있다. 배열 a의 i번째 원소를 표현식 e의 값으로 변경하기 위해서는, 다음과 같이 하면 된다.

```
a.[i] <- e
```

```
# val a = [|0,1,2,3|];;
val a :  int array = [|0, 1, 2, 3|]
# a.[1] <- 7;;
- :  unit = ()
# a;;
- :  int array = [|0, 7, 2, 3|]
```

물론, 이 연산은 우리가 지금까지 유지해 왔던 프로그래밍 스타일을 완전히 부수어 버린다. 사실상, 우리는 지금까지 항상 val v = e 형태의 정의를 실행한 후에는, 변수 v와 표현식 e는 항상 서로 바꿀 수 있다고 생각해 왔다. 특히 프로그램의 정확성에 대한 증명에서 그랬다. 하지만 할당 연산을 사용할 경우 이는 더 이상 맞지 않는다. 이는 바로 전의 예제에서 분명히 알 수 있다. 이 변화의 숨은 의미들은 4.4절에서 더욱 상세히 다룰 것이다.

이제 우리에게 할당 연산이 허용되었으므로, 우리는 PASCAL 이나 C에서 하던 것과 같은 스타일로 프로그래밍할 수 있다. 그런 의미에서, 다음은 배열 a내의 i번째 원소와 j번째 원소를 바꾸는 함수이다.

```
#fun swap a i j =
      let val x = a.[i] in
          a.[i] <- a.[j];
          a.[j] <- x
      end;;
val swap :  'a array -> int -> int -> unit = <fun>
```

다음은 배열의 원소들을 정렬하기 위한 quicksort 함수이다. 이는 원소들 간의 교환만을 사용한다. 함수 place는 배열 a의 [i,j] 구간을, 원소 a.[i]를 기준으로 해서 두 개로 분할한다.

```
#fun place c a i j =
      let fun place_rec i' j' =
          let
              fun move_right p =
                  if c (a.[i], a.[p]) orelse p=j' then p
                  else move_right (p+1)
              fun move_left p =
                  if c (a.[p], a.[i]) orelse p=i' then p
                  else move_left (p-1)
          in
              let
                  val k = move_right i'
                  val l = move_left j'
              in
                  if k>l then (swap a i l; l)
                  else if k=l then
                      if c (a.[l], a.[i]) then (swap a i l; l) else i
                  else (swap a k l; place_rec (k+1) (l-1))
              end
          end
      in
          place_rec (i+1) j
      end;;
val place :  ('a * 'a -> bool) -> 'a array -> int -> int -> int = <fun>
#fun quicksort c a =
      let fun quick i j =
          if i < j then
              let val p = place c a i j in
                  quick i (p-1);
                  quick (p+1) j
              end
      in
          quick 0 (Array.length a - 1)
      end
val quicksort :  ('a * 'a -> bool) -> 'a array -> unit = <fun>
#val a = [|18,2,13,4,6,25,1,10,12,9,15,7,3|];;
val a :  int array = [|18, 2, 13, 4, 6, 25, 1, 10, 12, 9, 15, 7, 3|]
#quicksort (<) a;;
- :  unit = ()
#a;;
- :  int array = [|1, 2, 3, 4, 6, 7, 9, 10, 12, 13, 15, 18, 25|]
#quicksort (>) a;;
- :  unit = ()
#a;;
- :  int array = [|25, 18, 15, 13, 12, 10, 9, 7, 6, 4, 3, 2, 1|]
```

4.3.2 참조

참조(reference)라는 개념은 어떤 면에서 변경 가능한 물체들의 정수를 보여 준다. 이 개념은 PASCAL이나 C와 같은 언어에서의 포인터(화살표)와 대응한다.

우리는 어떤 물체에 대해서든, ref 구성 방법을 이용하여 그에 대한 참조를 나타내는 값을 만들 수 있다. 이 때 ref 의 타입은 'a -> 'a ref 가 된다.

```
#val c = ref 0;;
val c :  int ref = ref 0
#val s = ref "hello";;
val s :  string ref = ref "hello"
#val l = [ref 1, ref 2, ref 3];;
val l :  int ref list = [ref 1, ref 2, ref 3]
```

참조되는 값(즉, 가리킴을 당하고 있는 값)은 연산자 "!" 를 이용하여 접근 가능하다.

```
#!c;;
- :  int = 0
#!(List.hd l);;
- :  int = 1
```

참조 형태의 값은 ":=" 로 표기되는 할당 연산에 의해 변경될 수 있다.

```
#c := !c + 3;;
- :  unit = ()
#c;;
- :  int ref = ref 3
#List.hd(List.tl l) := 7;;
- :  unit = ()
#l;;
- :  int ref list = [ref 1, ref 7, ref 3]
```

이 참조의 개념에 대한 많은 응용들 중에서, 우리는 반드시 내부 상태를 가진 함수의 정의와 호와 교점을 가진 그래프 구조 만들기에 대해서 언급하고 넘어가야 한다.

첫 번째 경우에 대해서는, 조금 후에 정의될 함수 gensym 이 있다. 이 기호를 만들어 내는 역할을 하는 함수는 unit -> string 의 타입을 가지고, 한번 실행할 때마다 새로운 문자열을 만들어 낸다. 이는 매 실행시마다 문자열 "ident" 뒤에 정수 하나를 붙여서 출력하는 것으로 가능하다. 이 정수는 함수의 내부 변수와 연결된 참조의 형태로 저장되며, 함수 호출시마다 1씩 증가된다.

```
#val gensym =
      let val count = ref (-1) in
          fn () => count := !count + 1;
              "ident"^(string_of_int !count)
      end;;
val getsym :  unit -> string = <fun>
```

```
#gensym();;
- :  string = "ident0"
#gensym();;
- :  string = "ident1"
#gensym();;
- :  string = "ident2"
```

4.3.3 PASCAL, C, LISP와의 비교

이 장에서 우리가 소개한 많은 기계 중심적인 구성 방법들은 nML을 이용하여 PASCAL이나 C와 같은 보다 전통적인 언어들과 비슷한 스타일로 프로그래밍할 수 있도록 해 준다. 또한 이 구성 방법들은 마치 LISP나 Scheme에서 rplaca, rplacd, displace_obj 연산을 사용해서 했던 것과 같이 파괴적으로 자료의 내용을 변경할 수 있게 해 준다. 위의 전통적인 언어들을 즐기던 사람들에게 있어서, nML와 다른 언어들간의 접근 방식 차이에 대해 토론해 보는 것은 흥미있는 일임에 틀림없을 것이다. 이와 같은 비교에서, 우리는 변수가 다루어지는 방법과 자료 구조가 유지되는 방법의 차이에 대해 살펴보아야 한다.

변수에 대해서는, nML과 기타 다른 언어들 – PASCAL, C, LISP – 간에 근본적인 차이가 있다. PASCAL, C, LISP에서의 할당 연산자(각각 :=, =, setq)들은 어떤 변수도 적용될 수 있고 그 변수의 값을 바꿀 수 있다. 하지만 nML에서 이는 불가능하다. 변수의 값은 바꿀 수 없고, 바뀌지 않으며, 이 사실은 쉽게 증명 가능하다. 만일 우리가 변수 x를 정의했고, val y = x 를 이용하여 다른 변수 y를 정의했다면, 테스트 x == y는 항상 참으로 남는다. nML에서 x := e라고 쓸 수 있는 유일한 상황은 변수 x가 t ref 꼴의 타입을 가질 때 뿐이다. 따라서 x와 그 값과의 연결은 바뀌지 않고, 변수 x의 값 내에 있는 화살표가 다른 곳을 향하게 되는 것이다. 전통적인 프로그래밍 언어에서는, 어떤 변수의 의미는 그것이 할당 연산자의 왼쪽에 있느냐 아니면 다른 곳에 있느냐에 따라 다르게 해석되어야 했다. 할당 연산자의 왼쪽에서는, 그것은 변수 그 자체[1]를 의미하는 반면, 다른 상황에서는 그것은 변수에 연결된 값을 나타내고, 따라서 문맥을 바꾸지 않고 그 값으로 대체될 수 있다. 예를 들면, 만일 변수 x가 3을 값으로 가진다면, 표현식 x := x + 1은 x := 3 + 1과 동등하지만, 분명히 3 := 3 + 1과는 다르다.(그리고 이는 의미없는 식이다.)

반면 nML에서는, 변수는 항상 그 값을 나타내고, 할당 연산자 왼쪽에서도 이는 참이다. 만일 이 값이 참조라면, 이 참조는 할당에 의해 변경될 수 있다.

역참조 연산자(!)는 그 값이 참조의 형태인 변수의 2가지 다른 사용방법을 명쾌하게 보여준다. 우리는 전통적인 언어에서 x := x + 1로 표기되는 연산을 x := !x + 1로 표기한다.

자료 구조의 동적인 생성에 대해서는, nML과 LISP는 PASCAL, C와 근본적인 차이를 보여준다. nML에서는, 단순히 레코드를 사용하는 것이나 자료 구성자를 적용하는 동작이 메모리를 내부적으로 할당하여 새로운 물체를 만든다. LISP의 경우 역시, 우리가 함수 cons 또는 연산자 quote를 썼을 때 그렇게 된다. 반면, PASCAL에서는, 새로운 물체는 함수 new 를 이용해서만 새로운 물체를 동적으로 생성할 수 있고, C에서는 malloc을 사용해야 한다. 이 물체들은 포인터(화살표)를 이용하여 참조되게 되고, 이 포인터는 nML에서의 참조 형태의

[1] 보다 정확하게는, 변수의 주소를 나타낸다.

값과 마찬가지로 할당 연산에 의해 변경될 수 있다. 결과적으로, C, PASCAL에서 물체는 매우 다른 방법으로 만들어지는 것처럼 보이지만, 동적인 자료 구조에 대한 포인터의 유지는 nML에서의 그것과 매우 유사하다.

nML과 LISP에서는, 포인터 변경 이후 쓸모없게 된 물체가 점유하고 있던 공간은 자동적으로 복구된다. 하지만, PASCAL, C에서는 사용자가 직접 어떤 물체가 더 이상 쓸모없다는 것을 dispose, free와 같은 함수를 이용하여 선언해 주어야 하고, 그것은 수많은 에러의 가능성을 내포한다.

컴파일에 대해 다루는 12장에서는 nML의 구현에서 어떻게 메모리가 관리되는지에 대해 다룰 것이다.

연습 문제

4.2 함수 gensym를 string -> unit -> string의 타입을 가지도록 고쳐서 사용자가 쓰고 싶은 문자열을 선택할 수 있도록 하라.
4.3 reset_gensym : unit -> unit 함수를 정의해서 gensym함수의 카운터를 0으로 초기화할 수 있도록 하라.
4.4 어떤 우선 순위의 리스트를 나타내기 위한 타입을 정의하라. 이 리스트의 원소들은 약간의 정보와, 그 정보의 우선 순위를 나타내는 정수 하나의 묶음으로 이루어져 있다. 리스트의 원소들은 우선 순위를 기준으로 정렬되어 있어야 한다. 그와 같은 리스트에 원소 하나를 삽입하는 함수를 만들어라.
4.5 참조 형태의 값을 이용하여 이중 연결 리스트를 만들어 보고, 이에 대해 동작하는 quicksort 함수를 만들어 보라.
4.6 그래프의 정점 – 약간의 정보와 그에 연결된 다른 정점들에 대한 정보를 가지는 – 들을 나타내는 타입을 정의하라. 다른 정점들에 대한 정보는 그 정점과 연결된 정점들의 집합으로 나타낸다. 이 그래프에 정점, 간선을 더하거나 빼는 함수들을 만들어 보아라.
4.7 LISP를 아는 사람의 경우, LISP의 자료 구조들을 나타내는 타입을 정의하라. 특히, rplaca, rplacd 연산을 정의해 보라.
4.8 nML에서의 참조 형태의 값이 어떻게 PASCAL에서의 '참조에 의한 인자 전달' 을 흉내낼 수 있는지 보여라.

4.4 파괴적 연산자들의 의미 구조

nML의 기계 중심적인 측면들은 다시 쓰기의 개념을 이용하여 바로 정의될 수는 없는데, 그 이유는 다시 쓰기의 개념은 메모리에서의 자료의 변경을 고려하기 않기 때문이다.

그러나, 이를 메모리에서의 변화가 명확하게 관리되는, 함수에 기반한 정형화된 시스템으로 변환하여 의미 구조를 정의하는 것이 가능하다.

이 변환에 대한 개념을 얻기 위해서, 96쪽에 정의된 함수 gensym 에 대해 살펴보자. 그리고 잠시 동안 이 함수가 이 시스템에서 함수 중심적이지 않은 유일한 함수라고 가정하자.

우리는 이 함수를 다음과 같은 변형된 함수 sym으로 대체할 것이다.

```
#fun sym c = ("ident"^(string_of_int c), c+1);;
```

이 함수는 정수 하나를 인자로 받아서, 이를 이용해 만들어진 심볼과 1 증가된 새로운 정수(카운터)로 이루어진 값묶음을 돌려준다.

이 새로운 함수를 gensym 대신 쓰기 위해서는, 우리는 다른 모든 nML함수들이 카운터값을 추가의 인자 형태로 받아들이고, 본래 결과에 함수 내에서 바뀌었을지 모르는 카운터값을 같이 묶어서 돌려주도록 해야 한다. 만일 고려 대상이 되는 함수가 내부에서 sym을 부르지 않았다면, 카운터값은 바뀌지 않을 것이다. 예를 들어, add_int 는 단순히 다음과 같이 정의된다.

```
#fun add_int c m n = (m+n, c);;
```

또 함수의 합성은 다음과 같이 된다.

```
#fun compose c f g x =
      let val (v1, c1) = g c x in
          f c1 v1
      end;;
```

따라서 수많은 함수들은 카운터 값을 단지 "넘겨주고", 그 카운터 값은 sym 함수에 의해서 어떤 비 함수 중심적인 연산도 수행하지 않고 증가된다.

그렇지만 이렇게 하는 것은 그리 현실적이지 못한데, 그 이유는 우리가 새로운 변경 가능한 값을 쓰고 싶을 때마다 모든 함수에 새로운 인자를 추가해야 하기 때문이다. 그렇게 하는 대신, 우리는 변경 가능한 값들을 모아서 메모리라고 불리는 새로운 하나의 물체를 만들 수 있다. 이는 loc -> value의 타입을 가지며, 여기서 loc은 메모리 주소의 타입이고 value는 쓰인 값들의 타입이다.

update 함수는 다음과 같이 순수 함수 중심적인 기능만을 사용해서 메모리를 변경한다.

```
#fun update mem loc value =
      fn loc' => if loc' = loc then value else mem loc;;
```

따라서 이제는 각 함수가 메모리에 해당하는 추가의 인자 하나만을 가지고, 돌려주는 값을 진짜 결과와 새로운 메모리의 묶음으로 바꾸는 것으로 충분하게 된다. 이 방법으로, 우리는 순수하게 함수적인 상황에서 메모리를 변경하는 모든 연산을 흉내낼 수 있다.

이 기술은 전통적으로 프로그래밍 언어의 수학 표기법을 이용한 의미 구조에서 쓰여 왔다. 이렇게 하고 나면 우리는 비 함수 중심적인 프로그램들에 대해, 그것들을 함수 중심적으로 변환하고 이에 대해 살펴봄으로써 여러 특성을 추론해 낼 수 있다. 또한, 3장에서 살펴본 여러 가지 방법들을 사용해서 이들을 다룰 수 있게 된다.

4.5 요약

우리는 nML의 주된 비 함수 중심적 특성들 – 명령 연속, 예외, 물체의 내용을 실제로 변경하기 등 – 에 대해 살펴보았다. 그것들을 이해하기 위해서, 여러분은 이 각각의 특성들을 nML의 값에 의한 값매김 전략에 관계시켜서 살펴봐야 한다. 게다가, 물체의 내용을 실체로 변경하는 연산들은 물체의 메모리 내에서의 표현 방법에 대한 어떤 가정들에 기반하고 있다.

C, PASCAL과 같은 전통적인 언어에서 찾을 수 있는 할당 연산들과 비교할 때, nML에서의 할당 연산들은 그 쓰임을 제한하는 약간의 "울타리" 들을 가지고 있다. nML에서는, 변수에 무언가 할당하는 것은 불가능하고, 그 변수에 할당된, 어떤 특별한 타입을 가지는 값들에 무언가를 할당하는 것만이 가능하다. 이 방법을 사용하면, 그런 특별한 타입을 가지지 않는 다른 값들은 변하지 않는다는 사실이 보존된다. 따라서, 예를 들어 변수 x가 변하지 않는 타입을 가진다면, 우리는 프로그램 내에서 그 변수는 항상 같은 값을 가진다는 사실을 확신할 수 있다. 이는 x가 내부에서 할당 연산을 수행하는지 안 하는지 모르는 임의의 함수의 인자로 전달되었다고 해도 마찬가지이다.

4.5.1 더 배울 내용들

우리는 예외 처리에 의해 수행되는 제어 구조와 할당 연산에 의해 수행되는 메모리 접근 구조를 명시적으로 만듬으로써, nML의 비 함수 중심적인 측면들, 또는 좀 더 일반적으로 더욱 전통적인 언어들의 의미 구조를 제시할 수 있다.

수학 표기법을 이용한 의미 구조에서는, 이를 목적으로 **연속됨**(continuation), **환경 함수**(environment function), **메모리 함수**(memofy function) 의 개념이 도입되었다. 이 개념들에 대한 소개를 위해서는 [3]를 보라.

II 편

응용

이제 이 책의 두번째 편을 공부할 차례이다. 이번 편에서는 nML로 여러가지 응용 프로그램을 작성해 볼 것이다. 각 장들은 서로 크게 연관이 없기 때문에, 독자의 흥미나 강의와의 연관성에 따라서 따로 읽어도 좋다.

5장은 형식 용어(formal terms)에 대해 기호 계산(symbolic computations)을 수행하는 것을 다룬다. 특히 자동 증명(automatic proof), 논리 프로그래밍(logic programming), 타입 유추(type synthesis)와 같은 많은 기호 계산에 대한 응용 프로그램에 사용되는 패턴 맞추기(pattern matching)이나 동일화(unification) 알고리즘을 정의한다.

6장에서는 큰 정보를 표현하는 균형잡힌 나무(balanced trees)를 사용하는 법을 살펴본다. 여기서는 알고리즘 강의에서 소개되는 이진 탐색 나무(binary search tree)와 AVL 나무의 아이디어를 사용한다.

7장은 그래프를 탐색하는 여러가지 방법을 깊이있게 다룬다. 6장에서 소개된 집합 표현 방식을 이용하여 빨간 원숭이(red donkey)나 솔리테어(solitaire)와 같은 게임 프로그램을 소개한다.

8장에서는 어휘 분석기(lexical analyzer)와 구문 분석기(syntax analyzer) 프로그램을 작성한다. 이것은 nML의 흐름(stream)이라는 아이디어를 이용한다.

9장은 어떻게 그림을 그리고 디자인하는 프로그램을 작성하는지 보여준다. 처음엔 개략적인 것을 설명하고, 그 후에 나무를 그리고 평면이나 표면을 타일로 덮는 MLgraph 라이브러리를 이용한다.

10장은 임의의 크기를 갖는 정수와 실수를 고려한 정확한 산술 연산을 다룬다. 여기서는 큰 숫자에 대한 완전하고 효율적인 라이브러리를 구현하기 보다는 이러한 작업이 가능하다는 것을 납득시키는 것이 목적이다.

5 장

형식 어구, 패턴 맞추기, 동일화

이번 장에서 우리는 변수를 가진 어구로 이루어진 일반적인 구조를 다룬다. 타입 유추(type synthesis), 값매김(evaluation), 편집, 형식 계산(formal computation), 자동 증명(automatic proof)과 같이 표현식을 형식적인 면에서 다루는 분야에서 유용한, 패턴 맞추기와 동일화에 대한 알고리즘을 살펴볼 것이다. 그리고 동일화 알고리즘을 타입 유추에 적용하는 것으로 마무리한다.

5.1 나무

이번 절에서 살펴 볼 기본 아이디어는 다음 절에서 변수를 가진 용어를 다루는 방법과 연관성이 있다.

우리는 노드의 개수가 다양한 나무에 대해 관심이 있다. 즉, 노드는 0개, 1개, 또는 그 이상의 자식 노드를 가질 수 있다. 한 노드의 자식 노드들의 집합은 리스트로 표현될 것이다. 말단 노드는 빈 리스트를 가진 노드로 표현된다. 그러므로 우리가 사용할 타입은 오직 한개의 생성자를 가진다.

```
# type 'a gentree = GenNode of 'a * 'a gentree list;;
```

평소와 같이, 이 타입의 객체에 대한 문법을 분석하는 함수가 있다고 가정한다. 나무의 구체적 문법 구조(concrete syntax)는 괄호를 이용한 간단한 형태이다.

- "a(t_1,$\cdots$,t_n)"은 GenNode(a,[t_1,$\cdots$,t_n])의 문법이다.
- "a"는 "a()"와 문법적으로 동일하다.

```
# gentree_of_string;;
val it:  (string -> 'a) -> string -> 'a gentree = <fun>
```

```
# gentree_of_string string_of_ident ‘‘a(b,c(d,e),f)’’;;
val it:  string gentree
       = GenNode
           ("a",        [GenNode ("b", []),
             GenNode ("c", [GenNode ("d", []), GenNode ("e", [])]),
             GenNode ("f", [])])
```

이 나무는 그림 5.1에 나타나있다.

5.1.1 몇가지 유용한 함수들

나무의 크기(size)는 나무가 가지고 있는 노드의 수이다. 크기는 다음과 같은 함수로 정의된다.

```
# val rec gentree_size = fn
  (GenNode(x,l)) ⇒ 1 + sigma(map gentree_size l);;
val gentree_size:  'a gentree -> int = <fun>
```

나무의 높이(height)는 나무의 가장 긴 가지의 길이이다. 높이는 다음의 함수로 정의된다.

```
# val rec gentree_height = fn
  (GenNode(x,l)) ⇒ 1 + (fold_left max_int 0 (map gentree_height l));;
val gentree_height:  'a gentree -> int = <fun>
```

때때로 나무에 나타나는 값들의 집합을 구하는 것도 필요하다.

```
# val rec gentree_set = fn
  (GenNode(x,l)) ⇒ union [x] (fold_left union [] (map gentree_set l));;
val gentree_set:  'a gentree -> 'a list = <fun>
```

만약 우리가 나무에 나타나는 값들의 리스트(중복된 리스트일 수도 있음)에만 관심이 있다면, 각각의 노드를 정확히 한번만 방문하면서 나무를 추적하는 방법의 개수 만큼 여러가지 리스트를 결과로 얻을 것이다.

모든 n에 대해서, 뿌리(root)로부터 거리 n+1만큼 떨어진 어떤 노드를 방문하기 전에 거리 n만큼 떨어진 모든 노드들을 모두 방문하는 방식을 폭 우선(breadth-first) 방식이라고 한다. 모든 레벨에서 노드를 왼쪽에서 오른쪽으로 또는 오른쪽에서 왼쪽으로 방문할 수 있다.

각각의 노드에 대해서 그것의 모든 자식 노드들을 다른 노드가 방문되기 전에 방문하는 방식을 깊이 우선(depth-first) 방식이라고 한다. 만약 자식노드들을 왼쪽에서 오른쪽의 순서로 방문하면 왼쪽 우선(leftmost)이라고 하며, 물론 오른쪽에서 왼쪽으로 방문하면 오른쪽 우선(rightmost)이라고 한다. 게다가 나무를 깊이 우선 방식으로 찾아가는 동안, 뿌리는 그것의 자식 이전 또는 이후에 방문될 수 있다. 따라서 우리는 접두 방식(prefix)과 접미 방식(postfix)를 구별할 수 있다.

그림 5.1의 나무에 대해서, 다음과 같이 노드를 방문하는 여러가지 순서가 있다.

폭 우선 왼쪽 우선 방식	a,b,c,f,d,e
폭 우선 오른쪽 우선 방식	a,f,c,b,e,d
깊이 우선 왼쪽 우선 접두 방식	a,b,c,d,e,f
깊이 우선 오른쪽 우선 접두 방식	a,f,c,e,d,b
깊이 우선 왼쪽 우선 접미 방식	b,d,e,c,f,a
깊이 우선 오른쪽 우선 접미 방식	f,e,d,c,b,a

예로써 깊이 우선, 오른쪽 우선, 접미 방식으로 노드의 리스트를 반환하는 함수를 살펴보자.

```
# val rec flat_gentree = fn
  (GenNode (x,l)) ⇒ x::fold_right ((prefix @) o flat_gentree) l [];;
val flat_gentree:  'a gentree -> 'a list = <fun>
```

5.1.2 나무를 위한 함수형

2.3.1절에서 리스트를 다루는, 재귀적으로 정의된 모든 함수들을 일반화 하는 list_hom이라는 함수를 알아보았다. 그러한 함수를 **동형성**(homomorphism)이라고 부른다. 이제 비슷한 방식을 소개하겠다.

```
# val rec gentree_hom = fn
  f (GenNode(x,l)) ⇒ f x (map (gentree_hom f) l)
val gentree_hom:  ('a -> 'b list -> 'b) -> 'a gentree -> 'b = <fun>
```

우리가 정의했던 모든 함수들은 gentree_hom를 이용해서 쉽게 재정의할 수 있다.

```
# val gentree_size =
  gentree_hom (fn x l ⇒ 1 + sigma l);;
val gentree_size:  'a gentree -> int = <fun>

# val gentree_height =
  gentree_hom (fn x l ⇒ 1 + fold_left max_int 0 l);;
val gentree_height:  'a gentree -> int = <fun>

# val gentree_set =
  gentree_hom (fn x l ⇒ union [x] (fold_left union [] l));;
val gentree_set:  'a gentree -> 'a list = <fun>

#val flat_gentree =
  gentree_hom (fn x l ⇒ x::fold_right (prefix @) l []);;
val flat_gentree:  'a gentree -> 'a list = <fun>
```

부수적으로, 함수 f를 나무의 모든 노드에 적용하는 것이나 나무의 거울상을 만드는 함수도 gentree_hom을 이용해 다음과 같이 쉽게 정의할 수 있다.

```
# val rec map_gentree = fn f ⇒
  gentree_hom (fn x l ⇒ GenNode(f x, l));;
val map_gentree:  ('a -> 'b) -> 'a gentree -> 'b gentree = <fun>
```

```
# val mirror_gentree =
  gentree_hom (fn x l ⇒ GenNode(x, rev l));;
val mirror_gentree:  'a gentree -> 'a gentree = <fun>
```

그렇지만 gentree_hom으로 함수를 정의하는 것은 엄청나게 비효율적일 수 있다. map을 사용하므로 gentree_hom 이 f를 적용하기 전에 임시 리스트를 만들기 때문이다. 이 임시 생성물 막을 수 있다. 예를 들어 뒤에서 정의할 함수 gentree_trav는 임시 리스트를 만들지 않는다. 여기서 gentree_hom의 인자 f가 세개의 인자 h, g, 그리고 x로 나뉘어 진다. 인자 g와 x는 함수 fold_right에 의해 노드의 자식들에게 재귀적으로 적용된 후의 결과물을 구성한다. 그리고 나서 그 결과는 함수 h에 의한 노드로부터 얻은 정보와 함께 재구성될 것이다.

```
# fun gentree_trav h g x (GenNode (a,l)) =
  h a (fold_right (g o (gentree_trav h g x)) l x);;
val gentree_trav:  ('a -> 'b -> 'c) -> ('c -> 'b -> 'b)
                   -> 'b -> 'a gentree -> 'c = <fun>
```

이 정의는 gentree_hom을 이용한 다음의 gentree_trav보다 좀더 효율적이다.

```
# fun gentree_trav f g x =
  gentree_hom (fn a l ⇒ f a (fold_right g l x));;
val gentree_trav:  ('a -> 'b -> 'c) -> ('c -> 'b -> 'b)
                   -> 'b -> 'a gentree -> 'c = <fun>
```

우리가 전에 정의한 모든 함수들은 다음과 같이 gentree_trav로 다시 정의할 수 있다.

```
# val gentree_size = gentree_trav (fn x y ⇒ y+1) add_int 0;;
val gentree_size:  'a gentree -> int = <fun>

# val gentree_height = gentree_trav (fn x y ⇒ y+1) max_int 0;;
val gentree_height:  'a gentree -> int = <fun>

# val gentree_set = gentree_trav (fn x l ⇒ union [x] l) union [];;
val gentree_set:  'a gentree -> 'a list = <fun>

# val flat_gentree = gentree_trav cons (prefix @) [];;
val flat_gentree:  'a gentree -> 'a list = <fun>

# val map_gentree = gentree_trav (fn x l ⇒ GenNode(f x, l)) cons [];;
val map_gentree:  ('a -> 'b) -> 'a gentree -> 'b gentree = <fun>
```

함수 mirror_gentree도 이런 방식으로 정의될 수는 있지만 이 정의는 다루기 불편하고 비효율적이다.

```
# val mirror_gentree =
  gentree_trav (fn x l ⇒ GenNode(x,l)) (fn x l⇒ l@[x]) [];;
val mirror_gentree:  'a gentree -> 'a gentree = <fun>
```

우리는 나무를 기계 중심적인 방식으로 조사할 수 있다. 함수 do_gentree는 왼쪽 우선 접두 방식의 순서로 나무의 모든 값들에 함수 f를 적용한다.

```
# fun do_gentree f (GenNode(x,l)) =
  f x; do_list (do_gentree f) l;;
val do_gentree:  ('a -> 'b) -> 'a gentree -> unit = <fun>
```

5.1.3 나무에서 경로 찾기(occurrence)

어떤 상황에서는 나무의 노드를 정확히 지정할 수 있는 것이 필요하다. 이를 위해 나무의 뿌리로부터 각 노드에 이르는 경로(path)를 이용한다. 경로를 정의하기 위해서는 주어진 노드에 도달하기 위해 여러 자식 노드들 중에서 어떤 선택을 해야하는 지를 리스트로 주면 된다. 우리는 노드가 n개의 자식노드를 가지고 있을 때 왼쪽에서 오른쪽으로 1부터 n까지 숫자를 매기기로 한다.

어떻게 노드를 지정할 수 있는 지에 대해 알아보자.

- 나무의 뿌리는 빈 리스트로 지정된다.
- 원하는 노드가 뿌리의 i번째 자식이 있는 가지에 존재한다면, 그곳에 도달하는 경로는 (i::path)이다. 여기서 path는 원하는 노드에서 i번째 자식 노드로의 경로이다.

이런 방식으로 표시된 경로는 occurrence라고 불린다. 이것들은 nML에서 정수 리스트로 표현된다. 형식적으로는 이것을 알파벳 N(자연수)을 기호로 하는 "단어(words)"(즉, 숫자의 문자열)로 간주할 수 있다.

함수 at는 infix로 선언된다. 이것은 occurrence가 있는 부 나무(sub tree)와 나무를 관계짓는다. 여기서 부 나무는 다음과 같이 occurrence에 의해 지정된 노드에 위치하고 있다.

```
# fun at (GenNode(x,l)) = fn
    [] => (GenNode(x,l))
  | (i::occ) => at (List.nth l i) occ;;
val at:  'a gentree -> int list -> 'a gentree = <fun>
```

다른 나무에 의해 주어진 occurrence에 위치한 부 나무를 바꾸기 위해서는 함수 replace_occ를 사용한다.

```
# fun replace_occ (GenNode(x,l)) occ t2 =
   case occ of
        [] ⇒ t2
      | (i::occ) ⇒ let val ti = List.nth l i in
          GenNode(x, replace_in_list l i (replace_occ ti occ t2)) end;;
val replace:  'a gentree -> int list -> 'a gentree -> 'a gentree = <fun>
```

5.1.4 나무와 추상적 문법 구조(Abstract Syntax)

n개 노드를 갖는 나무는 유일한 타입으로 추상적 문법 구조 (또는 생성자를 가진 nML의 타입)을 표시할 수 있다. 이런 방법으로 문법 구조를 다루는 일반적인 함수를 쓸 수 있다.

예를 들어 다음의 타입을 생각해 보자.

```
# type ty = A | B of ty | C of ty * ty;;
```

예를 들어 "A", "B", 그리고 "C"에 해당하는 생성자 A, B, 그리고 C를 만들 수 있다면, 다음과 같이 타입 string gentree의 값으로 타입 ty의 값을 표현할 수 있다.

```
# val rec gentree_of_ty = fn
    A ⇒ GenNode("A", [])
  | (B(x)) ⇒ GenNode("B",[gentree_of_ty x])
  | (C(x,y)) ⇒ GenNode("C", [gentree_of_ty x, gentree_of_ty y]);;
val gentree_of_ty:  ty -> string gentree = <fun>
```

이제 이런 종류의 변환이 생성자를 가진 타입들 전체에 대해 일반화 될 수 있음을 명확히 알았을 것이다. 반대 방향의 변환도 역시 쉽게 프로그램될 수 있지만, 이를 위해서는 타입 string gentree의 모든 나무들이 타입 ty의 값들에 해당하지 않는다는 사실에 주의해야한다. 사실상, 문자열 "A", "B", 그리고 "C"만이 생성자에 해당하고, 차수(arity)를 주의해야 한다.

이런 정보는 관계 리스트의 형태로 모듈타입(signature)에 의해 전달된다.

```
# type 'a sign = ('a * int) list
```

함수 arity는 주어진 모듈타입에서 기호의 차수를 반환한다. 기호가 알려지지 않은 경우에는 예외(exception) Sig_error을 발생한다.

```
# exception Sig_error;;
exception Sig_error

# fun arity sign x =
  List.assoc x sign
  handle _ => raise Sig_error;;
val arity:  ('a * 'b) list -> 'a -> 'b = <fun>
```

다음의 함수는 나무가 주어진 모듈타입을 따르는지를 확인한다.

```
# fun ok_sig sign t =
  gentree_trav (fn x bl => (List.for_all (fn b => b=true) bl)
                           && ((arity sign x) = List.length bl))
          cons [] t
  handle Sig_error => false;;
val ok_sig:  ('a * int) list -> 'a gentree -> bool = <fun>
```

이런 방식으로 차수를 검사하는 것은 한개의 타입의 객체를 다루는 상황에서만 가능하다. 이런 것은 일반적인 경우가 아니다. 예를 들어, 다음과 같은 상호 재귀적인 타입을 생각해보자.

```
# type ty1 = A | B of ty1 * ty2
  and ty2 = C | D of ty2 * ty1;;
```

모듈타입이라는 아이디어는 타입을 고려하도록 확장되어야 한다. 각각의 생성자의 차수를 정의하는 정수와 연관시키기 보다는, 인자의 타입의 리스트와 결과의 타입을 갖는 쌍과 연관시킬 것이다.

```
# type ('a, 'b ) signa = ('a * ('b list * 'b)) list;;
```

다음의 함수는 주어진 모듈타입에서 기호의 타입이 무엇인지를 알려준다.

```
# fun get_type sign x =
  List.assoc x sign
  handle _ => raise Sig_error;;
val get_type:  ('a * 'b) list -> 'a -> 'b = <fun>
```

타입 ty1, ty2인 경우에는 다음과 같다.

```
# val sig12 = [("A", ([],"ty1")), ("B", (["ty1","ty2"],"ty1")),
               ("C", ([],"ty2")), ("D", (["ty2","ty1"],"ty2"))];;
val sig12:  (string * (string list * string)) list
   = [("A", ([], "ty1")), ("B", (["ty1", "ty2"], "ty1")),
      ("C", ([], "ty2")), ("D", (["ty2", "ty1"], "ty2"))]
```

다음의 함수는 나무가 실제로 주어진 모듈타입을 따르는지 확인한다. 잘 따르는 경우에는 타입을 반환하고, 반대 상황에서는 실패한다.

```
# fun typing sign t =
  gentree_trav (fn x ts =>
                  let val (ts1, t) = get_type sign x
                  in if ts = ts1 then t else raise Sig_error
                  end)
  cons [] t;;
val typing:  ('a * ('b list * 'b)) list -> 'a gentree -> 'b = <fun>
```

함수 typing는 초보적인 타입 유추기이다. 여기서 초보적인이라는 설명이 붙은 이유는 이것이 오직 "편평한(flat)" 타입만을 다루기 때문이다. 이 함수가 고려하는 타입은 앞의 예에서 문자열로 표시되었던 상수 타입이다. 나중에 더 복잡한 타입을 고려하려면 어떻게 해야하는지 살펴볼 것이다.

5.2 변수를 가진 어구

지금까지 타입 gentree가 모든 종류의 표현식을 일관성있게 표현할 수 있다는 것을 배웠다. 그러나, 이런 표현식들은 오직 연산자와 상수만을 포함하였다. 이제 어떤 표현식이라도 다루기 위해 변수와 치환이라는 개념을 생각해야 한다. 5.1.3절에서 정의한 함수 replace_occ를 이용하여 특정한 종류의 치환을 할 수 있다. 그러나 이렇게 하는 것은 치환이 일어나는 경로가 직접적으로 명시되어야(즉, 뿌리로부터 목적지에 이르는 경로를 말해야 한다.) 하는 것을 가정한다. 하지만 이것은 대부분의 경우에 있어서 편리한 방식이 아니다. 우리가 수학에서 하는 것처럼, 치환이 이루어지는 위치를 변수에 의해 명시하는 것이 더 자연스럽다. 그래서 이제 우리는 변수라는 아이디어를 소개하려고 한다.

5.2.1 변수

곧 정의할 타입 ('a,'b) term은 타입 'a gentree와 비슷하지만, 타입 'b의 값에 의해 표현되는 변수라는 아이디어가 추가되어 있다.

```
# type ('a, 'b) term = Term of 'a * ('a, 'b) term list
                     | Var of 'b;;
```

이 절의 나머지 부분에서 우리는 타입 (string, string) term의 객체만을 사용할 것이다. 그래서 우리는 다음과 같이 문법을 분석하는 함수와 특별한 출력함수를 사용한다.

```
# term_of_string;;
val it:  string -> (string, string) term = <fun>

# print_term;;
val it:  (string, string) term -> unit = <fun>
```

우리는 상수를 인자로 빈 리스트를 적용한 함수로 표시함으로써, 문법적으로 상수와 변수를 구분할 것이다.

```
# term_of_string ``a(b(),c)'';;
val it:  (string, string) term = Term(``a'', [Term(``b'',[]), Var ``c''])

# new_printer ``term'' print_term;;
val it:  unit = ()

# term_of_string ``a(b(),c)'';;
val it:  (string, string) term = a(b(),c)
```

어구를 조사하기 위해, 함수 term_trav를 이용할 것이다. 이것은 gentree_trav와 비슷하지만 변수를 고려하도록 수정되었다.

```
# fun term_trav f g start v ter = case ter of
    (Term(oper,sons)) ⇒
          f(oper, fold_right (g o (term_trav f g start v)) sons start)
  | (Var n) ⇒ v n;;
val term_trav:  ('a * 'b -> 'c) -> ('c -> 'b -> 'b)
                  -> 'b -> ('d -> 'c) -> ('a, 'd) term -> 'c = <fun>
```

함수 vars는 어구에 나타나는 변수의 집합을 반환한다.

```
# fun vars t = term_trav snd Set.union [] (fn x ⇒ [x]) t;;
val vars:  ('a, 'b) term -> 'b list = <fun>
```

함수 occurs는 주어진 변수가 주어진 어구에 나타나는지 확인한다. 이것은 다음과 같다.

```
# fun occurs v t = List.mem v (vars t);;
val occurs:  'a -> ('b, 'a) term -> bool = <fun>
```

나무구조와 비슷하게, 어구는 일반적으로 차수와 타입 제한 조건을 따라야 한다. 이전 절에서 정의된, 차수를 검사하고 타입을 계산하는 함수들은 쉽게 어구로 확장될 수 있다. 단순히 차수를 검사하는 경우에 변수는 상수와 비슷하게 된다. 타입 유추의 경우에는 변수 자체가 타입이라고 가정해야 한다.

5.2.2 치환(substitution)

치환은 변수를 어구로 바꾸는 연산이다. 예를 들어 변수 x를 어구 g(z)으로 바꾸는 치환이 어구 f(x,y,x)에 적용되면, 어구 f(g(z),y,g(z))를 만들 것이다.

우리는 치환을 타입 ('b*('a,'b) term) list의 관계 리스트로 표현할 것이다. 이 관계 리스트에 나타나지 않는 변수는 치환 과정동안에 변하지 않는다. 여기에 치환을 멋지게 출력하는 함수가 있다.

```
# fun print_subst subst =
  do_list (fn (x,t) ⇒ print_string x,
                       print_string `` --> '',
                       print_term t,
                       print_newlist())
          subst;;
val print_subst:  (string * (string, string) term) list -> unit = <fun>
```

함수 apply_subst는 어구에 치환을 적용한다. 즉, 변수가 나타나면 치환에 있는 표현으로 바꾼다.

```
# fun apply_subst subst ter = case ter of
    (Term (f,tl)) ⇒ Term(f, (List.map (apply_subst subst) tl))
  | (Var x) ⇒ List.assoc x subst
                    handle _ ⇒ Var x;;
val apply_subst:  ('a * ('b, 'a) term) list -> ('b, 'a) term -> ('b, 'a) term
  = <fun>

# let val subst = [(``x'', term_of_string ``g(z)'')]
  in apply_subst subst (term_of_string ``f(x,y,x)'')
  end;;
val it:  (string, string) term = f(g(z),y,g(z))
```

두개의 치환 σ_1, σ_2의 합성은 단순히 그들의 표현으로 관계 리스트의 형태로 계산될 수 있다. 우리가 σ_2에 의한 상의 σ_1의 상을 얻는 방식으로, σ_2에 의해 수정된 변수의 합성에서 상을 얻는다. σ_2에 의해 수정되지 않은 변수의 합성에서 상은 σ_1에 의한 상이다. 그러므로 합성의 관계 리스트로 상을 얻을 수 있다. 앞으로 이런 방식으로 진행한다. 우선, σ_2에 해당하는 리스트를 얻는다. 여기에서 오른쪽 원소를 σ_1으로 고친다. 그리고 나서 σ_1에 해당하는 리스트를 얻는다. 이것으로부터 σ_2에 의해 수정된 변수를 제거한다. 마지막으로 이 결과들을 연결한다.

```
# fun compsubst subst1 subst2 =
   (List.map (fn (v,t) ⇒ (v, apply_subst subst1 t)) subst2)
   @ (let val vs = List.map fst subst2
      in filter (fn (x,t) ⇒ not (mem x vs)) subst1 end);;
val compsubst:  ('a * ('b * 'a) term) list -> ('a * ('b * 'a) term) list
             -> ('a * ('b * 'a) term) list = <fun>

# let val subst1 = [(``x'',term_of_string ``g(x,y)'')]
      and subst2 = [(``y'',term_of_string ``h(x,z)'')]
  in print_subst (compsubst subst1, subst2) end;;
y ⇒  h(g(x,y),z)
x ⇒  g(x,y)
val it:  unit = ()

# let val subst1 = [(``x'',term_of_string ``g(x,y)'')]
      and subst2 = [(``y'',term_of_string ``h(x,z)''), (``x'',term_of_string ``k(x)''
  in print_subst (compsubst subst1, subst2) end;;
y ⇒  h(g(x,y),z)
x ⇒  k(g(x,y))
val it:  unit = ()
```

σ_1에서 σ_2에 의해 수정된 변수를 제거하지 않음으로써 compsubst의 정의를 간단히 할 수 있다. 관계 리스트는 왼쪽에서 오른쪽으로 검색되기 때문에, 가장 왼쪽의 정의만이 고려될 것이다. 따라서 다음과 같이 쓸 수 있다.

```
# fun compsubst subst1 subst2 =
   (List.map (fn (v,t) ⇒ (v, apply_subst subst1 t)) subst2) @ subst1;;
val compsubst:  ('a * ('b * 'a) term) list -> ('a * ('b * 'a) term) list
             -> ('a * ('b * 'a) term) list = <fun>
```

5.2.3 여과와 패턴 매칭 (Filtering and Pattern Matching)

어구 t로 어구 u를 **여과** 또는 패턴 매칭을 하는 것은 t에서 u로 변화시키는 치환에서 필요한 연산이다. 어구 t가 단일 변수 x이라면, 치환 (x, u)는 분명히 유일한 결과를 낸다. 만약 어구 t가 $f(t_1, \cdots, t_n)$의 형태라면, 결과가 존재하기 위해서는 다음과 같은 조건이 필요하다.

- u는 $f(u_1, \cdots, u_n)$의 형태이어야 한다.

- 모든 i에 대해서, u_i와 t_i사이의 패턴 매칭이 가능해야 한다.
- u_i와 t_i사이의 패턴매칭에 해당하는 다양한 치환이 상호 양립(mutually compatible)해야한다. 즉, 이것이 같은 변수에 대해서 다른 결과를 만들어 내면 안된다.

그렇지 않으면 결과는 존재하지 않는다.

패턴 매칭 함수는 두개의 어구의 구조를 살펴본다. 이를 통해 이 구조가 양립할 수 있는 지를 점검하고, 더 간단한 어구에 대한 패턴 매칭 문제로 작게 자른다. x가 변수인 (x, u)의 형태를 만났을 때, 이 쌍을 $u \neq u'$인 (x, u')을 찾은 적이 없음을 점검한 곳에 추가한다. 이 점검을 통한 추가는 함수 som_subst에 의해 수행된다.

```
# exception Match_exc;;
exception Match_exc

# fun som_subst s1 s2 =
   fold_left (fn subst(x,t) ⇒
                      let val u = List.assoc x subst
                      in if t = u then subst
                          else raise Match_exc end
                      handle Not_found ⇒ (x, t)::subst)
      s1 s2;;
val som_subst:  ('a * 'b) list -> ('a * 'b) list -> ('a * 'b) list = <fun>
```

함수 matching은 이 패턴 매칭을 만들어 낸다.

```
# fun matching (t1,t2) =
      let fun matchrec subst = fn
        (Var v,t) ⇒ som_subst [v,t] subst
      | (t,Var v) ⇒ raise Match_exc
      | (Term(op1,sons1), Term(op2,sons2)) ⇒
          if op1 = op2 then fold_left matchrec subst (combine(sons1, sons2))
          else raise Match_exc;;
      in matchrec [] (t1,t2) end
val matching:  ('a, 'b) term * ('a, 'c) term -> ('b * ('a, 'c) term) list = <fun>
```

5.2.4 동일화(Unification)

t와 u, 두개의 어구가 주어진 상황에서 이들의 **동일화**과정에는 그들을 동일화시킬 치환 σ가 존재하는지 찾아보는 것도 포함된다. 즉, $\sigma(t) = \sigma(u)$을 만족하는 치환을 찾아본다. 따라서 치환 σ는 t와 u의 **동일화 장치(unifier)**라고 한다.

만약 어구 t가 변수 x를 만들어낸다면, 변수 x가 어구 u에 나타나지 않는 경우에 답은 (x, u)이다. 만약 x가 u에 나타나면 답은 존재하지 않는다.

비슷한 방식으로, 만약 어구 u가 변수 x로 줄여지면, x가 t에 나타나지 않는 경우에 답 (x, t)를 얻을 수 있다.

다른 경우로, t와 u가 모두 $f(t_1, \cdots, t_m)$과 $g(u_1, \cdots, u_n)$의 형태를 갖는다고 하자. 만약 f와 g가 다르다면, 동일화는 불가능하다. 만약 $f = g$라면, $m = n$이 되고 따라서 t와 u를 동일화하는 문제는 모든 i에 대해서 $\sigma(t_i) = \sigma(u_i)$인 치환 σ를 찾는 것으로 줄어든다.

결론적으로 두개의 어구 t와 u를 동일화하는 문제를 다루기 위해서는 동시에 어구쌍의 집합을 동일화하는 문제를 다룰 수 있어야 한다.

이를 위해 다음과 같은 방식을 이용할 것이다. 단일화할 어구쌍의 집합 $((t_1, u_1), (t_n, u_n))$이 주어졌을 때, 먼저 t_1와 u_1을 동일화 한다. 만약 이 동일화가 성공하면, 동일화 장치 σ_1을 만든다. 그리고 나서 다음을 시도한다.

$$((\sigma_1(t_2), \sigma_1(u_2)), \cdots, (\sigma_1(t_n), \sigma_1(u_n)))$$

만약 이것이 성공하고 동일화 장치 σ를 만들어 내면, 이 둘을 합성한 $\sigma \, o \, \sigma_1$이 분명히 $((t_1, u_1), \cdots, (t_n, u_n))$의 동일화 장치이다.

함수 unify는 이 알고리즘을 구현한 것이다.

```
# exception Unify_exc;;
exception Unify_exc


# val rec unify = fn
    (Var v,t2) ⇒ if Var v = t2 then [] else
                 if occurs v t2 then raise Unify_exc
                 else [v,t2]
  | (t1,Var v) ⇒ if occurs v t1 then raise Unify_exc
                 else [v,t1]
  | (Term(op1, sons1), Term(op2, sons2)) ⇒
      if op1 = op2 then
          let fun subst_unif s (t1,t2) =
              compsubst (unify(apply_subst s t1, apply_subst s t2)) s
          in (fold_left subst_unif [] (combine(sons1,sons2)) end
      else raise Unify_exc;;
val unify:  ('a, 'b) term * ('a, 'b) term -> ('b * ('a, 'b) term) list = <fun>
```

이것이 동작하는 예는 다음과 같다.

```
# let val t1 = term_of_string ``f(x,h(y))''
          and t2 = term_of_string ``f(k(z),z)''
  in print_subst (unify(t1,t2)) end;;
x ⇒  h(k(y))
z ⇒  h(y)
val it:  unit = ()
```

패턴매칭의 문제에 비해서 동일화 문제는 일반적으로 유일한 답을 갖지 않는다. 실제로 σ가 동일화 장치라면 모든 치환 ρ에 대해서 ρ와 σ를 합성한 $\rho \; o \; \sigma$도 역시 동일화 장치이다. 함수 unify에 의해 구현된 알고리즘이 다른 모든 것들에 비해 더 일반적인 동일화 장치, σ_0를 계산함을 보일 수 있다. 여기서 더

일반적인 동일화 장치라는 것은 이 특정 동일화 장치와 치환을 합성함으로써 다른 모든 동일화 장치 - 즉, 다른 모든 동일화 장치는 $\rho \ o \ \sigma_0$의 형태이다 - 를 만들 수 있다는 관점에서 일반적이라는 것이고, 특별히 MGU, `most general unifier`라고 부른다.

동일화 알고리즘은 전산학의 많은 부분에서 유용하다. 예를 들어 nML에서 타입을 유추하는 것을 가능하게 한다.

6 장

균형잡힌 나무

이번 장은 큰 크기의 자료 집합을 다루기 위한 일련의 함수들을 정의한다. 이 함수들은 자료 구조로 나무 구조를 사용하는데, 이것은 자료 집합의 크기가 계산과정 중에 동적으로 커지거나 작아지는 것을 표현하는 데에 나무 구조가 매우 적합하기 때문이다.

우리가 사용하는 자료 구조는 이항 나무이다. 우리가 제안하는 알고리즘은 이 이항 나무를 균형 잡히도록 유지하도록 하고, 이 성질은 수행하고자 하는 연산의 효율성을 좋게 한다.

자료 집합을 다루는 주요 연산은 다음과 같다.

- 자료 집합에서 원소 하나를 찾기
- 자료 집합에서 원소 하나를 추가하기
- 자료 집합에서 원소 하나를 제거하기

자료 집합에서 원소를 찾기 위해서, 단순히 집합의 모든 원소를 순차적으로 살펴볼 수 있다. 하지만, 이것은 자료 집합의 크기와 비례하는 시간이 걸린다. 따라서 이것의 복잡도는 $O(n)$이다. 이 방식은 우리가 자료 집합의 특성을 모르기 때문에, 일반적인 방식으로 자료 집합을 다루는 경우에 사용할 수 있는 유일한 방식이다.

좀더 효율적으로 찾기 위하여, 자료 집합의 구조에 대해 가정을 세워야 한다. 예를 들어, 만약 자료가 정렬되어 있다고 하면, 정렬된 리스트로 집합을 표현하기 위해 4.3.1절에서 했듯이 이항 탐색을 이용할 수 있다. 찾는데 걸리는 시간은 자료 집합의 크기의 로그에 비례하며, 따라서 복잡도는 $O(\log(n))$이다.

그렇지만 정렬된 리스트에 원소를 추가하는 것은 비싼 연산이다. 만약 우리가 원소를 추가할 때 새로 리스트를 할당하는 것을 원하지 않는다면, 우리가 다뤄야 하는 자료 집합의 최대 크기와 같은 크기를 갖는 리스트를 할당하고 시작하는 유혹을 받을 것이지만, 이 유혹은 쓸모없이 메모리를 낭비한다. 게다가, 이 경우에 원소를 추가하는 것은 많은 원소들을 건너야 해서, 추가하는 데에 걸리는 시간이 리스트의 크기에 비례하게 증가하게 한다.

반대로 자료 구조로 나무를 사용한다면, 자료 집합이 정렬되어 있는 경우에 찾기, 추가, 제거의 세가지 연산은 모두 로그 함수의 시간이 걸린다. 이를 위해, 우리는 균형 잡힌 이항 탐색 나무를 사용한다.

예를 들어 연도의 집합, 1819, 1824, 1830, 1834, 1839, 1840, 1841, 1848, 1863, 1867, 1868, 1869, 1880, 1882, 1892 를 생각해보자. 우리는 이 자료 집합을 그림 6.1과 같이 나무로 표현할 것이다.

이 나무는 다음과 같은 흥미로운 성질을 갖는다. 각 노드는 왼쪽 부 나무의 모든 연도보다는 늦고, 오른쪽 부 나무의 모든 연도보다는 빠르다. 나무에서 원하는 연도를 찾기 위해 다음을 수행한다.
먼저 그 나무의 뿌리에 있는 연도와 우리가 찾는 연도를 비교한다.

- 만약 그들이 같다면, 우리는 연도를 찾은 것이다.
- 만약 우리가 찾는 연도가 뿌리보다 이르다면, 왼쪽 부 나무를 찾는다.
- 그렇지 않다면, 오른쪽 부 나무를 찾는다.

그러므로 찾는 과정은 나무의 한쪽 가지만을 추적한다. 즉, 기껏해야 4번만 비교하게 된다.

이런 방식으로 구성된 나무를 이항 탐색 나무라고 부른다. 이항 탐색 나무 중에서, 그림 6.1과 같은 것은 모든 가지가 같은 길이를 가지는 완전 이항 탐색 나무(이 문맥에서, 완전이라는 단어는 말단 노드가 아닌 모든 노드는 정확히 두 개의 자식 노드를 갖는다는 것을 말한다.) 라고 하는 특별한 것이다. 일반적으로 이런 조건을 만족하긴 어렵다. 그렇지만, 가장 긴 가지의 길이가 나무의 전체 크기의 로그 함수로 제한하기 위해서, 앞으로 소개하는 알고리즘은 그 알고리즘으로 만들어진 나무가 "충분히 균형 잡혀"있다는 것을 보장한다.
이 조건은 최악의 경우에도 로그 함수의 탐색 시간이 걸린다는 것을 확실히 한다.

주목할 만한 사실은 추가나 제거 연산이 연산 결과로 균형 잡힌 나무를 유지해야 함에도 불구하고, 최악의 경우 역시 로그 함수의 복잡도를 갖도록 처리할 수 있다는 것이다. 이를 위해, 우리는 Adelson-Velskii와 Landis에 의해 소개된 잘 알려진 방식, AVL 나무를 사용할 것이다. 6.5.2절에서 AVL 나무에 대해 더 자세히 알아볼 것이다.

균형 잡힌 나무로 자료 집합을 관리하는 것은 많은 분야에서 유용하다. "키(key)"를 가진 정보를 담은 사전을 표현하기 위해 균형 잡힌 나무를 사용할 수 있다. 예를 들어, 사전은 심볼 표나 유한한 정의역을 갖는 함수를 구현하기 위해 사용될 지도 모른다. 그러므로 나무의 노드에 나타나는 원소들은 키와 해당하는 정보를 갖는 쌍이다. 예를 들어, 그림 6.1의 나무의 연도와 그 해에 태어난 유명인사의 리스트를 연결한 사전으로 생각할 수 있다.

형식적으로 보면, 키를 배치하는 데 사용한 순서관계는 쌍 (`키`, `정보`) 사이의 선순서 관계(pre-order)를 유도해낸다. 특히 사전을 구현할 수 있도록 가장 일반적인 방식으로 탐색, 추가, 제거 연산을 구현하기 위하여, 선순서 관계로 그것들이 구성될 수 있고, 꼭 순서 관계로 구성될 필요는 없다는 것을 분명히 할 것이다. (둘 사이의 차이를 알기 위해, 6.3절을 참조하시오.)

6.1 이항 나무

이번 장에서 기본이 되는 자료 구조는 이항 나무이다. **이항 나무**는 텅 비어 있거나, 각자가 이항 나무인 왼쪽과 오른쪽의 두개의 자식노드와 정보를 갖는 **뿌리**라고 부르는 기본 노드로 만들어 진다. 이러한 나무를 nML에서 타입 `btree`라고 정의한다.

```
# type 'a btree = Empty | Bin of 'a btree * 'a * 'a btree;;
```

나무 뿌리의 여러가지 항목들을 접근하는 함수가 있으면 편리할 것이다. 이 함수는 나무가 비어있으면 예외를 발생한다.

```
# exception Btree_exc of string;;

# val root = fn
    Empty ⇒ raise (Btree_exc "empty tree")
  | (Bin(_,a,_)) ⇒ a;;
val root:  'a btree -> 'a = <fun>

# val left_son = fn
Empty ⇒ raise (Btree_exc "left_son:  empty tree")
| (Bin(t,_,_)) ⇒ t;;
val left_son:  'a btree -> 'a btree = <fun>

val right_son = fn
Empty ⇒ raise (Btree_exc "right_son:  empty tree")
| (Bin(_,_,t)) ⇒ t;;
val right_son:  'a btree -> 'a btree = <fun>
```

타입 `btree`는 5.1절에서 정의한 타입 `gentree`의 특화된 형태가 아니다. 실제로 타입 `btree`는 한쪽 자식 노드는 비어있고 다른 한쪽은 비어있지 않은 하나의 노드가 왼쪽 자식 노드를 갖는지, 오른쪽 자식 노드를 갖는지를 구분한다.

그림 6.2는 타입 `int btree`의 몇가지 간단한 예제를 그림으로 보여준다.

이번 장의 이후에서 사용되는 나무는 타입 `'a btree`의 개체로 표현될 것이다. 따라서 다음 절에서 정의하는 나무를 추적하는 함수는 이후에 소개되는 특화된 나무에 적용하기 위해 수정할 필요가 없다.

타입 `btree`에 원소를 쉽게 입력하기 위해, 평소와 같이 문법을 분석하는 함수를 필요로 한다. 이런 함수 `btree_of_string`를 정의했다고 가정한다.

```
# btree_of_string;;
val it:  (string -> 'a) -> string -> 'a btree = <fun>
```

이항 나무의 구체적 문법 (conctrete syntax)는 5.1절에서 일반적인 나무를 위해 사용한 것과 본질적으로 같지만, 각 노드의 자식의 숫자가 이제는 정확히 둘이다.

예제를 살펴보자.

```
# btree_of_string int_of_string "2(3,4)";;
val it:  int btree = Bin (Bin (Empty, 3, Empty), 2, Bin (Empty, 4, Empty))
```

이 예제에서, btree_of_string의 인자로 넘긴 분석기 int_of_string은 정수를 인식하고, 타입 int의 값의 형태로 인식한 정수를 결과로써 반환한다.

비슷한 방식으로, 우리는 출력 함수를 정의했다고 가정한다.

```
# print_btree;;
val it:  ('a -> unit) -> 'a btree -> unit = <fun>
```

그렇지만 그림 6.1과 6.2처럼 그림으로 표현하는 것이 편리하기 때문에, 이 함수는 거의 사용하지 않을 것이다.

6.2 나무 추적과 변형성 (Tree Traversals and Morphisms)

일반적인 나무에 대해 살펴본 5.1절에서 만든 것과 비슷한 함수들을 이항 나무에 대해서도 정의할 수 있다. 그렇지만 빈 나무에 해당하는 생성자는 특별한 경우로 다루어야 하기 때문에 차수에서 타입이 조금 다르다.

예를 들어, 이항 연산의 동형성때문에 생성자 Bin과 Empty에 해당하는 두 개의 인자 f와 v가 필요하다.

주어진 타입 'b의 상수 v와 타입 ('b * 'a * 'b) -> 'b의 삼항 연산자 f에 대해서, 다음과 같이 나무의 구조에 대해 재귀적으로 수행함으로써 타입 'b의 값과 타입 'a btree의 모든 나무들을 연관시킬 수 있다.

- val() = v
- val(a(t_1,t_2)) = f (val(t_1),a,val(t_2))

우리는 이런 연산을 **동형성**(homomorphism)이라고 부른다. 함수 btree_hom은 가능한 모든 동형성을 만든다.

```
# val rec btree_hom = fn f v t ⇒
  case t of
    (Bin (t1,a,t2)) ⇒ f (btree_hom f v t1,a,btree_hom f v t2)
  | Empty ⇒ v;;
val btree_hom:  ('a * 'b * 'a -> 'a) -> 'a -> 'b btree -> 'a = <fun>
```

나무의 높이는 가장 긴 가지의 길이이다. 나무의 크기는 이항 노드의 총 개수이다. 두 함수, btree_height와 btree_size는 정수에 대한 동형성으로 볼 수 있다.

```
# fun btree_height t = btree_hom (fn (x1,_,x2) ⇒ 1 + max x1 x2) 0 t;;
val btree_height:  'a btree -> int = <fun>

# fun btree_size t = btree_hom (fn (x1,_,x2) ⇒ 1 + x1 + x2) 0 t;;
val btree_size:  'a btree -> int = <fun>
```

함수 map_tree는 함수 f를 이항 나무의 각 노드에 적용하고 결과 나무를 반환한다. 이 함수는 이항 나무의 동형성이다.

```
# fun map_btree f t =
  btree_hom (fn (t1,a,t2) ⇒ Bin(t1,f a,t2)) Empty t;;
val map_btree:  ('a -> 'b) -> 'a btree -> 'b btree = <fun>
```

함수 mirror_btree는 나무의 거울상을 반환한다. 즉, 각 높이에서 왼쪽과 오른쪽 자식을 바꿔서 만들어 지는 나무를 얻는다. 이 함수도 역시 이항 나무의 동형성이다.

```
# fun mirror_tree t =
  btree_hom (fn (t1,a,t2) ⇒ Bin(t2,a,t1)) Empty t;;
val mirror_tree:  'a btree -> 'a btree = <fun>
```

이 장에서, 우리가 사용하는 이항 나무는 모두 검색 나무, 즉, 가운데 노드부터 추적하는(infix traversal)것이 원소 사이의 순서 관계에 해당하는 성질을 갖는 나무일 것이다. 결론적으로, 왼쪽과 오른쪽의 가운데 노드부터 추적하는 것을 지향하는 함수를 자주 사용할 것이다. 함수 btree_it과 it_btree는 가운데 노드부터 추적하는 것을 만든다. 이것들의 타입은 list_it과 it_list에서 영감을 얻었다.

```
# fun btree_it f t x =
  case t of
    Empty ⇒ x
  | Bin (t1,a,t2) ⇒
          btree_it f t1 (f a (btree_it f t2 x));;
val btree_it:  ('a -> 'b -> 'b) -> 'a btree -> 'b -> 'b = <fun>

# fun it_btree f x t =
  case t of
    Empty ⇒ x
  | Bin (t1,a,t2) ⇒
          it_btree f (f (it_btree f x t1) a) t2;;
val it_btree:  ('a -> 'b -> 'a) -> 'a -> 'b btree -> 'a = <fun>
```

함수 flat_btree는 btree_it을 이용하여 나무에 나타나는 값의 리스트를 만들고, 이것은 왼쪽의 가운데 추적으로 이루어진다.

```
# fun flat_btree t = btree_it (fn x l ⇒ x ::  l) t [];;
val flat_btree:  'a btree -> 'a list = <fun>
```

마지막으로, 우리는 이항 나무에 대해서 기계 중심적인 추적(imperative traversal)을 할 수 있다. 함수 do_btree는 왼쪽의 가운데 추적 순서로 나무의 모든 값에 대해 함수 f를 적용한다.

```
# fun do_btree (f:'a -> unit) = fn
        Empty ⇒ ()
      | (Bin (t1,a,t2)) ⇒
          do_btree f t1; f a; do_btree f t2;;
val do_btree:  ('a -> unit) -> 'a btree -> unit = <fun>
```

6.3 순서과 선순서 관계

순서 관계는 타입 comparison의 값을 이용한 함수들로 표현된다.

```
# type comparison = Smaller | Equiv | Greater;;
```

이렇게 선택함으로써 같은 함수에서 순서관계와 동등관계를 묶을 수 있는 장점이 생긴다. 또한 이것은 선순서를 표현할 수 있게 한다.

타입 bool의 값을 이용한 함수로 주어진 순서 관계로부터 이런 함수를 만들기 위해, 우리는 보조 함수 mk_order을 사용할 것이다.

```
# val mk_order = fn ord x y ⇒
  if ord x y then Smaller else
  if x = y then Equiv
  else Greater;;
val mk_order:  ('a -> 'a -> bool) -> 'a -> 'a -> comparison = <fun>
```

여기에 정수의 순서를 사용하는 몇가지 예제가 있다.

```
# val int_comp =
  mk_order (fn (a:int) (b:int) ⇒ if a > b then true else false);;
val int_comp:  int -> int -> comparison = <fun>

# int_comp 2 3;;
val it:  comparison = Greater

# int_comp 2 2;;
val it:  comparison = Equiv
```

선순서는 재귀적이고 (reflexive), 이행적이나 (transitive) 반드시 비대칭적 (antisymmetric)일 필요는 없는 관계이다. 모든 선순서 $\leq$에는 관련되어 있는 동등 관계 $\equiv$가 존재한다. 이 동등 관계는 다음과 같이 정의된다.

$$x \equiv y \longleftrightarrow x \leq y \; and \; y \leq x$$

여기서 $\equiv$는 동등 관계이고 $\leq$는 순서 관계이다.

하나의 선순서와 이 선순서와 관련된 동등관계로 만들어진 쌍은 역시 타입 comparison의 값을 이용한 함수에 의해 표현된다. 다음을 이용해서 이것을 만들 수 있다.

```
# val mk_preorder = fn (lt,eq) x y ⇒
  if lt x y then Smaller else
  if eq x y then Equiv
  else Greater;;
val mk_preorder:  ('a -> 'b -> bool) * ('a -> 'b -> bool) -> 'a -> 'b ->
                comparison
              = <fun>
```

순서나 선순서를 이용한 계산을 하는 동안에는 최소나 최대 값이 있다고 가정하는 것이 편리하다. 우리는 다음의 타입을 사용할 것이다.

```
# type 'a minmax = Min | Plain of 'a | Max;;
```

그리고 순서 관계를 다음과 같이 확장한다.

```
# val extent_order = fn ord x y ⇒
  case (x,y) of
    ((Min,Min)|(Max,Max)) ⇒ Equiv
  | ((Min,_)|(_,Max)) ⇒ Smaller
  | ((Max,_)|(_,Min)) ⇒ Greater
  | (Plain x, Plain y) ⇒ ord x y;;
val extent_order:  ('a -> 'b -> comparison) -> 'a minmax -> 'b minmax ->
                comparison
              = <fun>
```

6.4 이항 탐색 나무

이항 탐색 나무는 값들이 정렬된 자료 집합에 속해 있는 이항 나무이다. 각 노드의 값은 왼쪽 자식에 있는 값보다는 크고, 오른족 자식에 있는 모든 것보다는 작다. 이 성질 때문에, 나무에서 값을 찾을 때 오직 그 값이 속한 하나의 가지만을 탐색하는 것이 가능하다. 사실상, 나무 t=a(t_1,t_2)에 값 x가 있는지 알기 위해서 해야할 모든 것은 x와 a를 비교하는 것에서부터 시작한다.

- 만약 x=a이라면, x는 t에 존재한다.
- 만약 x<a이라면, x가 t_1에 존재하는 경우에 x는 t에 존재한다.
- 만약 x>a이라면, x가 t_2에 존재하는 경우에 x는 t에 존재한다.

다시 말해 이항 탐색 나무는 정렬된 자료 집합을 표현할 때, 최악의 경우에 어떤 원소가 나무에 속해 있는지 확인하는 데에 걸리는 시간이 나무의 가장 긴 가지에 비례하는 방식을 이용한다.

주어진 정렬된 자료 집합에 대해서 이항 탐색 나무는 많이 존재할 수 있다. 예를 들어, 다음과 같은 정수 집합 {1, 2, 4, 6, 8, 10, 12, 15, 17, 20, 21}을 표현하는 두가지 나무가 그림 6.3에 나와있다.

이항 탐색 나무는 타입 btree의 객체로서 구현된다. 이 타입은 임의의 이항 나무와 이항 **탐색** 나무를 구분하지 않는다. 이항 탐색 나무를 만들고, 추적하고, 수정하는 모든 함수들은 인자로 순서 관계를 받아들이고, 그들이 작업하는 나무가 사실 이 순서 관계에 대한 이항 탐색 나무라고 가정한다.

이항 탐색 나무가 정렬된 집합이나 사전과 같은 여러가지 특화된 자료 구조를 구현하는 경우, 우리는 해당하는 타입을 사용하는 순서 관계를 통합할 것이다.

예를 들어, 이항 나무가 주어진 관계에 대해 실제로 이항 탐색 나무인지 확인하는 함수는 다음과 같이 정의할 수 있다.

```
# val is_bst = fn order ⇒
  let val ext_order = extend_order order in
      let val rec check_bst = fn (a,b) t ⇒
          case t of
            Empty ⇒ true
          | (Bin (t1,x,t2)) ⇒
              let val x = Plain x
              in  not (ext_order a x = Greater) &&
                  not (ext_order x b = Greater) &&
                  check_bst (a,x) t1 &&
                  check_bst (x,b) t2
              end
      in check_bst (Min,Max)
      end
  end;;
val is_bst:  ('a -> 'a -> comparison) -> 'a btree -> bool = <fun>
```

6.4.1 원소 찾기

함수 search_bst는 선순서 order에 대해 e와 똑같은 원소를 찾기 위해 이항 탐색 나무를 추적한다. 이항 탐색 나무에서 원소를 찾기 위해서, 원소가 위치한 나무의 가지만을 추적해야 한다. 이 함수로부터 결과를 얻기 위해, 우리는 인자 answer을 뿌리의 부나무에 적용한다. 여기서 부나무는 우리가 찾고 있는 원소가 있는 부나무이다.

```
# exception Bst_search_exc of string;;

# fun search_bst order answer e =
  let val rec search = fn
        Empty ⇒ raise (Bst_search_exc "search_bst")
      | Bin(t1,x,t2) ⇒
          case order e x of
            Equiv ⇒ answer Bin(t1,x,t2)
          | Smaller ⇒ search t1
          | Greater ⇒ search t2
  in search end;;
val search_bst:  ('a -> 'b -> comparison) -> ('b btree -> 'c) -> 'a -> 'b
                 btree -> 'c
               = <fun>
```

함수 find_bst는 약간 변형하여 (정보가 뿌리인 부나무 대신에) 정보 자체를 반환한다.

```
# fun find_bst order = search_bst order root;;
val find_bst:  ('a -> 'b -> comparison) -> 'a -> 'b btree -> 'b = <fun>
```

또한 우리는 때때로 bool 값을 반환하는 함수 belongs_to_bst를 사용할 것이다.

```
# fun belongs_to_bst order e t =
  find_bst order e t; true
  handle Bst_search_exc _ ⇒ false;;
val belongs_to_bst:  ('a -> 'b -> comparison) -> 'a -> 'b btree -> bool
                  = <fun>
```

우리가 선순서를 사용하는 경우에, 나무의 한 원소를 선순서의 관점에서 볼 때 동일한 다른 원소로 바꾸는 것이 필요할 수 있다. 함수 change_bst는 find_bst처럼 나무를 탐색하지만, 이것은 발견한 원소를 반환하지 않는다. 대신에, 이것은 함수 modify에 의해 수정된 원소를 가지고 있는 새로운 나무를 반환한다. 이 함수가 선순서에 관련된 동일성을 유지하는가는 전적으로 프로그래머의 책임이다. 이 연산은 발견한 원소(그리고 오직 이 원소들)만 나무의 뿌리와 수정된 원소 사이에 복사한다.

```
# fun change_bst order modify e =
  let val rec change = fn
        Empty ⇒ raise (Bst_search_exc "change_bst")
      | Bin(t1,x,t2) ⇒
          case (order e x) of
            Equiv ⇒ Bin(t1,modify x,t2)
          | Smaller ⇒ Bin(change t1,x,t2)
          | Greater ⇒ Bin(t1,x,change t2)
  in change end;;
val change_bst:  ('a -> 'b -> comparison) -> ('b -> 'b) -> 'a -> 'b btree ->
                    'b btree
                 = <fun>
```

6.4.2 원소 추가하기

이항 탐색 나무에 원소 e를 추가하는 데에는 종단 노드에 추가하는 것과 뿌리에 추가하는 것의 두가지 방법이 존재한다. 간단한 것은 종단 노드에 추가하는 것이다. 이를 위해 추가할 원소와 교체될 수 있는 비어있는 부나무를 찾는다. 이런 부나무를 찾기 위해서, 원소 e와 뿌리를 비교하고 계속 탐색해 나가면 된다. 즉, e가 뿌리보다 작은 경우 왼쪽 부 나무를 찾고, e가 뿌리보다 큰 경우에는 오른쪽 부 나무를 찾는다. 만약 원소가 나무의 뿌리와 같다면, 우리가 해야하는 일은 오래된 뿌리를 유지하기를 원하는지, 새로운 뿌리로 바꾸길 원하는지, 또는 연산을 거부하기를 원하는 지에 달려있다.

함수 add_bottom_to_bst는 우리가 방금 간략히 살펴본 알고리즘을 구현한다. 우리가 더하고자 하는 것과 동일한 원소가 있는 경우, 우리는 오래된 것과 새로운 것중에서 하나를 선택하기 위해 option 인자를 사용할 것이다.

- 새로운 것으로 오래된 것을 바꾸길 원한다면, 우리는 옵션으로 함수 fun x y ⇒ y를 사용한다.

- 만약 오래된 것을 유지하고 싶다면, `fun x y ⇒ y`를 사용한다.
- 만약 이런 종류의 충돌이 발생하는 경우에, 예를 들어 함수 `fun x y ⇒ raise (Bst_search_exc "clash")`를 사용할 수 있다.

```
# fun add_bottom_to_bst option order t e =
  let val rec add = fn
        Empty ⇒ Bin(Empty,e,Empty)
      | Bin(t1,x,t2) ⇒
          case (order e x) of
            Equiv ⇒ Bin(t1,option e x,t2)
          | Smaller ⇒ Bin(add t1,x,t2)
          | Greater ⇒ Bin(t1,x,add t2)
  in add t end;;
val add_bottom_to_bst:  ('a -> 'a -> 'a) -> ('a -> 'a -> comparison) -> 'a
                        btree -> 'a -> 'a btree
                      = <fun>
```

함수 `add_list_bottom_to_bst`는 원소 리스트를 이항 탐색 나무에 추가한다.

```
# fun add_list_bottom_to_bst option order =
  fold_left (add_bottom_to_bst option order);;
val add_list_bottom_to_bst:  ('a -> 'a -> 'a) -> ('a -> 'a -> comparison) ->
                         'a btree -> 'a list -> 'a btree
                       = <fun>
```

함수 `mk_bst`는 원소 리스트로부터 이항 탐색 나무를 만든다.

```
# val mk_bst = fn option order ⇒
  add_list_bottom_to_bst option order Empty;;
val mk_bst:  ('a -> 'a -> 'a) -> ('a -> 'a -> comparison) -> 'a list -> 'a
           btree
         = <fun>
```

그림 6.4은 표현식 `mk_bst (fun x y ⇒ x) int_comp [10,15,12,4,6,21,8,1,17,2])`를 값매김 했을 때 순서대로 나무가 만들어지는 것을 보여준다.

6.4.3 뿌리에 원소 추가하기

이항 탐색 나무의 뿌리에 원소를 추가하는 것도 물론 가능하다. 이 기법은 우리가 종종 최근에 추가된 원소를 찾을 필요가 있을 때 유용하다. 만약 그러한 원소들이 뿌리 근처에 있다면, 좀더 빠르게 찾을 수 있다.

이를 위해, 주어진 나무 `t`와 추가할 원소 `e`에 대해서 원래 나무를 두개의 이항 탐색 나무 t_1과 t_2로 "자른다". 여기서 `t1`은 `e`보다 작은 원소로 구성되어 있고 `t2`는 `e`보다 큰 원소들로 구성되어 있다. 원래 나무 `t`가 `e`와 동일한 원소를 포함하지 않거나 `e`로 그것을 바꾸기 원할 경우에는, 원소를 추가하기 위해 나

무 Bin(t_1,e,t_2)를 만들어야 한다. 만약 예전의 동일한 원소를 유지하고 싶을 때에는, 자르는 과정에서 원소와 함께 두개의 부 나무도 반환해야 한다. 다음 알고리즘은 이 가능성을 고려한다.

Empty 나무를 원소 e에 대해 자르기 위해서, 삼중항 (Empty,e,Empty)를 반환하라.

나무 t=a(t_1,t_2)를 값 e를 두고 두 부분으로 자르기 위해서, 우리는 e와 a를 비교한다.

- 만약 e=a라면, 삼중항 (t_1,a,t_2)를 반환하라.
- 만약 e<a라면, 나무 t_1를 원소 e에 대해서 자른다. 이를 통해 삼중항 (t'_1,e′,t'_2)를 얻는다. 삼중항 (t'_1,e′,a(t'_2,t_2))를 반환하라.
- 만약 e>a라면, 대칭적으로 위의 연산을 수행하라.

함수 cut_bst는 이 알고리즘을 구현한다.

```
# val rec cut_bst = fn order e ⇒
  let val rec cut = fn
        Empty ⇒ (Empty,e,Empty)
      | Bin(t1,a,t2) ⇒
          case (order e a) of
            Smaller ⇒ let val (t,e',t') = cut t1
                      in (t,e',Bin(t',a,t2)) end
          | Equiv ⇒ (t1,a,t2)
          | Greater ⇒ let val (t,e',t') = cut t2
                      in (Bin(t1,a,t),e',t') end
  in cut end;;
val cut_bst:  ('a -> 'a -> comparison) -> 'a -> 'a btree -> 'a btree * 'a * 'a
                btree
              = <fun>
```

그림 6.5는 자르기의 예를 보여준다.

원소 e를 이항 탐색 나무 t에 추가하기 위해서, t를 잘라 삼중항 (t_1,e′,t_2)를 얻는다. 그리고나서 만약 새로운 원소를 원하면 나무 t=e(t_1,t_2)를 만들고, 그렇지 않다면 t=e′(t_1,t_2)을 만든다. 나무 t가 e와 동일한 원소를 갖지 않는 경우에는, 부나무 Empty에서 자르기가 끝나게 되고 따라서 e′=e임을 주목하라.

함수 add_root_to_bst는 이 알고리즘을 구현한다.

```
# val add_root_to_bst = fn option order e t ⇒
  let val (t1,e',t2) = cut_bst order e t
  in Bin(t1,option e e',t2) end;;
val add_root_to_bst:  ('a -> 'a -> 'a) -> ('a -> 'a -> comparison) -> 'a -> 'a
                    btree -> 'a btree
                  = <fun>
```

함수 mk_bst2는 원소 리스트로부터 뿌리에 하나하나 더해가는 방식으로 이항 탐색 나무를 만든다.

```
# val mk_bst2 = fn option order l ⇒
  fold_right (add_root_to_bst option order) l Empty;;
val mk_bst2:  ('a -> 'a -> 'a) -> ('a -> 'a -> comparison) -> 'a list -> 'a
                  btree
                = <fun>
```

그림 6.6은 표현식 `mk_bst2 (fun x y ⇒ x) int_comp [10,15,12,4,6,21,8,1,17,2]`를 값매김 할때 연속적으로 나무가 만들어지는 것을 보여준다.

6.4.4 원소 제거하기

우리가 이항 탐색 나무의 뿌리에 위치한 원소를 제거할 때, 만약 적어도 둘중에 하나의 자식이 빈 나무라면, 뿌리를 제거하는 것은 순간적으로 해낼 수 있다. (만약 하나의 비지 않은 자식 나무가 존재하면) 이 비어있지 않는 자식 나무로 원래 나무를 바꾸거나, 그렇지 않은 경우에는 빈 나무로 원래 나무를 바꾸는 것으로 충분하다.

이에 반해, 고려대상인 나무의 자식 나무가 둘다 비어있지 않다면, 뿌리에 있는 원소들은 교체되어야 한다. 물론 여기에 두가지 대안이 있다. 왼쪽 자식들 중에서 가장 큰 원소나 오른쪽 자식들 중에서 가장 작은 원소이다.

여기서 우리가 제거할 원소를 대신하기 위해 왼쪽 자식 중에서 가장 큰 원소를 사용할 것이다. 다시 말해 왼쪽 자식의 가장 오른쪽 가지 끝에 있는 원소를 선택한다. 이를 얻기 위해, 보조 함수 `remove_biggest`를 이용할 것이다. 이 함수가 이항 탐색 나무에 적용되면, 가장 큰 원소와 가장 큰 원소가 없어진 원래 나무의 쌍을 반환한다.

```
# exception Bst_exc of string;;
exception Bst_exc of string

# val rec remove_biggest = fn
    (Bin(t1,a,Empty)) ⇒ (a,t1)
  | (Bin(t1,a,t2)) ⇒
        let val (a',t') = remove_biggest t2 in (a',Bin(t1,a,t')) end
  | Empty ⇒ raise (Bst_exc "remove_biggest:  tree is empty");;
val remove_biggest:  'a btree -> 'a * 'a btree = <fun>
```

함수 `rem_root_from_bst`는 `remove_biggest`를 이용하여 뿌리를 왼쪽 자식 중 가장 큰 원소로 바꾼다. 함수 `rem_from_bst`는 제거할 원소를 찾기 위해 나무를 탐색하고 함수 `rem_root_from_bst`를 부른다.

```
# val rec rem_root_from_bst = fn
    Empty ⇒ raise (Bst_exc "rem_root_from_bst:  tree is empty")
  | (Bin(Empty,a,t2)) ⇒ t2
  | (Bin(t1,_,t2)) ⇒
            let val (a',t') = remove_biggest t1
            in Bin(t',a',t2) end;;
val rem_root_from_bst:  'a btree -> 'a btree = <fun>
```

```
# val rec rem_from_bst = fn order e ⇒
  let val rec rem = fn
        Empty ⇒ raise (Bst_search_exc "rem_from_bst")
      | (Bin(t1,a,t2)) ⇒
          case (order e a) of
            Equiv ⇒ rem_root_from_bst (Bin(t1,a,t2))
          | Smaller ⇒ Bin(rem t1,a,t2)
          | Greater ⇒ Bin(t1,a,rem t2)
  in rem end;;
val rem_from_bst:  ('a -> 'b -> comparison) -> 'a -> 'b btree -> 'b btree
                 = <fun>
```

함수 rem_list_from_bst는 이항 나무에서 원소의 리스트를 제거한다.

```
# val rem_list_from_bst = fn order ⇒
  fold_right (rem_from_bst order);;
val rem_list_from_bst:  ('a -> 'b -> comparison) -> 'a list -> 'b btree -> 'b
                  btree
                 = <fun>
```

그림 6.7은 이항 탐색 나무에서 원소 10, 15, 그리고 6을 차례대로 제거하는 것을 보여준다.

6.5 균형 잡힌 나무

이항 탐색 나무에서 원소를 찾는 데에 걸리는 비용은 나무의 뿌리로부터 우리가 찾는 원소 사이의 거리에 비례한다. 따라서 탐색할 때 걸리는 최대 비용은 가장 긴 가지의 길이에 비례하고 평균 비용은 뿌리와 노드 사이의 평균 거리에 비례한다.

그러므로 크기 n인 이항 나무에서, 평균이나 최악의 경우를 생각해 볼 때 가장 좋은 성능을 보이는 것은 **완전히 균형 잡힌 나무**, 즉 높이가 $\log_2(n)$인 것이다. 평균 탐색 시간을 최소화 하기 위해 이상적인 것은 완전히 균형 잡힌 나무를 사용하는 것뿐이다.

이를 위해, 우리는 원소를 추가하거나 제거할 때 완전히 균형잡힌 특성을 유지할 수 있어야만 하며, 이것을 상식적인 비용 한도 내에서 해내야 한다. 다음의 예제에서 금방 알 수 있듯이, 이것은 가능하지 않다. 그림 6.8의 왼쪽 나무는 완전히 균형 잡힌 이항 탐색 나무이다. 만약 여기에 값 1을 추가한다면, 가능한 완전히 균형잡힌 이항 탐색 나무의 형상은 같은 그림의 오른족 나무와 같을 것이다. 여기서 두 나무는 모든 노드가 다르다는 점에 주목하라. 게다가 두 나무 구조는 부모 자식 사이의 관계가 완전히 다르다. 요약해서 말하면, 오른쪽에 있는 나무를 만들기 위해서는 원래 나무를 완전히 재구성해야 한다.

임의의 큰 크기의 나무 구조에 대해 이런 종류의 예를 들 수 있다. 따라서 만약 나무 구조를 완전히 균형잡히도록 유지하게 만들고자 한다면, 원소를 추가하고 나서 이항 탐색 나무를 재 구성하는 비용은 최악의 경우에 나무의 크기에 비례한다.

추가와 제거 과정에서 최악의 경우에 로그에 비례하는 비용을 사용하도록 하려면, "균형잡힌" 나무구조의 개념을 조금 약화시켜야 한다. 6.5.2절에서 소개되는 AVL 나무구조는 이 문제에 대해 하나의 해결방법을 제시한다. 이런 나무 구조들에서는 **회전**이라 알려진 연산을 이용해 나무의 균형을 잡는다.

6.5.1 회전

회전은 이항 탐색 나무의 구조를 유지하기 위해 나무를 재구성하는 방법이다. 그러나 이것은 나무의 균형을 고칠 수 있도록 한다. 두가지 종류의 회전이 존재한다. 그림 6.10에 있는 것이 **오른쪽 회전**이고, 그림 6.10에 있는 것이 **왼쪽 회전**이다.

이 연산을 다음과 같이 프로그램한다.

```
# val rot_right = fn
    Bin(Bin(u,p,v),q,w) ⇒ Bin(u,p,Bin(v,q,w))
  | _ ⇒ raise (Btree_exc "rot_right");;
val rot_right:  'a btree -> 'a btree = <fun>

# val rot_left = fn
    Bin(u,p,Bin(v,q,w)) ⇒ Bin(Bin(u,p,v),q,w)
  | _ ⇒ raise (Btree_exc "rot_left");;
val rot_left:  'a btree -> 'a btree = <fun>
```

우리는 이 연산의 조합을 사용하여 그림 6.11과 같은 **왼쪽-오른쪽 회전**이나, 그림 6.12와 같은 **오른쪽-왼쪽 회전**을 사용할 것이다. 이들을 다음과 같이 프로그램할 수 있다.

```
# val rot_left_right = fn
    Bin(Bin(t,p,Bin(u,q,v)),r,w) ⇒ Bin(Bin(t,p,u),q,Bin(v,r,w))
  | _ ⇒ raise (Btree_exc "rot_left_right");;
val rot_left_right:  'a btree -> 'a btree = <fun>

# val rot_right_left = fn
    Bin(t,r,Bin(Bin(u,q,v),p,w)) ⇒ Bin(Bin(t,r,u),q,Bin(v,p,w))
  | _ ⇒ raise (Btree_exc "rot_right_left");;
val rot_right_left:  'a btree -> 'a btree = <fun>
```

이러한 회전 연산이 이항 탐색 나무의 구조를 유지한다는 것을 보이는 것은 간단하다.

6.5.2 AVL 나무

이제 함수 δ를 정의한다. 이것은 모든 나무의 뿌리에 대한 "불균형" 정도를 말한다.

$$\delta(a(t_1, t_2)) = \text{height}(t_1)\text{-height}(t_2)$$

만약 각각의 부 나무들에 대해서 함수 δ의 절대값이 기껏해야 1이라면, 이항 나무를 **H-균형**이라고 말한다. **AVL 나무**는 H-균형 이항 탐색 나무이다.

그림 6.13은 왼쪽으로 기울어져 있는 노드에는 왼쪽에 검은 점이 찍혀 있고, 오른쪽으로 기울어져 있는 노드에는 오른쪽에 검은 점이 찍혀 있는 AVL 나무이다.

크기가 n인 H-균형 나무는 높이가 n의 로그함수로 제한된다. 이 성질을 간단하게 실험할 수 있다. 이를 위해, 모든 높이 h에 대해서 우리는 최소한의 정점을 갖는 해당하는 높이의 나무를 만든다.

- 높이가 1이면, 크기 1인 나무를 만든다.
- 높이 h인 나무를 만들기 위해서, 뿌리의 자식중 하나는 필수적으로 높이가 h-1이어야 한다. 그렇지 않다면, 즉 그런 자식이 없다면 나무는 높이가 h가 될 수 없다. 추가로, 또다른 자식중 하나는 h-2의 높이여야 한다. 만약 높이가 h-2보다 작다면, 뿌리에 대해서 나무는 H-균형이 아닐 것이다. 만약 높이가 h-1이라면, 나무는 최소의 크기가 아닐 것이다.

만약 우리가 높이 h의 H-균형 나무의 최소 크기를 $N(h)$라고 한다면, 우리는 다음과 같이 이 함수의 재귀적인 정의를 얻는다.

- $N(0)$=0
- $N(1)$=1
- $N(h)$=$N(h$-1)+$N(h$-2)+1

$N\prime(h) = N(h) + 1$로 하면, 다음과 같은 수식을 얻는다.

$$N\prime(h) = N\prime(h-1) + N\prime(h-2)$$

의심할 여지 없이 당신은 이 수식이 우리의 친구, 피보나치 수열임을 발견할 것이다. 이 숫자들은 지수함수적으로 증가하기 때문에, 역함수는 로그 함수로 증가한다. 사실, 크기가 n인 H-균형잡힌 나무의 높이가 $1.44 \log_2(n+2)$로 제한된다는 것을 보일 수 있다.

AVL 나무구조를 구현하기 위해서, 세가지의 경우인 $\delta = -1$, $\delta = 0$, $\delta = +1$에 해당하는 세개의 생성 타입을 만드는 것이 가장 효율적이다. 그렇지만 이 타입은 타입 `btree`와는 다를 것이다. 그리고 이 차이 때문에 `btree`에 대해서 정의한 함수를 재사용할 수 없을 것이다. 결론적으로, 각각의 노드에 타입 `balance`의 정보를 가지고 있는 이항 나무로 AVL 나무 구조를 구현할 것이다. 타입 `balance`는 그 노드에서 나무가 균형 잡혀 있는지 ($\delta = 0$), 왼쪽으로 기울었는지 ($\delta = 1$), 아니면 오른쪽으로 기울어 졌는지 ($\delta = -1$)를 표시하는 세개의 값을 갖는다.

```
# type balance = Left | Balanced | Right;;
```

사용되는 타입은 다음과 같이 짧게 쓸 수 있다.

```
# type 'a avltree = ('a * balance) btree;;
```

이항 탐색 나무에서 그랬던 것처럼, AVL 나무구조에 대해 특별한 예외를 사용할 것이다.

```
# exception Avl_exc of string;;
exception Avl_exc of string

# exception Avl_search_exc of string;;
exception Avl_search_exc of string
```

AVL 나무의 많은 함수들은 이항 탐색 나무로부터 직접 상속받는다. 그러나 비교 함수는 각 노드의 유용한 정보에 영향을 미쳐야 하고 균형에 관한 자세한 사항은 내버려 두어야 한다.

예를 들어 함수 belongs_to_avl은 값이 AVL 나무 구조에 속하는지 검사한다. 이것은 비교 함수를 적절히 수정한 belongs_to_bst를 이용한다.

```
# val belongs_to_avl = fn order ⇒
  belongs_to_bst (fn x y ⇒ order x (fst y));;
val belongs_to_avl:  ('a -> 'b -> comparison) -> 'a -> ('b * 'c) btree -> bool
                  = <fun>
```

이와 비슷하게, 다음의 함수들이 정의되었다고 가정한다.

```
flat_avl :  ('a * balace) btree -> 'a list
map_avl :  ('a -> 'b) -> ('a * balance) btree -> ('b * balance) btree
do_avl :  ('a -> unit) -> ('a * balace) btree -> unit
```

추가로, AVL 나무구조에 대한 함수들이 동작중에 예외를 발생시킬 때, AVL 나무 구조에 특화되어있는 예외를 사용한다는 것을 확인해야 한다.

```
# val find_avl = fn order e (t:('a*alance) btree) ⇒
  fst(find_bst (fn x y ⇒ order x (fst y)) e t)
  handle Bst_search_exc _ ⇒ raise(Avl_search_exc "find_avl");;
val find_avl:  ('a -> 'b -> comparison) -> 'a -> ('b * balance) btree -> 'b
                  = <fun>
```

반대로, 균형 정보에 의존한 모든 함수들은 특별히 정의되어야 한다. 예를 들어, AVL 나무가 정확히 균형잡혀 있는지 확인하는 함수는 다음과 같다.

```
# val h_balanced = fn (t:('a*balance) btree) ⇒
  let val rec correct_balance = fn
    Empty ⇒ 0
  | (Bin(t1,(x,b),t2))
      ⇒ let val n1 = correct_balance t1
                and n2 = correct_balance t2
        in
             if (b=Balanced && n1=n2) then n1+1 else
             if (b=Left && n1=n2+1) then n1+1 else
             if (b=Right && n2=n1+1) then n2+1
```

```
              else raise (Avl_exc "not avl")
         end
  in
      correct_balance t; true
      handle Avl_exc _ ⇒ false
  end;;
val h_balanced:  ('a * balance) btree -> bool = <fun>

# val is_avl = fn order (t:('a*balance) btree) ⇒
  h_balanced t && is_bst (fn x y ⇒ order (fst x) (fst y)) t;;
val is_avl:  ('a -> 'a -> comparison) -> ('a * balance) btree -> bool = <fun>
```

다음은 AVL 나무의 거울 상을 만들어 내는 함수이다.

```
# val mirror_avl = fn t ⇒
  btree_hom
      (fn (t1,(x,b),t2)
          ⇒ let val b' = case b of
            Left ⇒ Right
          | Balanced ⇒ Balanced
          | Right ⇒ Left
          in Bin(t2,(x,b'),t1)
          end)
      Empty t;;
val mirror_avl:  ('a * balance) btree -> ('a * balance) btree = <fun>
```

AVL 나무에 원소를 추가하는 것이나 제거하는 것은 균형에 영향을 미친다. 예를 들어, 그림 6.13에 보이는 AVL 나무의 종단 노드에 값 10을 추가하기 위하여 이항 탐색 나무의 종단 노드에 값을 추가하는 알고리즘을 사용한다면, 결과는 더이상 AVL 나무가 아닐 것이다. 노드 11, 4, 6은 각각 δ 값이 2, -2, -2이다. 또한 뿌리에 10을 더하는 경우에도 균형에 영향을 미친다. 결론적으로, 이항 탐색 나무를 위해 정의한 원소를 추가하는 함수는 더이상 사용할 수 없다.

원소를 제거하는 것도 역시 AVL 나무의 균형을 무너뜨리는 작용을 한다. 만약 그림 6.13의 나무에서 원소 12와 17을 제거한다면, 뿌리는 균형잡지 못하게 될 것이다.

결론적으로, 이항 탐색 나무에서 사용한 원소 추가 및 제거 함수는 더이상 사용할 수 없다.

다행히, "회전" 함수를 사용하여 이 연산에 의해 발생한 문제점을 수정할 수 있다.

6.5.3 AVL 나무에 원소 추가하기

우선 6.4.2절에서 살펴본 알고리즘을 가지고 시작하자. t=a(t_1,t_2)인 AVL 나무가 있고, 여기에 원소 e를 추가하려고 한다. e와 a를 비교하면서 시작하자.

만약 e=a이면, 선택한 옵션을 생각해보고, 나무 t 그자체나 나무 e(t_1,t_2)를 반환한다.

만약 e<a이면, 부 나무 t_1에 원소를 추가한다. 이것은 나무 t'_1을 반환한다. 이제 몇가지 경우를 생각해 보자.

- 만약 height(t'$_1$)=height(t_1)+1이고 δ(t)=0 또는 δ(t)=-1이라면, 원소를 추가한 결과는 여전히 a(t'$_1$,t_2)이다. 그렇지만, 결과로 얻는 나무의 뿌리의 균형정보를 정확히 수정해야 한다.
- 만약 height(t'$_1$)=height(t_1)+1이고 δ(t)=1이면, 나무 t'=a(t'$_1$,t_2)는 δ(t')=2이기 문에 AVL나무가 아니다. 따라서 회전을 이용해 다시 균형을 맞추어야 한다. 이를 위해, 나무 t'$_1$의 균형을 생각해야 한다. 이 나무는 균형이 맞을리가 없다. 만약 그렇다면 이것은 t_1과 같은 높이를 가졌을 것이다. 따라서 다음의 두가지 경우만이 존재한다.
  - δ(t_1)=1. t'은 오른쪽 회전으로 다시 균형을 맞출 수 있다. 따라서 원소를 추가한 결과는 rot_right(a(t'$_1$,t_2))이다.
  - δ(t_1)=-1. t'=a(t'$_1$,t_2)는 전체 나무를 오른쪽 회전한 후에 t'$_1$을 왼쪽 회전함으로써 다시 균형을 맞출 수 있다. 원소를 추가한 결과는 rot_right(a(rot_left(t'$_1$),t_2))이다.
- 다른 경우는 이상의 것과 비슷하게 할 수 있다.

만약 원소를 추가한 후에 부 나무가 다시 균형을 맞추어야 한다면, 다시 균형 잡은 후의 높이는 원소를 추가하기 이전과 같다. 이것은 이를 포함하는 어떤 부 나무도 다시 균형 잡을 필요가 없다는 관찰에 근거를 두고 있다. 그러므로 추가는 기껏해야 한번의 회전(일단이나 이단회전)을 하게 된다.

원소를 추가한 후에 AVL 나무를 다시 균형 맞추는 동안에 무슨 일이 벌어지는지, 예제를 통해 살펴보면 이해하기 쉬울 것이다. 그림 6.15와 6.16은 원소 10, 7, 2, 5, 4을 차례대로 추가하면서 만들어지는 AVL 나무에서 벌어지는 회전을 보여준다. 만약 6.4.2절에 있는 알고리즘으로 이항 탐색 나무처럼 원소를 추가했다면 결과는 그림 6.15에 있는 변형된 나무일 것이다.

원소 10, 7, 2를 추가하면 그림 6.15에서와 같이 왼쪽은 균형이 맞지 않은 나무이다. 이것은 오른쪽 회전을 통해 조정할 수 있다.

원소 5와 3 차례로 추가할 때, 그림 6.16의 왼쪽에 있는 것과 같이 균형이 맞지 않는 나무를 다시 만나게 된다. 이것은 이단 회전(오른쪽-왼쪽)으로 조정할 수 있다.

이런 개념을 염두에 두고, 이제 원소를 추가하는 함수를 작성해 보자. 우선, 균형에 대한 정보를 수정하는 것을 추가한 새로운 회전 연산을 살펴본다. 여기서는 오른쪽 회전의 정의를 공부한다. 다른 것들도 비슷한 방식으로 재정의 된다.

```
# exception Avl_rotation_exc of string;;

# val rot_right = fn
  (Bin(Bin(u,(p,b),v),(q,_),w)) ⇒
      (case b of
         Balanced ⇒ Bin(u,(p,Right),Bin(v,(q,Left),w))
       | Left ⇒ Bin(u,(p,Balanced),Bin(v,(q,Balanced),w))
       | Right ⇒ raise (Avl_rotation_exc "rot_right"))
  | _ ⇒ raise (Avl_rotation_exc "rot_right");;
val rot_right:  ('a * balance) btree -> ('a * balance) btree = <fun>
```

어떤 나무의 균형을 다시 맞춰주어야 하는지 결정하기 위해서, 그리고 뿌리로부터 노드 추가 지점사이의 가지들(그 외의 것들은 상관없다.)의 균형 정보를 갱신하기 위해서, 원소를 추가할 때, 재귀적인 용법을 통해 원소를 추가한 결과는 물론이고 나무의 높이가 변경되었는지에 대한 정보와 만약 변경되었다면 어떤 자식이 변경했는지에 대한 정보도 얻어야 한다. 이 정보는 타입 `avl_add_info`의 값이다.

```
# type avl_add_info = No_inc | Incleft | Incright;;
```

`Incleft`는 높이가 왼쪽에서 증가되었을 때 사용되고, `Incright`는 높이가 오른쪽에서 증가되었을 때 사용한다. 만약 원래 나무가 비어있었다면 아무쪽이나 반환해도 상관없는데, 이는 이후의 계산과정에 어떤 영향도 미치지 못하기 때문이다. `No_inc`는 높이가 변경되지 않은 경우에 사용한다.

```
# val rec add_to_avl = fn option order t e ⇒
  let val rec add = fn
    Empty ⇒
          (Bin(Empty,(e,Balanced),Empty),Incleft)
  | (Bin(t1,(x,b),t2)) ⇒
          (case (order e x, b) of
            (Equiv,_) ⇒ (Bin(t1,(option e x,b),t2),No_inc)
          | (Smaller,Balanced) ⇒
              let val (t,m) = add t1
              in if m=No_inc then (Bin(t,(x,Balanced),t2),No_inc)
                 else (Bin(t,(x,Left),t2),Incleft)
              end
          | (Greater,Balanced) ⇒
              let val (t,m) = add t2
              in if m=No_inc then (Bin(t1,(x,Balanced),t),No_inc)
                 else (Bin(t1,(x,Right),t),Incright)
              end
          | (Greater,Left) ⇒
              let val (t,m) = add t2
              in if m=No_inc then (Bin(t1,(x,Left),t),No_inc)
                 else (Bin(t1,(x,Balanced),t),No_inc)
              end
          | (Smaller,Left) ⇒
              let val (t,m) = add t1
              in case m of
                    No_inc ⇒ (Bin(t,(x,Left),t2),No_inc)
                  | Incleft ⇒ (rot_right(Bin(t,(x,Balanced),t2)),No_inc)
                  | Incright ⇒ (rot_left_right(Bin(t,(x,Balanced),t2)),No_inc)
              end
          | (Smaller,Right) ⇒
              let val (t,m) = add t1
              in if m=No_inc then (Bin(t,(x,Right),t2),No_inc)
                 else (Bin(t,(x,Balanced),t2),No_inc)
```

```
            end
          | (Greater,Right) ⇒
            let val (t,m) = add t2
            in (case m of
                  No_inc ⇒ (Bin(t1,(x,Right),t),No_inc)
                | Incleft ⇒ (rot_right_left(Bin(t1,(x,Balanced),t)),No_inc)
                | Incright ⇒ (rot_left(Bin(t1,(x,Balanced),t)),No_inc))
            end)
  in fst(add t)
  end;;
val add_to_avl:  ('a -> 'a -> 'a) -> ('a -> 'a -> comparison) -> ('a
                   * balance) btree -> 'a -> ('a * balance) btree
                 = <fun>
```

add_to_avl함수를 반복함으로써, 다음의 함수를 얻을 수 있다.

```
# val add_list_to_avl = fn option order ⇒ fold_left (add_to_avl option order);;
val add_list_to_avl:  ('a -> 'a -> 'a) -> ('a -> 'a -> comparison) -> ('a
                        * balance) btree -> 'a list -> ('a * balance) btree
                      = <fun>

# val mk_avl = fn option order ⇒ add_list_to_avl option order Empty;;
val mk_avl:  ('a -> 'a -> 'a) -> ('a -> 'a -> comparison) -> 'a list -> ('a
                   * balance) btree
                 = <fun>

# val merge_avl = fn option order ⇒
  it_btree (fn t x ⇒ add_to_avl option order t (fst x));;
val merge_avl:  ('a -> 'a -> 'a) -> ('a -> 'a -> comparison) -> ('a * balance)
                   btree -> ('a * 'b) btree -> ('a * balance) btree
                 = <fun>
```

그림 6.17은 다음의 값매김을 했을 때 순서대로 만들어지는 나무를 보여준다.

```
mk_avl (fn x y ⇒ x) int_comp [10,15,12,4,6,21,8,1,17,2]
```

그리고 AVL 나무를 만드는 함수로부터 정렬 함수를 만들 수도 있다. 이를 위해 다음과 같이 단순히 선형화 함수와 합성하면 된다.

```
# val avl_sort = fn order ⇒
  flat_avl o (mk_avl (fn x y ⇒ x) order);;
val avl_sort:  ('a -> 'a -> comparison) -> 'a list -> 'a list = <fun>
```

이 정렬 함수의 복잡도는 최악의 경우 $O(n \log(n))$이다. 참고로 quicksort는 최악의 경우에 n^2의 복잡도를 가진다.

6.5.4 AVL 나무에서 원소 제거하기

이항 탐색 나무에서 원소를 제거하는 연산과 비슷하게, AVL 나무에서 원소를 제거하는 것은 원소가 발견된 노드의 왼쪽 자식중에서 가장 큰 노드를 제거하기 위해 원소를 바꾸는 것에 의존한다.

나무에서 가장 큰 원소를 제거하는 것은 원래 나무의 오른쪽 부 나무의 높이를 줄일 수 있기 대문에 전체 나무의 균형에 영향을 줄 수 있다.

가장 큰 원소를 제거하는 함수는 6.4.4절에 있는 것과 비슷한 구조를 갖는다. 이 함수는 나무의 높이가 줄어들었는지에 대한 추가 정보를 함께 반환해야만 한다. 이를 위해 우리는 다음의 타입을 사용한다.

```
# type avl_rem_info = No_dec | Dec;;

# val balance = fn
    (Bin(_,(_,b),_)) ⇒ b
  | Empty ⇒ Balanced;;
val balance:  ('a * balance) btree -> balance = <fun>

# val balance_right = fn (t,x,t') ⇒
  case balance t of
    (Left | Balanced) ⇒ rot_right (Bin(t,(x,Balanced),t'))
  | Right ⇒ rot_left_right (Bin(t,(x,Balanced),t'));;
val balance_right:  ('a * balance) btree * 'a * ('a * balance) btree -> ('a
                    * balance) btree
                  = <fun>

# val balance_left = fn (t,x,t') ⇒
  case balance t' of
    (Right | Balanced) ⇒ rot_left (Bin(t,(x,Balanced),t'))
  | Left ⇒ rot_right_left (Bin(t,(x,Balanced),t'));;
val balance_left:  ('a * balance) btree * 'a * ('a * balance) btree -> ('a
                   * balance) btree
                 = <fun>

# val rec avl_remove_biggest = fn
    (Bin(t1,(a,_),Empty)) ⇒ (a,t1,Dec)
  | (Bin(t1,(a,Balanced),t2)) ⇒
      let val (a',t',b) = avl_remove_biggest t2
      in case b of
                Dec ⇒ (a',Bin(t1,(a,Left),t'),No_dec)
              | No_dec ⇒ (a',Bin(t1,(a,Balanced),t'),No_dec)
      end
  | (Bin(t1,(a,Right),t2)) ⇒
      let val (a',t',b) = avl_remove_biggest t2
      in case b of
                Dec ⇒ (a',Bin(t1,(a,Balanced),t'),Dec)
              | No_dec ⇒ (a',Bin(t1,(a,Right),t'),No_dec)
```

```
      end
  | (Bin(t1,(a,Left),t2)) ⇒
      let val (a',t',b) = avl_remove_biggest t2
      in case b of
                Dec ⇒ (a', balance_right(t1,a,t'),
                  case snd(root t1) of
                    (Left|Right) ⇒ Dec
                  | Balanced ⇒ No_dec)
              | No_dec ⇒ (a', Bin(t1,(a,Left),t'),No_dec)
      end
  | Empty ⇒ raise (Avl_exc "avl_remove_biggest:  empty avl");;
val avl_remove_biggest:  ('a * balance) btree -> 'a * ('a * balance) btree
                         * avl_rem_info
                       = <fun>
```

주요한 제거 함수는 역시 6.4.4절에 소개된 것과 같은 구조를 반복한다. 이것은 나무에서 제거할 원소를 찾는다. 이 원소를 찾으면, 즉, e가 제거할 원소라고 할 때 e(t_1,t_2)형태의 작은 나무를 찾으면, t_1을 인자로 넘겨서 함수 remove_biggest를 호출한다. 이 함수는 t_1에서 가장 큰 원소 a, 새로운 나무 t'_1와, t_1과 t'_1사이의 높이 차에 대한 정보, 이렇게 세가지를 반환한다. 이 정보는 필요하다면 나무 a(t'_1,t_2)의 균형을 맞추기 위해 나중에 사용된다.

게다가, 나무 a(t'_1,t_2)를 다시 균형 맞추는 것은 (이것이 교체된) 나무 e(t_1,t_2)보다 높이가 작은 나무를 만들 수 있다. 그러므로 뿌리를 향해 돌아가면서 다른 곳에도 다시 균형을 맞추기 위해 그 정보를 전달해야 한다.

그러므로 원소를 추가했을 때 했던 것과 반대로 원소를 제거할 때에는, 여러개의 노드를 다시 균형 맞추어야 할지도 모른다. 제거과정에서 발생하는 평균 회전 수에 대한 이론적인 결과는 없다. 그러나 실험 결과 5번의 제거 과정에 평균 한번의 회전이 필요하다고 한다. 이에 비해 추가 과정의 경우 2번 추가에 평균 한번 회전하게 된다.

그림 6.18은 그림 6.17에서 만들었던 AVL 나무에서 원소 21, 17, 2, 4, 15, 12, 8, 10, 6을 차례대로 제거했을 때 만들어지는 나무들을 보여준다.

이제 마지막으로 원소를 제거하는 함수를 살펴보자.

```
# val rec remove_from_avl = fn order t e ⇒
  let val rec remove = fn
    Empty ⇒ raise (Avl_search_exc "remove_from_avl")
  | (Bin(t1,(a,b),t2)) ⇒
      case (order e a) of
        Equiv ⇒
          if t1=Empty then (t2,Dec)
          else if t2=Empty then (t1,Dec)
          else
              let val (a',t',m) = avl_remove_biggest t1
              in (case m of
                    No_dec ⇒ (Bin(t',(a',b),t2),No_dec)
                  | Dec ⇒ (case b of
```

```
                        Balanced ⇒ (Bin(t',(a',Right),t2),No_dec)
                      | Left ⇒ (Bin(t',(a',Balanced),t2),Dec)
                      | Right ⇒ (balance_left(t',a',t2),
                          if balance t2=Balanced
                          then No_dec
                          else Dec)))
              end
      | Smaller ⇒
          let val (t',m) = remove t1
          in (case m of
                No_dec ⇒ (Bin(t',(a,b),t2),No_dec)
              | Dec ⇒ (case b of
                    Balanced ⇒ (Bin(t',(a,Right),t2),No_dec)
                  | Left ⇒ (Bin(t',(a,Balanced),t2),Dec)
                  | Right ⇒ (balance_left(t',a,t2),
                          if balance t2=Balanced
                          then No_dec
                          else Dec)))
          end
      | Greater ⇒
          let val(t',m) = remove t2
          in (case m of
                No_dec ⇒ (Bin(t1,(a,b),t'),No_dec)
              | Dec ⇒ (case b of
                    Balanced ⇒ (Bin(t1,(a,Left),t'),No_dec)
                  | Right ⇒ (Bin(t1,(a,Balanced),t'),Dec)
                  | Left ⇒ (balance_right(t1,a,t'),
                          if balance t1=Balanced
                          then No_dec
                          else Dec)))
          end
  in fst (remove t)
  end;;
val remove_from_avl:  ('a -> 'b -> comparison) -> ('b * balance) btree -> 'a
                        -> ('b * balance) btree
                      = <fun>
```

6.6 사전

여기서 사전이라는 것은 정렬 집합인 키를 이용해 이와 연관되어 있는 정보를 저정하는 구조이다. 보통 사전은 기본적인 예제이지만, 좀더 일반적으로 한정되어 있는 정렬된 영역에서 정의된 함수를 사전으로 간주될 수 있다.

이제 균형잡힌 AVL 나무에 기반해서 사전을 구현하는 것을 보일 것이다. 효율성의 측면에서, 이 구현 방식은 만약 고려 대상인 사전이 크기가 거의 고정되어 있거나 예상할 수 있는 타당한 범위내에서 변하는 것이라면, 해쉬를 이용

한 구현과 비교가 되지 않는다. 그렇지만 사전의 크기가 예상할 수 없거나 매우 심하게 변한다면, 균형잡힌 트리로 구현하는 것은 꽤 합당한 선택이다.

타입 dictionary는 사용할 순서 관계를 가리키는 항목 dict_rel과 저장할 자료를 저장하는 또다른 항목 dict_data로 이루어진 레코드이다. 정보는 인자 α와 관련되어 있고, 키는 인자 β와 연관되어 있다.

```
# type ('a,'b) dictionary =
      {dict_rel:  'b -> 'b -> 'comparison,
       dict_data:  ('a * 'b) avltree};;
```

원소를 탐색, 추가, 제거하는 함수는 AVL 나무에서 사용한 함수들을 직접 이용한다. 단지 여기에서 함수가 키에 접근하기 위해서 비교 함수를 고쳐야 한다. 탐색 함수는 리스트에서 assoc함수를 쓰는 것과 유사하게 dict_assoc이다.

```
# val dict_assoc = fn e d ⇒
  fst(fst(find_avl(fn x y ⇒ d.dict_rel x (snd y)) e d.dict_data));;
val dict_assoc:  'a -> {dict_data:  ((('b * 'c) * 'd) * 'e) btree,
                        dict_rel:  'a -> 'd
                        -> comparison}
                   -> 'b
                 = <fun>
```

일반적으로 사전에 원소를 추가하는 것은 존재하는 원소를 교체하는 것을 수반한다.

```
# val dict_add_or_replace = fn {dict_rec=c,dict_data=t} e ⇒
  {dict_rel=c,
   dict_data=add_to_avl (fn x y ⇒ y) (fn x y ⇒ c (snd x) (snd y)) t e};;
val dict_add_or_replace:  {dict_data:  (('a * 'b) * balance) btree,
                           dict_rec:  'b -> 'b -> comparison}
                          -> 'a * 'b ->
                          {dict_data:  (('a * 'b) * balance) btree,
                           dict_rel:  'b -> 'b -> comparison}
                        = <fun>
```

그렇지만 만약 함수 add_to_avl에서 옵션을 다르게 준다면, 또다른 추가 함수를 만들 수도 있다.

제거 함수는 다음과 같다

```
# val dict_remove = fn {dict_rel=c,dict_data=t} key ⇒
  {dict_rel=c,
   dict_data=remove_from_avl (fn x y ⇒ c x (snd y)) t key};;
val dict_remove:  {dict_data:  (('a * 'b) * balance) btree, dict_rel:  'c -> 'bc
                   -> comparison}
                  -> 'c ->
                  {dict_data:  (('a * 'b) * balance) btree, dict_rel:  'c -> 'b
                   -> comparison}
                = <fun>
```

때때로 다음과 같이 두개의 사전을 합치는 것이 필요할 수 있다.

```
# val dict_merge = opt d1 d2 ⇒
  if not (d1.dict_rel = d2.dict_rel)
  then failwith "dict_merge:  dictionaries have different orders"
  else {dict_rel=d1.dict_rel,
          dict_data=merge_avl opt (fn x y ⇒ d1.dict_rel (snd x) (snd y))
                  d1.dict_data d2.dict_data};;
val dict_merge:  ('a * 'b -> 'a * 'b -> 'a * 'b) ->
                    {dict_data:  (('a * 'b) * balance) btree, dict_rel:  'b -> 'b
                     -> comparison}
                    ->
                    {dict_data:  (('a * 'b) * 'c) btree, dict_rel:  'b -> 'b ->
                     comparison}
                    ->
                    {dict_data:  (('a * 'b) * balance) btree, dict_rel:  'b -> 'b
                     -> comparison}
                  = <fun>
```

6.7 정렬 집합

균형잡힌 나무구조는 정렬 집합을 구현하는 데에 편리하다. 이 집합은 균형잡힌 나무로 구성된 집합 원소들을 저장하는 항목 set_elements와 나무에서 사용하는 순서 관계를 보여주는 항목 set_order로 구성된 레코드로 표현된다.

```
# type 'a set =
  {set_elements:  'a avltree,
   set_order:  'a -> 'a -> comparison};;
```

집합을 만드는 연산은 두개의 인자를 받는다. 하나는 순서 관계이고, 다른 하나는 집합에 들어갈 원소의 리스트이다.

```
# val make_set = fn c l ⇒
  {set_elements=mk_avl (fn x y ⇒ x) c l,
   set_order=c};;
val make_set:  ('a -> 'a -> comparison) -> 'a list ->
                    {set_elements:  ('a * balance) btree, set_order:  'a -> 'a ->
                     comparison}
                  = <fun>
```

다른 연산들은 더이상 원소들 사이의 순서 관계를 가지고 다닐 필요가 없는데, 이것은 순서 관계가 이미 집합의 내부 표현에 들어가 있기 때문이다. 이 연산들은 AVL 나무구조에서 이미 정의한 연산들을 사용한다.

비어있는지 확인

```
# val set_isempty = fn s ⇒
  (s.set_elements=Empty);;
val set_isempty:  {set_elements:  'a btree} -> bool = <fun>
```

원소인지 확인

```
# val set_member = fn x s ⇒
  belongs_to_avl s.set_order x s.set_elements;;
val set_member:  'a ->
                    {set_elements:  ('b * 'c) btree, set_order:  'a -> 'b ->
                     comparison}
                    -> bool
                  = <fun>
```

집합원소들에 대한 반복기

다음 두가지 반복 함수들은 집합의 원소들을 많이 사용하는 것을 프로그램하도록 해준다. 그럼에도 불구하고 이러한 반복기들이 자료 집합의 표현 방식과 연관되어 있다는 사실에 주의를 기울여야 한다.

```
# val set_it = fn f s ⇒
  btree_it (fn x y ⇒ f (fst x) y) s.set_elements;;
val set_it:  ('a -> 'b -> 'b) -> {set_elements:  ('a * 'c) btree} -> 'b -> 'b
              = <fun>

# val it_set = fn f x s ⇒
  it_btree (fn x y ⇒ f x (fst y)) x s.set_elements;;
val it_set:  ('a -> 'b -> 'b) -> 'a -> {set_elements:  ('b * 'c) btree} -> 'a

# val do_set = fn f s ⇒ do_avl f s.set_elements;;
val do_set:  ('a -> unit) -> {set_elements:  'a} -> unit = <fun>
```

시험

```
# exception Set_exc of string;;

# val set_forall = fn p s ⇒
  do_set (fn x ⇒ if not (p x) then raise (Set_exc "")) s; true
  handle Set_exc _ ⇒ false;;
val set_forall:  ('a -> bool) -> 'a set -> bool = <fun>

# val set_exists = fn p s ⇒
  do_set (fn x ⇒ if (p x) then raise (Set_exc "")) s; false
  handle Set_exc _ ⇒ true;;
val set_exists:  ('a -> bool) -> 'a set -> bool = <fun>
```

포함과 동등관계

```
# val sub_set = fn s1 s2 ⇒
  set_forall (fn e ⇒ set_member e s2) s1;;
val sub_set:  'a set -> 'a set -> bool = <fun>

# val set_equiv s1 s2 =
  sub_set s1 s2 && sub_set s2 s1;;
val set_equiv:  'a set -> 'a set -> bool = <fun>
```

원소들의 정렬 리스트

```
# val list_of_set = fn s ⇒
  flat_avl s.set_elements;;
val list_of_set:  'a set -> 'a list = <fun>
```

임의의 원소 선택
나무의 뿌리에 있는 것을 얻는다.

```
# val set_random_element = fn s⇒
  fst (root s.set_elements);;
val set_random_element:  'a set -> 'a = <fun>
```

추가

```
# val add_to_set = fn s x ⇒
  set_elements=add_to_avl (fn x y ⇒ x) s.set_order s.set_elements x,  ;;
val add_to_set:  'a set -> 'a -> 'a set = <fun>

# val add_list_to_set = fn s l ⇒
  fold_left add_to_set s l;;
val add_list_to_set:  'a set -> 'a list -> 'a set = <fun>
```

제거

```
# val remove_from_set = fn s x ⇒
  set_elements=remove_from_avl s.set_order s.set_elements x,
  handle _ ⇒ raise (Set_exc "remove_from_set");;
val remove_from_set:  'a set -> 'a -> 'a set = <fun>

# val remove_list_from_set = fn s ⇒ fold_left remove_from_set s;;
val remove_list_from_set:  'a set -> 'a list -> 'a set = <fun>
```

연산 remove_from_set은 제거할 원소가 집합에 존재하지 않으면 실패한다. 또한 이러한 경우에 실패하지 않는 함수도 있으면 유용하다. 특히 이 함수는 집합사이의 차집합이나 교집합 연산에 도움이 된다.

```
# val subtract_from_set = fn s x ⇒
  remove_from_set s x
  handle _ ⇒ s;;
val subtract_from_set:  'a set -> 'a -> 'a set = <fun>
```

집합연산

```
# val set_union = fn s1 s2 ⇒
  if not(s1.set_order=s2.set_order)
  then raise (Set_exc "set_union:  different set orders")
  else it_set add_to_set s1 s2;;
val set_union:  'a set -> 'a set -> 'a set = <fun>

# val set_diff = fn s1 s2 ⇒
  if not(s1.set_order=s2.set_order)
  then raise (Set_exc "set_diff:  different set orders")
  else it_set subtract_from_set s1 s2;;
val set_diff:  'a set -> 'a set -> 'a set = <fun>

# val set_intersection = fn s1 s2 ⇒ set_diff s1 (set_diff s1 s2);;
val set_intersection:  'a set -> 'a set -> 'a set = <fun>
```

6.8 함수형 큐(Functional Queues)

큐는 원소의 출력 순서가 입력 순서와 같은 임시로 자료를 저장하는 구조이다. 앞에서 원형 리스트과 파괴적 연산을 사용하는 구현을 살펴보았다. 사실, 큐는 순수 함수형 언어에서 구현하기 힘든 자료 구조의 전형적인 예이다. 예를 들어 큐를 리스트로 구현하면, 새로운 원소를 추가할 때마다 완전히 리스트를 새로 복사해야 한다.

여기서 제공하는 구현 방식은 차수에 대해 관심을 줄이기로 한다. 그래서 파괴적인 연산을 이용해 구현을 하면 추가, 제거 연산을 하는 데에 상수 시간이 걸리지만, 이번 것은 로그 함수에 비례하는 시간이 걸린다. 그렇다 하더라도, 이 구현 방식은 순수 함수형 언어에서 큐를 프로그램하는 것이 완전히 불가능 하지 않다는 것을 보여줄 것이다.

큐는 원소가 들어온 순서에 의해 정렬되어 있는 AVL 나무이다. 가장 늦게 들어온 원소가 가장 크다고 간주되고, 따라서 가장 오른쪽 가지 끝에 위치한다. 가장 처음 들어온 원소는 가장 작다고 간주되며, 따라서 가장 왼쪽 가지의 끝에 위치한다. 왼쪽 회전 연산을 이용해 AVL 나무를 계속 균형 잡히게 만든다.

```
# type queue_mod1 = Inc | No_Inc;;
type queue_mod1 = Inc | No_Inc

# val rec enqueue = fn t e ⇒
  let val rec add = fn
    Empty ⇒ (Bin(Empty,(e,Balanced),Empty),Inc)
```

```
  | (Bin(t1,(x,b),t2)) ⇒
      let val (t,m) = add t2
      in case (b,m) of
                (Balanced,No_Inc) ⇒ (Bin(t1,(x,Balanced),t),No_Inc)
              | (Balanced,_) ⇒ (Bin(t1,(x,Right),t),Inc)
              | (Left,No_Inc) ⇒ (Bin(t1,(x,Left),t),No_Inc)
              | (Left,_) ⇒ (Bin(t1,(x,Balanced),t),No_Inc)
              | (Right,No_Inc) ⇒ (Bin(t1,(x,Right),t),No_Inc)
              | (Right,Inc) ⇒ (rot_left (Bin(t1,(x,Balanced),t)),No_Inc)
      end
  in fst (add t)
  end;;
val enqueue:  ('a * balance) btree -> 'a -> ('a * balance) btree = <fun>

# type queue_mod2 = Dec | No_Dec;;
type queue_mod2 = Dec | No_Dec

# val dequeue = fn t ⇒
  let val rec sub = fn
    Empty ⇒ failwith "dequeue:  empty queue"
  | (Bin(Empty,(a,b),t2)) ⇒ ((a,t2),Dec)
  | (Bin(t1,(a,b),t2)) ⇒
      let val ((a',t'),m) = sub t1
      in case m of
            No_Dec ⇒ ((a',Bin(t',(a,b),t2)),No_Dec)
          | Dec ⇒ (case b of
                    Balanced ⇒ ((a',Bin(t',(a,Right),t2)),No_Dec)
                  | Left ⇒ ((a',Bin(t',(a,Balanced),t2)),Dec)
                  | Right ⇒ ((a',balance_left(t',a,t2)),
                         if balance t2=Balanced
                         then No_Dec
                         else Dec))
      end
  in fst (sub t)
  end;;
val dequeue:  ('a * balance) btree -> 'a * ('a * balance) btree = <fun>
```

큐를 사용해서 나무를 폭 우선 방식으로 추적할 수 있다.

```
# val breadth_it_btree = fn orient f e t ⇒
  let val rec trav = fn x xx ⇒ case xx of
    Empty ⇒ x
  | q ⇒ let val (t,q') = dequeue q
          in case t of
                Empty ⇒ trav x q'
              | Bin(t1,a,t2) ⇒
                     trav (f x a) (enqueue (enqueue q' t1) t2)
```

```
          end
  in trav e (enqueue Empty t)
  end;;
val breadth_it_btree:  'a -> ('b -> 'c -> 'b) -> 'b -> 'c btree -> 'b = <fun>

# val breadth_btree_it = fn f t e ⇒
  let val rec trav = fn
    Empty ⇒ e
  | q ⇒ let val (t,q') = dequeue q
          in case t of
                Empty ⇒ trav q'
              | Bin(t1,a,t2) ⇒
                      f a (trav (enqueue (enqueue q' t1) t2))
          end
  in trav (enqueue Empty t)
  end;;
val breadth_btree_it:  ('a -> 'b -> 'b) -> 'a btree -> 'b -> 'b = <fun>

# val breadth_flat_bree = fn f t ⇒
  breadth_btree_it (fn x l ⇒ f x::l) t [];;
val breadth_flat_bree:  ('a -> 'b) -> 'a btree -> 'b list = <fun>
```

6.9 요약

이번 장에서는 **이항 탐색 나무**의 개념을 알아보았다. 이것은 정렬 집합을 효율적으로 탐색하는 데 도움이 된다. 그리고 나서 **AVL 나무**와 추가, 제거 연산을 한 후에도 **균형**을 유지하는 관련된 알고리즘을 공부하였다. 그 후에, AVL 나무를 사용하여 집합과 표를 효율적으로 다룰 수 있는 타입 set과 dictionary를 정의하였다.

6.10 더 배울 내용들

AVL 나무는 커다란 정보의 자료집합을 다루는 많은 기법 중에 하나일 뿐이다. 해쉬와 같은 다른 기법들이 많이 존재한다. 이런 기법에 대한 많은 책, 예를 들어 [35], 이 있다.

nML은 사전과 표를 다루는 라이브러리 (Set, Map)이 있고 또한 해쉬 테이블을 다루는 (Hashtbl)이 있다.

7 장

그래프와 문제해결

8 장

문법 구조 분석

9 장

기하와 그리기

10 장

정확한 산술 연산

III 편

구현

이번 편에서는 nML과 같은 언어를 구현하는 방법에 대해서 살펴본다. 우리는 nML의 구현방법을 완벽하게 다루지는 않고, 대신 그러한 구현이 실현 가능하다는 점을 알아보려고 한다. 그래서 컴파일과 타입 유추(type synthesis)를 구현할 때 겪는 주요한 어려운 문제를 살펴보기 위해 nML의 부분 집합인 작은 언어를 다룰 것이다.

11장에서는 nML로 nML 값매김 장치(evaluator)를 정의한다. 이를 통해 컴파일러를 만드는 것이 가능함을 강조한다. 환경(environment)이라는 아이디어로 변수를 다루고, 함수값을 표현하기 위한 함수 묶음(closure)에 관한 아이디어를 사용할 것이다.

12장에서는 두가지 주제에 대해 동시에 살펴본다. 그것은 함수형 언어를 구현하는 데에 필요한 컴파일 개요(schema)와 메모리 관리 방식에 관한 것이다. 메모리 관리에 대해서는 메모리 공간을 잡는 것만을 자세히 다룬다. 메모리 재활용(garbage collection)은 개략적으로만 알아본다.

우리가 사용하는 기계어는 어셈블리 언어의 명령어에 비하면 상대적으로 추상적인 단계의 것이다. 그럼에도 불구하고 이 기계어도 실제 기계어로 직접 변환될 수 있다.

13장에서는 타입 유추기(type synthesizer)를 다룬다. 여기서는 순수 함수형 방식으로 구현한 임시 버전을 알아본다. 그리고 나서 파괴적 동일화 알고리즘을 사용한, 좀더 효율적인 버전으로 관심을 옮긴다. 이 버전은 nML의 실제 타입 유추기와 꽤 유사하다.

11 장

값매김

3장에서 표현식의 값을 매기는 방법을 알아보았다. 그것은 다시 쓰기(rewrite)에 기반을 두고 있다. 이 방식은 각각의 구현이 참조하는 의미 구조를 정의한다. 그러나 실제로 다시 쓰기를 사용하는 구현 방식은 매우 비효율적이다. 좀더 효율적으로 만들기 위해서는, 다시 쓰기 대신에 이것을 흉내낼 수 있는 다른 방식으로 대체해야 한다. 한가지 방법은 **환경**(`environment`)를 이용하는 아이디어다.

이번 장에서 환경에 기반한 값매김을 알아본다. 환경과 nML 표현식을 인자로 받아 표현식의 값을 결과로 반환하는 값매김 함수를 만들 것이다. 이런 값매김 함수를 구현하는 것은 12장에서 컴파일러를 만들기 위해 더 고민해 볼 것이다.

11.1절에서 환경을 이용해 표현식을 값매김 하는 것을 살펴본다. 3장에서와 같이 언어에서 표현식을 만들어내는 각각의 방법에 대한 추론 규칙(inference rule)을 이용한다. 이 추론 규칙은 nML에 적용된 값매김 방법인, 값에 의한 값매김을 정의한다. 그리고 나서 11.2절에서는 이러한 규칙을 nML로 쓰여진 값매김 장치로 어떻게 구현하는지를 공부한다. 마지막으로 11.3절에서 값에 의한 값매김 대신에 미루어진 값매김(delayed evaluation)을 고려하도록 값매김 장치를 수정하는 것에 대해 알아본다.

11.1 환경을 이용한 값매김

환경이란 특정 변수가 어떤 값에 매여져 있는지를 적어놓은 표로 생각할 수 있다. 예를 들어 nML 대화식 모드에서 변수 $x_1,\cdots,x_n$을 정의할 때, 이 변수와 관련된 값 $v_1,\cdots,v_n$은 *toplevel*과 연관된 환경에 저장된다. 이 환경은 변수 x_1과 값 v_1을, 변수 x_2와 값 v_2를, 이런 식으로 관계를 맺어주는 표이다.

계산과정 도중에 `fn`이나 `let val ···in···end`, `let val rec···in···end`으로 매여지는(bounded) 프로그램의 변수들의 값들을 저장하기 위해, 이러한 표를 사용하는 것을 일반적으로 적용할 수 있다. `(fn x ⇒ e) v` 형태의 표현식의 값을 매겨야 하는 경우에 `e`에서 `x`대신 `v`로 치환하는 대신에, `x`가 `v`로 매여져 있는 환경에서 `e`를 값매김 하는 것이다.

환경을 이용해 계산하는 방식에서 함수 표현식은 문제가 될 수 있는데, 이

것은 함수 표현식 안에 환경 안에서가 아니면 아무 의미도 없는 자유변수(free variable)가 존재할 수 있기 때문이다. 따라서 환경에서 함수 표현식의 값은 표현식와 환경의 쌍이어야 한다. 이 쌍을 **함수 묶음**(closure)라고 한다.

예를 들어 (fn x ⇒ fn y ⇒ x * y) 3이라는 표현식을 생각해보자. 치환을 이용하는 계산을 하면 이 표현식의 값은 (fn y ⇒ 3 * y)이다. 반면에 환경을 이용해 계산을 하면 값은 <[x:3],(fn y ⇒ x*y)>가 될 것이다.

환경을 이용한 표현식의 값매김은 다음과 같은 삼항 연산자를 이용해 정식으로 정의할 수 있다.

$$\mathtt{E} \vdash \mathtt{e} \Rightarrow \mathtt{V}$$

이것은 다음을 나타낸다.

"표현식 e는 환경 E에서 v라는 값을 갖는다."

이제 언어를 만들어내는 생성자에 대해서 추론 규칙을 만들어보자. 다음의 규칙들은 값에 의한 값매김에 해당한다. (이 주제에 대해서는 69쪽을 참고한다.)

$$\frac{\mathtt{E}(\mathtt{x}) = \mathtt{v}}{\mathtt{E} \vdash \mathtt{x} \Rightarrow \mathtt{v}}(\text{변수})$$

환경 E에서 변수 x가 매여진 값이 v라면, 표현식 x는 이 환경에서 값 v를 갖는다.

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \Rightarrow \mathtt{v}_1 \quad \cdots \quad \mathtt{E} \vdash \mathtt{e}_n \Rightarrow \mathtt{v}_n}{\mathtt{E} \vdash (\mathtt{e}_1, \cdots, \mathtt{e}_n) \Rightarrow (\mathtt{v}_1, \cdots, \mathtt{v}_n)}(\text{N-튜플})$$

각각의 표현식 $\mathtt{e}_i$가 환경 E에서 값 $\mathtt{v}_i$를 갖는다면, 표현식 $(\mathtt{e}_1,\cdots,\mathtt{e}_n)$은 이 환경에서 값 $(\mathtt{v}_1,\cdots,\mathtt{v}_n)$을 갖는다.

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \Rightarrow \mathtt{true} \qquad \mathtt{E} \vdash \mathtt{e}_2 \Rightarrow \mathtt{v}}{\mathtt{E} \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \Rightarrow \mathtt{v}}(\text{조건1})$$

만약 환경 E에서 표현식 $\mathtt{e}_1$이 값 true이고 표현식 $\mathtt{e}_2$가 값 v를 갖는다면, 표현식 if $\mathtt{e}_1$ then $\mathtt{e}_2$ else $\mathtt{e}_3$는 이 환경에서 값 v를 갖는다.

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \Rightarrow \mathtt{false} \qquad \mathtt{E} \vdash \mathtt{e}_3 \Rightarrow \mathtt{v}}{\mathtt{E} \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \Rightarrow \mathtt{v}}(\text{조건2})$$

만약 환경 E에서 표현식 $\mathtt{e}_1$이 값 false이고 표현식 $\mathtt{e}_3$가 값 v를 갖는다면, 표현식 if $\mathtt{e}_1$ then $\mathtt{e}_2$ else $\mathtt{e}_3$는 이 환경에서 값 v를 갖는다.

$$\frac{}{\mathtt{E} \vdash (\mathtt{fn}\ \mathtt{x} \Rightarrow \mathtt{e}) \Rightarrow < \mathtt{E}, (\mathtt{fn}\ \mathtt{x} \Rightarrow \mathtt{e}_) >}(\text{함수1})$$

표현식 (fn x ⇒ e)는 환경 E에서 함수 묶음 <E,(fn x ⇒ e)>를 값으로 한다.

$$\frac{}{\mathtt{E} \vdash (\mathtt{fn}(\mathtt{x}_1, \cdots, \mathtt{x}_n) \Rightarrow \mathtt{e}) \Rightarrow < \mathtt{E}, (\mathtt{fn}(\mathtt{x}_1, \cdots, \mathtt{x}_n) \Rightarrow \mathtt{e}) >}(\text{함수2})$$

표현식 (fn (x_1,$\cdots$,x_n) ⇒ e) 는 환경 E에서 함수 묶음 <E,(fn (x_1,$\cdots$,x_n) ⇒ e)> 를 값으로 한다.

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \Rightarrow <\mathtt{E}\prime, (\mathtt{fn\ x} \Rightarrow \mathtt{e})> \quad \mathtt{E} \vdash \mathtt{e}_2 \Rightarrow \mathtt{v}_2 \quad (\mathtt{x} : \mathtt{v}_2) :: \mathtt{E}\prime \vdash \mathtt{e} \Rightarrow \mathtt{v}}{\mathtt{E} \vdash (\mathtt{e}_1\ \mathtt{e}_2) \Rightarrow \mathtt{v}} (\text{적용1})$$

만약 환경 E에서 표현식 e_1이 값으로 함수 묶음 <E/,(fn x ⇒ e)>를 갖고, 표현식 e_2가 값 v_2를 가지며, v_2와 x의 묶음을 추가한 환경 E/에서 표현식 e가 값 v를 갖는다면, 표현식 (e_1 e_2)는 환경 E에서 값 v를 갖는다.

새로운 값 묶음을 환경에 추가하기 위해 사용한 표기방식은 환경이 관계 리스트로 표현되었다고 가정하고 있다. 이러한 리스트에서 같은 변수가 한번 이상 나타날 수 있지만, 실제로 영향을 미치는 값 묶음은 이 변수를 정의한 가장 왼쪽의 값 묶음이다. 즉, 리스트의 머리부분에 가장 가까운 것이나 환경에 가장 최근에 추가된 값 묶음이 될 것이다.

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \Rightarrow <\mathtt{E}\prime, (\mathtt{fn}(\mathtt{x}_1, \cdots, \mathtt{x}_n) \Rightarrow \mathtt{e})> \quad \mathtt{E} \vdash \mathtt{e}_2 \Rightarrow (\mathtt{v}_1, \cdots, \mathtt{v}_n) \quad (\mathtt{x}_1 : \mathtt{v}_1) :: \cdots :: (\mathtt{x}_n : \mathtt{v}_n) :: \mathtt{E}\prime \vdash \mathtt{e} \Rightarrow \mathtt{v}}{\mathtt{E} \vdash (\mathtt{e}_1\ \mathtt{e}_2) \Rightarrow \mathtt{v}} (\text{적용2})$$

만약 환경 E에서 표현식 e_1이 함수 묶음 <E/,(fn (x_1,$\cdots$,x_n) ⇒ e)>를 갖고, 표현식 e_2가 값 (v_1,$\cdots$,v_n)를 가지며, v_i와 x_i사이의 묶음들을 추가한 환경 E/에서 표현식 e가 값 v를 갖는다면, 표현식 (e_1 e_2)는 환경 E에서 값 v를 갖는다.

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \Rightarrow \mathtt{v}_1 \qquad (\mathtt{x} : \mathtt{v}_1) :: \mathtt{E} \vdash \mathtt{e}_2 \Rightarrow \mathtt{v}}{\mathtt{E} \vdash \mathtt{let\ val\ x} = \mathtt{e}_1\ \mathtt{in\ e}_2\ \mathtt{end} \Rightarrow \mathtt{v}} (\text{Let})$$

만약 환경 E에서 표현식 e_1이 값 v_1을 갖고, v_1과 x의 값 묶음을 추가한 환경 E에서 표현식 e_1가 값 v를 갖는다면, 표현식 let val x=e_1 in e_2 end는 환경 E에서 값 v를 갖는다.

재귀적 표현을 고려하려면 다음과 같은 방식을 적용해야 한다. let val rec f x = e와 같이 정의된 재귀함수 f는 함수 묶음 <E,(fn x ⇒ e)>으로 표현될 수 있으며 이때 환경 E는 무한히 f:<E,(fn x ⇒ e)> 의 쌍을 포함해야 한다.

주어진 환경 E, 함수 이름 f, 그리고 (fn x ⇒ e) (여기서 e는 f를 포함할 수 있다.)라는 함수 표현식에 대해서, E/=(f:<E/,(fn x ⇒ e)>)::E 와 성질을 갖는 환경 E/을 만들 수 있다고 가정한다. 이렇게 하면 재귀적 표현을 고려한 규칙은 다음과 같다.

$$\frac{\mathtt{E}\prime = (\mathtt{f} :< \mathtt{E}\prime, (\mathtt{fn\ x} \Rightarrow \mathtt{e}) >) :: \mathtt{E} \vdash \mathtt{e}_2 \Rightarrow \mathtt{v}}{\mathtt{E} \vdash \mathtt{let\ val\ rec\ f\ x} = \mathtt{e}_1\ \mathtt{in\ e}_2\ \mathtt{end} \Rightarrow \mathtt{v}} (\text{Letrec})$$

만약 변수 f가 함수 묶음 <E/, (fn x ⇒ e)>를 값으로 갖고 E/=(f:<E/,(fn x ⇒ e)>)::E 를 만족하는 환경 E/ 에서 표현식 e_2가 값 v를 갖는다면, 표현식 let val rec f x =e_1 in e_2 end 는 환경 E에서 값 v를 갖는다.

만약 이 규칙을 반대방향으로 즉, 아래에서 위방향으로 읽는다면, 어떠한 표현식이라도 값을 매길 수 있는 효과적인 수단을 제공한다는 것을 알 수 있을 것이다. 우리는 nML로 nML 값매김 장치를 만들 때 이러한 관찰 결과를 이용할 것이다.

11.2 nML로 값매김 장치 만들기

nML로 쓰여진 값매김 함수로 nML 값매김 장치를 정의하는 것은 몇가지 교묘한 수법을 필요로 한다. 왜냐하면 이미 값매김 장치, 자체를 값매김할 수 있는 방법을 알고 있어야 함을 가정하고 있기 때문이다. 그럼에도 불구하고 이 과정에서 언어의 모든 표현식의 값매김을 매우 작은 수의 간단한 함수들을 값매김하는 것으로 간소화 하기 때문에, 도움이 될 것이다. 특히 값매김 함수는 고차 특성을 이용하지 않는다. 따라서 언어의 구현을 정확히 정의하기 위해서, 어떻게 값매김 함수 그 자체가 기계어로 값매김되거나 컴파일될 수 있는지 보이면 된다.

이제 nML 커널의 추상적 문법 구조(abstract syntax)를 정의하는 타입 `ml_exp`와 계산의 결과값을 정의하는 타입 `val`을 소개하면서 시작해보자. 함수 묶음안의 환경은 타입이 `(string * val) list`인 관계 리스트로 표현한다.

우리가 사용할 nML커널은 n-튜플에 대한 표기법은 없고 오직 쌍(pair)만 가지고 있다. 모든 함수는 인자로 변수의 n-튜플이 아닌, 유일한 변수를 가진다. 쌍을 접근하는 함수(`fst`와 `snd`)들은 산술연산자(`+`, `-`, `*`)와 비교연산자(`=`, `<`)등과 마찬가지로 미리 정의되어있다.

미리 정의된 연산자들은 추상적 문법구조에서 타입 `ml_unop`와 `ml_binop`에 해당한다.

```
# type ml_unop = Ml_fst | Ml_snd;;

# type ml_binop = Ml_add | Ml_sub | Ml_mult | Ml_eq | Ml_less;;

# type ml_exp =
   Ml_int_const of int                          (* 정수 상수*)
 | Ml_bool_const of bool                         (* 불 상수*)
 | Ml_pair of ml_exp * ml_exp                        (* 쌍 *)
 | Ml_unop of ml_unop * ml_exp                (* 단항 연산자 *)
 | Ml_binop of ml_binop * ml_exp * ml_exp     (* 이항 연산자 *)
 | Ml_var of string                              (* 변수 *)
 | Ml_if of ml_exp * ml_exp * ml_exp            (* 조건문 *)
 | Ml_fun of string * ml_exp                      (* 함수 *)
 | Ml_app of ml_exp * ml_exp                      (* 적용 *)
 | Ml_let of string * ml_exp * ml_exp             (* 선언 *)
 | Ml_letrec of string * ml_exp * ml_exp       (* 재귀 선언 *)
;;
```

표현식의 값은 타입 `val`로 표시된다. 그것은 정수 상수, 불(bool) 상수, 쌍, 함수 묶음이다.

```
# type value =
        Int_Const of int
      | Bool_Const of bool
      | Pair of value * value
      | Clo of (string * value) list * ml_exp;;
```

이전과 같이 이들 타입에 해당하는 문법구조를 분석하는 함수를 이미 가지고 있다고 가정한다.

```
# ml_exp_of_string;;
val ml_exp_of_string:  string -> ml_exp = <fun>

# ml_exp_of_string "let val x=1 in x+2 end";;
val it:  ml_exp = Ml_let ("x", Ml_int_const 1,
                  Ml-binop (Ml_add, Ml_var "x", Ml_int_const2))
```

함수 `ml_eval`은 환경에서 표현식의 값을 계산한다. 이를 위해 타입이 `value -> value -> value`인 함수와 기본 연산자 사이의 관계를 맺어주는 보조 함수 `ml_eval_binop`를 사용한다.

```
# val ml_eval_binop = fn
        Ml_add (Int_Const m) (Int_Const n) ⇒ Int_Const (m + n)
      | Ml_sub (Int_Const m) (Int_Const n) ⇒ Int_Const (m - n)
      | Ml_mult (Int_Const m) (Int_Const n) ⇒ Int_Const (m * n)
      | Ml_eq(Int_Const m) (Int_Const n) ⇒ Bool_Const (m = n)
      | Ml_less (Int_Const m) (Int_Const n) ⇒ Bool_Const (m < n)
      | _ _ _ ⇒ failwith "ml_eval_binop:  wrong types";;
val ml_eval_binop:  ml_binop -> value -> value -> value = <fun>

# fun ml_eval env exp = case exp of
        (Ml_int_const n) ⇒ Int_Const n
      | (Ml_bool_const b) ⇒ Bool_Const b
      | (Ml_pair (e1,e2)) ⇒ Pair(ml_eval env e1, ml_eval env e2)
      | (Ml_unop (op,e)) ⇒ (case (op, ml_eval env e) of
                              (Ml_fst, Pair(v1,v2)) ⇒ v1
                            | (Ml_snd, Pair(v1,v2)) ⇒ v2
                            | _ ⇒ failwith "ml_eval:  wrong types")
      | (Ml_binop (op, e1, e2)) ⇒
             let val v2 = ml_eval env e2
                 val v1 = ml_eval env e1
             in ml_eval_binop op v1 v2
             end
      | (Ml_var x) ⇒ (List.assoc x env
                 handle Not_found ⇒ failwith "unbound variable")
      | (Ml_if (c,e1,e2)) ⇒
             (case ml_eval env c of
                       (Bool_Const true) ⇒ ml_eval env e1
                     | (Bool_Const false) ⇒ ml_eval env e2
                     | _ ⇒ failwith "ml_eval:  wrong types")
      | (Ml_fun (x,e)) ⇒ Clo(env,(x,e))
      | (Ml_app (e1,e2)) ⇒
             (case (ml_eval env e1, ml_eval env e2) of
                   (Clo(env',Ml_fun (x,e)),v)
```

```
                        ⇒ ml_eval ((x,v)::env') e
                | _ ⇒ failwith "ml_eval:  wrong types")
      | (Ml_let (x,e1,e2)) ⇒
             let val v = ml_eval env e1
             in ml_eval ((x,v)::env) e2
             end
      | (Ml_letrec (f,e1,e2)) ⇒
             (case e1 of
                Ml_fun _ ⇒ let val rec env' = (f,Clo(env',e1))::env
                       in ml_eval env' e2
                       end
              | _ ⇒ failwith "illegal recursive definition");;
val ml_eval:  (string * value) list -> ml_exp -> val = <fun>
```

재귀적인 정의는 환경을 재귀적인 정의한다는 점을 주목하라. 이러한 종류의 (함수가 아닌 객체에 영향을 미치는) 재귀적 정의는 방금 만든 값매김 장치로는 다룰 수 없다. 그렇지만 값의 재귀적인 정의들은 12장에서 볼 컴파일러에 의해 잘 다루어 질 것이다.

여기에 값매김에 대한 두가지 예가 있다.

```
# ml_eval []
      (ml_exp_of_string
          ("let val double = fn f ⇒ fn x ⇒ f(f x) in " ^
           "let val sq = fn x ⇒ x*x in " ^
           "(double sq) 5" end end));;
val it:  value = Int_Const 625

# ml_eval []
      (ml_exp_of_string
          ("let val rec fact=" ^
           " fn n ⇒ if n=0 then 1 else n*(fact(n-1)) in " ^
           "fact 10" end));;
val it:  value = Int_Const 3628800
```

연습 문제

11.1 튜플과 n개의 튜플을 인자로 갖는 함수를 다루도록 타입 ml_exp와 함수 ml_eval을 완성하라.

11.3 필요가 있을 때 값매김 = 늦게 값매김 = 미루어진 값매김

이제 필요가 있을 때 값매김을 하도록 값매김 장치를 수정한다. 그래서 계산을 수행하기 위해 **필요**할 때까지, 어떤 표현식의 값을 값매김하는 과정을 **미루기**로 한다.

다음과 같은 원리를 이용할 것이다. 환경을 이용한 값매김 장치에서는, 표현식을 값매김 하는 것이 표현식에 있는 자유 변수들의 값을 가지고 있는 환경에 의존하고 있다. 따라서 표현식의 값매김을 미루기 위해서 우리는 그 환경을 저장해야 한다.

표현식과 값매김 장치의 환경을 맺어줌으로써, 표현식을 값매김하는 데에 필요한 모든 정보를 보존할 수 있고, 따라서 값매김을 늦게 수행할 수 있다.

그 결과 값매김을 미루고자 하는 표현식을 다루기 위해서는, 표현식과 환경을 포함하는 쌍을 만드는 것으로 가능하다. 이 과정을 값매김을 동결(freezing)한다고 말하고 우리가 만든 구조를 **값 매김 묶음(thunk or promise)**이라고 부를 것이다.

동결되어 있는 값매김 묶음의 실제 값을 알 필요가 있을 때, 이것을 "녹여야(thaw)" 한다. 즉, 미루어진 값매김을 수행해야 한다. 그러기 위해서는 값매김 장치를 다시 불러서, 그 장치에게 환경과 표현식을 넘겨주어야 한다.

어떠한 종류의 값매김이 미루어져야 하는가?

- 프로그램의 인자
- 주어진 자료 구조의 구성 요소

언제 미루어진 값매김을 수행해야 하는가? 해당하는 값에 접근하려고 할 때 값매김을 수행하면 된다. 자료 구조의 구성 요소나 함수의 인자에 접근하려는 시도(즉, 현재 환경에 있는 변수의 값에 접근하려는 시도)를, 해당하는 값을 필요로 하는 것이라고 간주할 수 있다.

지난번에 언급했던 값매김 함수의 근본 원리는 다음과 같다. 표현식으로 불려지면, 이것은 표현식의 값이 필요로 하다는 것을 의미한다. 값매김을 할 인자가 함수를 적용하는 것 (e_1 e_2)이라면, e_1이 필요하게 되었기 때문에 e_1을 값매김하기 시작한다. 이 값매김을 통해 함수 묶음을 얻으면, 표현식 e_2의 값매김 묶음이 추가된 환경에서 이 함수 묶음의 몸체가 값매김된다. 함수 묶음의 몸체가 값매김 되는 동안에 e_2의 값은 필요하게 될지도 (아닐지도) 모른다. 오직 필요하게 될 때에만 미뤄놓았던 e_2의 값매김을 할 것이다.

그러므로 이 값매김 함수는 절대로 얼려진 값을 반환하지 않는다. 물론, 얼려진 구성요소를 갖는 값은 반환할 수 있다. (예를 들어 자료 구조가 있다.)

만약 자료 구조를 사용하지 않는다면 미루어진 값매김은 그리 유용하지는 않다. 이것을 이용하기 시작한다면, 우리는 무한의 자료 구조를 사용할 가능성을 얻게 된다. 그 점을 염두에 두고 우리는 리스트로 계산할 수 있도록 언어를 확장해 나갈 것이다.

우선 생성자에 해당하는 타입 `ml_constructor`를 소개한다. 지금은 리스트의 생성자를 다루는 것에 만족하지만, 나중에 소개할 값매김 함수는 어떠한 생성자도 받아들일 것이다.

```
# type ml_constructor = NilList | Cons;;
```

다음으로 패턴매칭에 의해 정의된 함수(생성자 `Ml_func`)와 생성자의 적용에 의해 정의된 함수(생성자 `Ml_capp`)를 다룰 수 있도록 타입 `ml_exp`를 확장할 것이다. 또한 값에 생성자를 적용하는 것(생성자 `Constr`)과 특히 어떤 표현식은 얼려질 수 있다는 사실을 다룰 수 있도록 타입 `value`를 확장할 것이다. 이를

위해 Clo와 비슷한 생성자 Fre를 정의한다. 이것은 값매김 환경을 따라 값매김이 미루어진 표현식을 저장한다.

```
# type ml_exp =
   Ml_int_const of int                                              (* 정수 상수 *)
 | Ml_bool_const of bool                                             (* 불 상수 *)
 | Ml_pair of ml_exp * ml_exp                                           (* 쌍 *)
 | Ml_unop of ml_unop * ml_exp                                      (* 단항 연산 *)
 | Ml_binop of ml_binop * ml_exp * ml_exp                           (* 이항 연산 *)
 | Ml_var of string                                                     (* 변수 *)
 | Ml_constr0 of ml_constructor                                    (* 영 생성자*)
 | Ml_if of ml_exp * ml_exp * ml_exp                                 (* 조건문 *)
 | Ml_fun of string * ml_exp                                           (* 함수 *)
 | Ml_func of (ml_constructor * string list * ml_exp) list  (* 패턴매칭 함수 *)
 | Ml_app of ml_exp * ml_exp                                           (* 적용 *)
 | Ml_capp of ml_constructor * ml_exp list                    (* 생성자 적용 *)
 | Ml_let of string * ml_exp * ml_exp                                  (* 선언 *)
 | Ml_letrec of string * ml_exp * ml_exp                        (* 재귀 선언*)
;;
```

```
# type value =
         Int_Const of int
      | Bool_Const of bool
      | Pair of value * value
      | Clo of (string * value) list * ml_exp   (* 함수 묶음 *)
      | Fre of (string * value) list * ml_exp   (* 값매김 묶음 *)
      | Constr0 of ml_constructor
      | Constr_n of ml_constructor * value list (* 자료 구조 *)
;;
```

이 문법에서 사용된 패턴은 $C(x_1,\cdots,x_n)$의 형태로 제한 되어 있다. 여기서 n은 생성자 C의 차수이고 x_i는 변수들이다. 여기 이 문법을 이용한 몇가지 예가 있다.

```
# exp_of_string "Cons(x,l)";;
val it:  Ml_exp = Ml_capp (Cons, [Ml_var "x", Ml_var "l"])
```

```
# exp_of_string "fn NilList ⇒ NilList | (Cons(x,l)) ⇒ x";;
val it:  Ml_exp = Ml_func [(Nil, [], Ml_constr0 Nil),
                  (Cons, ["x", "l"], Ml_var "x")]
```

select라는 추가 함수는 매칭할 패턴에서 옳은 것을 선택한다. combine이라는 추가 함수는 패턴 매칭에 의해 생성된 묶음들 모두를 환경에 추가할 것이다.

```
# val rec select = fn p x ⇒ case x of
        (a::l) ⇒ if p a then a else select p l
      | _ ⇒ failwith "select";;
val select:  ('a -> bool) -> 'a list -> 'a = <fun>
```

```
# val rec combine = fn x ⇒ case x of
        ([],[]) ⇒ []
      | (a::l,a'::l') ⇒ (a,a')::combine(l,l')
      | _ ⇒ failwith "combine";;
val combine:  'a list * 'b list -> ('a * 'b) list = <fun>
```

새로운 값매김 함수는 얼려진 값을 녹여주는 함수 unfreeze와 함께 수행된다.

```
# val rec ml_eval = fn env x ⇒ case x of
    (Ml_int_const n) ⇒ Int_Const n
  | (Ml_bool_const b) ⇒ Bool_Const b
  | (Ml_pair (e1,e2)) ⇒ Pair(Fre(env,e1), Fre(env,e2))
  | (Ml_unop (op,e)) ⇒(case (op, ml_eval env e) of
                (Ml_fst, Pair(v1,v2)) ⇒ unfreeze v1
              | (Ml_snd, Pair(v1,v2)) ⇒ unfreeze v2
              | _ ⇒ failwith "ml_eval:  wrong types")
  | (Ml_binop (op, e1, e2)) ⇒ let val v1 = ml_eval env e1
                       val v2 = ml_eval env e2 in
                       (ml_eval_binop op) v1 v2 end
  | (Ml_var x) ⇒ unfreeze(assoc x env)
  | (Ml_constr0 c) ⇒ Constr0 c
  | (Ml_if (c,e1,e2)) ⇒ (case ml_eval env c of
                (Bool_Const true) ⇒ ml_eval env e1
              | (Bool_Const false) ⇒ ml_eval env e2
              | _ ⇒ failwith "ml_eval:  wrong types")
  | (Ml_fun body) ⇒ Clo(env,Ml_fun body)
  | (Ml_func body) ⇒ Clo(env,Ml_func body)
  | (Ml_app (e1,e2))
    ⇒ (case ml_eval env e1 of
            (Clo(env',Ml_fun (x,e)))
              ⇒ ml_eval ((x,Fre(env,e2))::env') e
          | (Clo(env',Ml_func case_list))
              ⇒ let val (c,vl) = case ml_eval env e2 of
                      (Constr_n(c,vl)) ⇒ (c,vl)
                    | (Constr0 c) ⇒ (c,[])
                    | _ ⇒ failwith "ml_eval:  wrong types" in
                      let val (c',sl,e) =
                         select (fn (c',sl,e) ⇒ (c'=c)) case_list
                      in ml_eval (combine(sl,vl)@env') e end end
          | _ ⇒ failwith "ml_eval:  wrong types")
  | (Ml_capp (c,el)) ⇒ Constr_n(c, map (fn e ⇒ Fre(env,e)) el)
  | (Ml_let (x,e1,e2)) ⇒ ml_eval ((x,Fre(env,e1))::env) e2
  | (Ml_letrec (f,e1,e2)) ⇒ let val rec env' = (f,Fre(env',e1))::env in
              ml_eval env' e2 end
  and unfreeze = fn
```

```
    (Fre(env',e)) ⇒ ml_eval env' e
  | v ⇒ v
val ml_eval:  (string * value) list -> ml_exp -> value = <fun>
val unfreeze:  value -> value = <fun>
```

이제 지금까지 해온 것들을 설명할 몇가지 예를 알아보자. 첫번째 예는 간단한 계승의 계산이다. 이 후의 3개의 예는 우리가 방금 정의한 값매김 장치가 필요없는 계산을 하지 않는다는 사실을 보여준다. 특히 마지막 예는 값에 의한 호출 방식으로는 끝나지 않고 무한 루프에 빠질 것이다.

또한 새로운 값매김 장치는 무한 길이의 배열를 다룰 수 있다. 다음의 예는 3과 7의 곱셈없이 무한한 길이의 순서 배열를 만든다. 이 배열의 시작을 보이기 위해, nML로 추가함수인 `nfirst`를 정의할 것이다. 이것은 `value` 형식의 리스트의 처음 n개의 원소를 nML의 방식으로 만든다.

미루어진 값매김을 효율적으로 구현하기 위해서, 변수가 한번 이상 나타날 수 있으므로 같은 값을 여러번 계산하는 문제를 피하기 위해 값매김 장치를 수정할 필요가 있다. 사실상 환경에서 변수를 접근할 때마다 값매김 묶음을 녹이면 같은 값매김 묶음에 여러번 접근을 시도할 때마다 똑같은 값매김을 수행해야 한다.

우리가 해야하는 것은 모든 얼려진 값들이 필요할 때 재사용될 수 있도록(다시 계산되는 것이 아니라), 얼려진 값들이 한번 계산되면 이 계산된 값으로 체계적으로 치환되도록 하는 것이다. 지금까지 살펴본 값매김 장치에 이점을 고려하도록 수정하는 간단한 방법은, 환경과 값을 실제로 수정될 수 있는 구조로 표현하는 것이다. 이런 방식으로 구현하는 것은 연습문제로 남겨둔다.

반복 계산을 피하기 위해 이런 방법으로 미루어진 값매김을 구현할 때, 이것을 **늦은 값매김**(lazy evaluation)이라고 부른다.

11.4 요약

우리는 nML로 쓰여진 2개의 nML 값매김 장치를 정의하였다. 이 값매김 장치들은 **환경**이라는 아이디어를 이용한다. 함수를 부르는 동안이나 `let`의 값매김을 하는 동안에 함수의 형식 인자나 `let`에 의해 정의된 변수는 글자 그대로 바뀌지는 않지만 해당하는 묶음이 환경이라고 불리는 표에 저장된다.

함수 값은 함수 묶음의 형태, 즉, 함수의 내용과 환경으로 이루어진 쌍으로 표현된다.

11.5 더 배울 내용들

1965년 Landin은 환경과 함수 묶음이라는 아이디어를 이용해 함수형 언어를 값매김 하는 생각을 소개하였다. 이 기법은 특히 우리가 다음 장에서 다룰 컴파일러를 만드는 데에 유용하다. 다음 장의 마지막에서 우리는 함수형 언어를 구현하는 다양한 기법에 대해서 살펴볼 것이다.

컴파일러와 비교하면, 우리가 이번에 정의한 값매김 장치는 본질적인 비효율성 때문에 고민하게 된다. 사실 이러한 값매김 장치는 어떤 행동을 할지 결

정하기 위해 값매김 대상 프로그램의 구조를 반복적으로 분석한다. 반면에 컴파일러에서는 이런 분석이 실행 전에 단 한번 이루어진다. 그럼에도 불구하고 값매김 장치를 사용하는 것은 때때로 유익하다. 이것은 컴파일러가 애매한 방식으로 이용하는 값매김 과정을 이해하는 데에 도움을 준다.

12 장

컴파일

이번 장에서 nML과 같은 함수형 언어를 기계어로 컴파일하는 방식에 대해 살펴본다. 이 컴파일 방식은 이전 장에서 공부한 값매김 기법과 깊은 관계가 있다. 특히, 환경이란 아이디어가 이번에도 중요한 역할을 수행하고, 함수값을 표현하기 위해 함수 묶음의 아이디어를 계속 사용한다. 그렇지만 이 두가지 아이디어가 컴파일러에서는 약간 다르게 대응한다는 것을 알게 될 것이다.

컴파일러를 충분히 개념적인 수준으로 유지하기 위해서, 우리의 컴파일 계획에 잘 맞는 명령어로 이루어진 기계어를 사용할 것이다. 이 명령어들은 실제 어셈블리 언어의 명령어들보다 좀더 복잡한 동작을 한다. 그렇다고 하더라도 이 명령어들이 같은 효과를 갖는 기계어들로 확장될 수 있다는 것은 분명하다. 그러므로 우리가 앞으로 보일 컴파일 계획은 실제 컴파일러로 우리를 이끌어 줄 것이다.

우리의 컴파일러로 만들어진 코드를 수행하기 위해 계산에 필요한 중간값들을 저장하는 스택을 사용한다. 또한 전통적인 방식대로 스택에 부 프로그램(subprograms)의 주소를 저장한다. 함수형언어를 컴파일할 때 특이한 것은 구조화된 값, 환경, 그리고 함수 묶음을 만드는 데 필요한 메모리를 할당하는 명령어들이다. 이 명령어들과 함수형언어를 컴파일할 때 이것들의 역할을 설명하기 위해서, 우선 어떻게 우리가 복잡한 객체를 표현하기 위해 컴퓨터 메모리를 사용하는지을 설명해야 한다. 이런 이유로 이번 장은 맨 처음 절에서 자료 구조를 표현하는 방법에 대해서 다룬다. 그리고 나서 12.3절에서 기계어와 코드 생성을 공부한다. 12.4절에서는 코드 시뮬레이터로 nML의 컴파일러를 만드는 것으로 마무리 짓는다.

12.1 간단한 컴퓨터 메모리 모델

컴퓨터 메모리는 본질적으로 기억 소자로 이루어진 큰 순차 배열이다. 이 기억 소자들은 0부터 시작하여 차례로 숫자가 매겨져 있다. 이 숫자들이 주소이다.각각의 기억 소자는 일반적으로 32비트의 정보를 보관한다. 이 모델이 그림 12.1에 나타나있다.

각각의 메모리 기억 소자에 저장된 정보는 여러가지 방식으로 해독할 수 있다. 정수, 부동소수점 수, 4개의 문자 집합, 또는 심지어 주소나 기계어로 읽을

수 있다.

일반적으로 사용자는 컴퓨터의 물리적 메모리에 직접 접근하지 않고, 시스템에 제공하는 기본적인 명령어를 통해 메모리를 사용한다. 특히 컴파일된 프로그램을 실행하는 것은 컴퓨터가 가상 메모리 공간을 갖는 프로세스를 만들도록 한다. 한 순간에 이 가상 메모리는 주 메모리와 디스크에 나누어져 있다. 하지만 실행 프로그램은 주 메모리에 직접 접근함에도 불구하고 모든것은 잘 동작한다.

프로세스에 할당된 메모리는 개념적으로 영역으로 나누어진다. 여기에 우리가 관심을 갖는 세개의 영역이 있다. **코드 영역**은 명령어들을 가지고 있고, **메모리영역**은 (엄밀히 말하여) 프로그램에 의해 다루어지는 데이터를 가지고 있으며(이 영역은 때때로 힙이라 불린다), **스택 영역**은 어떻게 계산이 일어나는 지에 대한 정보를 갖는다. 추가로 우리는 **레지스터**라 알려진 특별한 세개의 메모리 소자가 있다고 가정한다. 코드 레지스터(CR)은 코드 영역의 한개의 주소(현재 명령어)를 가지고 있다. 메모리 레지스터(MR)은 메모리 영역의 주소(비어있는 첫번째 메모리 기억소자)를 저장한다. 스택 레지스터(SR)은 스택의 주소(스택의 꼭대기)를 저장한다. 이런 종류의 구성을 그림 12.2에서 볼 수 있다.

기본 동작에 해당하는 명령어들은 스택으로부터 인자를 받고, 역시 스택을 통해 결과를 반환한다.

12.2 자료구조의 구현

설명을 간단히 하기 위해, 다음과 같은 종류의 자료로 제한하기로 한다.

- 단일 메모리 기억 소자로 표현될 수 있는 값(정수, 부동소수점 수, 불)
- 쌍
- 리스트

12.2.1 자료 표현하기

이런 객체를 표현하는 기본 생성자는 **이중항(doublet)** 즉, 두개의 연속된 메모리 소자이다. 이중항은 항상 메모리영역에 위치하고, 이것은 실제 값이나 주소를 포함한다. 주소는 이중항를 연결하도록 하여 복잡한 자료 구조를 만들 수 있도록 한다.

이번 논의에서 우리는 메모리의 내용을 위해 다음과 같은 표기법을 사용할 것이다.

- `n` 직접적으로 표현될 수 있는 값
- `#n` 주소
- `/` 빈 리스트를 나타내는 특별한 값

그림 12.3은 리스트[1,2,3](1::2::3::[]로 표현할 수도 있다.)를 메모리에 나타내는 두가지 방식을 보여준다.

메모리에 나타나는 리스트 원소들의 순서는 실제 리스트에서 그들의 순서와 무관하다. 포인터(주소)가 **서로 연결된** 방식이 리스트를 정의한다.

우리가 객체를 메모리에 표현하는 것을 이것의 메모리 기억소자에 숫자를 매기는 방식과는 독립적인 상(image)으로 보기를 원할 때, 그림 12.4에 있는 것과 같은 방식을 이용할 것이다.

12.2.2 연산자 CONS

nML프로그램이 (e$_1$,e$_2$)나 (e$_1$::e$_2$)의 형태를 갖는 표현식을 포함하고 있는 경우, 해당하는 기계어 코드는 반드시 표현식 e$_1$과 e$_2$의 값을 계산하여야 하고, 처음 기억소자는 e$_1$의 값(의 주소)를, 두번째 기억소자는 e$_2$의 값(의 주소)를 저장하는 이중항을 할당해야 한다. 이런 연산을 CONS라고 부른다. 우리는 e$_1$의 값이 스택의 꼭대기(STACK[SR])에서 찾을 수 있고, e$_2$의 값은 그 바로 아래(STACK[SR-1])에서 찾을 수 있다고 가정한다. 연산으로써 CONS의 효과는 다음과 같은 연속된 명령어로 정의될 수 있다.

```
MEM[MR] ← STACK[SR];
MEM[MR+1] ← STACK[SR-1];
STACK[SR-1] ← MR;
MR ← MR+2;
SR ← SR-1
```

(우리는 모든 자유 이중항이 주소 MR이후에 위치해 있다고 가정한다. 그러므로 주소 MR의 이중항이 사용되었을 때, 새로운 첫 자유 이중항는 주소 MR+2이다.)

그림 12.5는 예로서 이 연산자를 설명한다.

12.2.3 자료 공유하기

nML표현식에서 구조화되어 있는 값이 참조될 때, 이것의 주소는 표현식의 컴파일된 코드에서 이 값을 가리키기 위해 사용된다. 따라서 값은 항상 메모리에서 하나의 단일한 워드의 내용에 의해 참조된다.

복잡한 값을 이용해 새로운 값을 만들때, 오직 이것의 주소만이 복사된다. 예를 들어, 변수 x의 값 v가 주소 #n에 있다면, 표현식 (x,x)의 계산은 값 v를 복사하지 않는다. 대신 이중항의 두 요소가 주소 #n를 갖는 이중항을 만들어 낸다. (x,x)의 값을 표현할 때 값 v는 그림 12.6에서처럼 이 쌍의 두 요소에 의해 **공유**된다.

일반적으로 값의 구조를 표현하는 것은 나무가 아니라 그래프이다.(대개 재귀적인 값을 제외하고는 사이클이 존재 하지 않는다.) 심지어 표현식에 같은 변수가 여러번 나타나지 않을 때에도, 값매김을 할 때 계산중인 값과 이미 존재하고 있는 값 사이를 공유할지도 모른다. 예를 들어 표현식 (l$_1$@l$_2$)의 값매김은 두개의 리스트 l$_1$, l$_2$를 이어 붙인다. 이를 위해 l$_1$의 요소의 주소와 l$_2$의 주소를 포함하는 구조를 만든다.

그림 12.7는 이런 자료 공유의 결과를 보여준다. 표현식 (l$_1$@l$_2$)의 값매김은 l$_1$와 같은 길이의 이중항을 할당하지만 이 리스트들의 원소를 복잡하게 만든다.

nML자체의 수준에서 당신은 연산자 ==을 사용할 때 값의 표현에서 공유가 일어남을 알 수 있다. (논리적 값들 사이의 동등함을 계산하는) 연산자 = 에 비해서, ==는 물리적 동등성을 검사한다. 즉, 이것은 인자가 포함하는 메모리 기억소자의 내용이 같은지를 검사한다. 표현식 e_1==e_2는 e_1과 e_2의 값이 같은 실제값이거나 이것이 똑같은 구조화 값을 갖는다면(즉, 같은 주소에 위치해있다면) 참이다.

여기에 이제까지 살펴본 것을 잘 보여주는 몇가지 예가 있다.

```
# val x=(1,2);;
val x:  int * int = (1, 2)
# val y=(x,x);;
val y:  (int * int) * (int * int) = ((1, 2), (1, 2))
# val z=((1,2),(1,2));;
val z:  (int * int) * (int * int) = ((1, 2), (1, 2))
# fst y == snd y;;
val it:  bool = true
# fst z == snd z;;
val it:  bool = false
# fst z = snd z;;
val it:  bool = true
```

12.2.4 다른 자료 타입들 표현하기

이중항은 생성자를 가진 타입의 값들을 표현하는 데에 사용될 수도 있다. 만약 값 생성자 C에 값 v를 적용하여 만들어진 (C v)를 생각한다면, 이중항의 첫번째 원소는 기호화된 C를 포함할 것이고 두번째 원소는 값 v(의 주소)를 가질 것이다.

이 표현은 여러가지 방식으로 최적화 될 수 있다. 예를 들어 만약 타입이 오직 한개의 생성자를 갖는다면, 이것을 표현할 필요가 없다.[1]

n-튜플, 레코드, 그리고 벡터을 표현하는 것은 이중항만으로는 쉽게 되지 않는다. 우리는다른 연속된 메모리 소자를 할당할 수 있어야 한다.

실제로 벡터와 문자열은 객체의 크기가 명시된 특별한 구조를 필요로 한다.

12.2.5 할당된 메모리 재활용하기

계산은 임시적으로만 쓰이는 많은 중간값들을 빈번히 사용한다. 그들은 한번 사용되면 없어질 수 있다. 그렇지만 프로그래머가 정확히 어떤 값이 제거되고 어떤 값이 저장되어야 하는지를 결정하는 것은 항상 가능하지 않다. 이 문제는 일반적으로 부결정적(undecidable)이다.

함수형 언어가 채택한 관점은 프로그래머는 프로그래머가 만든 자료구조가 실제로 얼마나 오래동안 살아있어야 할지 걱정해서는 안된다고 보는 것이다. 그렇지만 어떤 주어진 프로그램에서 사용 가능한 메모리의 크기는 분명히 유한하기 땜누에, 만약 메모리가 재활용되지 않고 할당만 한다면 메모리가 모두

[1] 실제로는 많은 생성자의 경우, 그 생성자가 영향을 미치는 값의 표현을 이용한다면 이 생성자는 제거할 수 있다.

사용되어 새로운 값이 만들어질 수 없는 때가 올 것이다. 이런 이유로 우리는 필요없는 구조가 차지하고 있는 메모리를 재활용할 수단을 제공해야만 한다. 이 수단은 **메모리 재활용**(Garbage collector), 줄여서 GC라고 알려져 있다.

실제로 사용중인 메모리 재활용 방식은 여러가지가 있다. 일반적으로 이것들은 두가지 기본적인 방식의 변형이거나 조합이다.

- 필요없는 메모리의 복구를 이용한 메모리 재활용
- 사용중인 메모리의 복사를 이용한 메모리 재활용

이 두가지 종류의 메모리 재활용 과정에서, 실행 스택이나 전역 변수가 저장되어 있는 기호표(symbol table)를 이용하여 값을 접근하는 모든 포인터를 따라가면서 현재 사용하고 있는 구조를 결정해야 한다.

메모리 재활용이 필요없는 메모리를 복구할 때, 아직 사용되지 않은 메모리는 이중항의 리스트로 표현된다. 이런 경우에, 이중항을 할당하는 수단은 175쪽에서 살펴본 것과 조금 다르다. 레지스터 MR은 이용할 수 있는 이중항의 리스트의 첫부분을 가리킨다. 그리고 CONS 연산을 한 후에 다음의 이중항을 가리켜야만 한다. 만약 이 리스트가 비게 되면, 메모리 재활용이 전체 메모리 영역을 두번 살펴봐야 한다. 첫번째 살펴볼 때는 다음을 가리키는 포인터를 이용해 사용하고 있는 이중항을 찾아서 표시하고, 두번째 살펴볼 때 사용 가능한 이중항의 리스트를 만들기 위해 표시 안된 이중항들을 모은다. 그리고 나서 일반 계산을 다시 수행한다.

메모리 재활용이 사용하고 있는 메모리를 복사하는 경우는, 메모리를 두개로 나누어 번갈아 사용한다. 활동중인 영역에서 사용되지 않은 메모리는 순차적으로 할당된 연속된 이중항으로 표현된다. 만약 활동중인 영역이 가득차면, 사용중인 이중항이 연속적으로 두번째 영역으로 복사된다. 두번째 영역이 이제 활동중인 영역이 되고, 아직 차지되지 않은 부분이 할당을 위해 사용된다.

이론적으로 사용중인 메모리를 복사하는 메모리 재활용가 더 빠르다. 이것은 이 방식은 오직 사용중인 메모리만을 방문하지만, 불필요한 메모리를 복구하는 방식은 전체 메모리 영역을 살펴보기 때문이다. 그리고 추가적인 이점이 있는데, 이것은 자료를 조밀하게 잘 위치하여 계산중에 디스크와 메모리에서 이동량을 줄인다.그렇지만 이 방식은 사용가능한 메모리를 두개로 나누기 때문에 모든 메모리를 사용하지 못한다는 점이 단점이다.

살펴보지는 않겟지만, 실제로 메모리 재활용의 효율은 복잡한 개선방안들에 달려있다.

연습 문제

12.1 105쪽에서 정의한 타입 `'a gentree`를 생각해보자. 타입 gentree의 트리 크기는 정의해 의해 노드의 숫자이고, 이것은 106쪽에서 정의한 함수 `gentree_size`에 의해 정의된다. 그렇지만 이런 트리가 실제 기계에 저장되어있는 것의 크기는 몇몇 노드가 공유될 수 있기 때문에 이론적인 값과 다를 수 있다. 그들이 가지고 있는 물리적으로 다른 노드들의 개수, 즉, predicate `==`로 서로 구별되는 노드들의 개수를 이런 트리의 "실제 크기"라고 하자. 타입 `gentree`의 실제 크기를 계산하는 함수를 작성하시오.

12.2 매우 큰 트리를 다룰 때, 같은 부 트리를 공유함으로서 트리표현의 크기를 줄이는 것은 도움이 된다. 주어진 트리 `t`에 대해서 모든 같은 부 트리를 공유하도록 표현되는 똑같은 트리를 만들어 내는 타입 `'a gentree` →`'a gentree`의

함수를 작성하시오. 다시 말해, 원래 트리 t의 모든 부 트리의 쌍 (t',t'')에 대해서, 만약 t'==t''이면 t'=t''이고 반대방향도 성립하도록 한다. 예를 들어, 맞춤법 검사를 위해 트리를 만들 때, 이런 방식으로 트리를 줄이는 것은 매우 유용하다.

12.3 코드 생성하기

우리는 실제 기계어는 아니지만 간단히 기계어로 번역될 수 있을 정도로 충분히 간단한 명령어들의 집합으로 컴파일 코드를 사용할 것이다. 몇개는 상수나 주소를 인자로 갖는다.

몇몇 명령어들은 인자를 가지기 때문에 모든 명령어들이 메모리의 단일 워드에 딱 맞는다거나, 이들이 메모리에서 같은 수의 워드를 차지한다고 가정할 수 없다. 따라서 다음 명령어를 얻기 위해 코드 레지스터 CR을 증가시키는 방식은 현재 실행 중인 명령어의 크기에 달려있다.

여러가지 명령어를 표현하는 방식을 간단히 하기 위하여, 우리는 프로그램의 실행순서를 바꾸지 않는 명령어들에 대해 CR을 명시적으로 바꾸지는 않을 것이다. 그렇지만 물론 CALL이나 RETURN과 같이 명령어의 목적이 프로그램의 실행 순서를 바꾸는 데에 있는 명령어들에 대해서는 명시적으로 CR을 수정할 것이다.

모든 명령어는 메모리 소자의 내용을 옮기는 것들의 집합으로 정의된다. 이런 옮김은 병렬적으로 일어나야만 하고 기호로는 ''‖''로 표시한다.

12.3.1 명령어 리스트

기본 연산자

```
PLUSTACK[SP-1] ← (STACK[SR]+STACK[SP-1]) ‖ SR ← SR-1
MUL$TACK[SP-1] ← (STACK[SR]*STACK[SP-1]) ‖ SR ← SR-1
```

명령어 PLUS는 스택의 꼭대기에 있는 두개의 원소를 그들의 합으로 변경한다. 명령어 MULT는 스택의 꼭대기에 있는 두개의 원소를 그들의 곱으로 변경한다. 우리는 모든 이항 산술 연잔자들이 같은 방식으로 동작한다고 가정한다.

스택 연산자

```
LOAD v   STACK[SR] ← v
PUSH v   STACK[SR+1] ← v ‖ SR ← SR + 1
DUPL     STACK[SR+1] ← STACK[SR] ‖ SR ← SR+1
SWAP     STACK[SR] ← STACK[SR-1] ‖ STACK[SR-1] ← STACK[SR]
ROT3     STACK[SR] ← STACK[SR-1] ‖ STACK[SR-1] ← STACK[SR-2]
         ‖ STACK[SR-2] ← STACK[SR]
IROT3    STACK[SR] ← STACK[SR-2] ‖ STACK[SR-1] ← STACK[SR]
         ‖ STACK[SR] ← STACK[SR-1]
```

명령어 LOAD v는 스택의 꼭대기를 v로 바꾼다.

명령어 PUSH v는 스택에 값 v를 추가한다.

명령어 DUPL은 스택의 꼭대기에 있는 값을 복사하여 스택에 추가한다.

명령어 SWAP은 스택의 꼭대기에 있는 값과 바로 아래에 있는 값을 바꾼다.

명령어 ROT3는 스택 위의 세 개의 원소를 순환하여 바꾼다.

명령어 IROT3는 스택 위의 세 개의 원소를 반대 방향으로 순환하여 바꾼다.

자료 구조 명령어

```
FST      STACK[SR] ← MEM[STACK[SR]]
SND      STACK[SR] ← MEM[STACK[SR]+1]
SETFST   MEM[STACK[SR-1]] ← STACK[SR] ‖ SR ← SR-1
SETSND   MEM[STACK[SR-1]+1] ← STACK[SR] ‖ SR ← SR-1
CONS     MEM[MR] ← STACK[SR] ‖ MEM[MR+1] ← STACK[SR-1]
         ‖ STACK[SR-1] ← MR ‖ MR ← MR+2 ‖ SR ← SR-1
SPLIT    STACK[SR+1] ← MEM[STACK[SR]]
         ‖ STACK[SR] ← MEM[STACK[SR]+1]
         ‖ SR ← SR + 1
```

명령어 FST가 실행되려면 스택의 꼭대기에는 이중항의 주소가 들어가 있어야 한다. 이 주소 값이 이중항의 첫번째 요소로 바뀌어진다.

명령어 SND가 실행되려면 스택의 꼭대기에는 이중항의 주소가 들어가 있어야 한다. 이 주소 값이 이중항의 두번째 요소로 바뀌어진다.

명령어 SETFST가 실행되려면 스택의 위에서 두번째 원소에 이중항의 주소가 들어 있어야 한다. 이 이중항의 첫번째 요소는 스택의 꼭대기에 위치한 값으로 바뀌고, 스택에서 원소를 한 개 빼낸다.

명령어 SETSND가 실행되려면 스택의 위에서 두번째 원소에 이중항의 주소가 들어 있어야 한다. 이 이중항의 두번째 요소는 스택의 꼭대기에 위치한 값으로 바뀌고, 스택에서 원소를 한 개 빼낸다.

명령어 CONS는 175쪽에서 설명한 연산을 수행한다.

명령어 SPLIT는 CONS의 반대 작용을 한다. 이것은 스택의 꼭대기에 이중항의 주소가 있다고 가정하고, 스택의 꼭대기를 이중항의 두개의 원소로 바꾼다.

분기 명령어

```
CALL           STACK[SR] ← STACK[SR-1] ‖ STACK[SR-1] ← CR + 1
               ‖ CR ← STACK[SR]
RETURN         CR ← STACK[SR-1] ‖ STACK[SR-1] ← STACK[SR]
               ‖ SR ← SR -1
BRANCH a1 a2   STACK[SR] ← if STACK[SR] then a1 else a2
```

명령어 CALL은 부 프로그램을 실행하기 위해 코드 레지스터의 값을 변경한다. 그리고 부 프로그램이 실행되면 원래 프로그램으로 돌아가기 위해 돌아올

주소를 저장한다. CALL이 실행될때마다, 스택의 꼭대기는 부 프로그램의 주소를 저장하고 있고, 그 아래 원소는 부 프로그램에 필요한 값(인자)를 가지고 있다고 가정한다. 부 프로그램으로 제어를 넘기기 전에, 이 인자를 스택의 꼭대기에 넣고, 돌아올 주소를 그 아래에 넣는다.(즉, 코드 레지스터의 값은 이미 증가되었다.)

명령어 RETURN은 부 프로그램을 부른 프로그램으로 돌아올 수 있도록 부 프로그램의 끝에 위치한다. 우리는 부 프로그램이 끝날 때 계산 결과값이 스택의 꼭대기에 있다는 것과 돌아올 주소가 바로 그 아래에 있다는 것을 가정한다.

명령어 BRANCH a1 a2는 스택의 꼭대기가 bool이라고 가정한다. 만약 이것이 참이라면 주소 a1으로 바뀌어질 것이고, 그렇지 않다면 주소 a2로 바뀔 것이다.

12.3.2 컴파일의 원리

자유 변수 $x_1,\cdots,x_n$을 포함한 표현식 e를 계산하는 것은 이 자유변수들을 포함하는 리스트인 환경 $[v_1,\cdots,v_n]$에서 일어난다. 여기에서 변수의 이름을 유지하는 것은 필요하지 않다. 왜냐하면 이들에 접근하기 위한 정보는 컴파일된 코드에 들어가 있기 때문이다. 우리는 $[x_1,\cdots,x_n]$을 **컴파일 환경**이라 하고, $[v_1,\cdots,v_n]$을 **실행 환경**이라고 부를 것이다.

실행 환경은 메모리에 표현된다. 그리고 우리는 실행을 시작할 때 이 환경의 주소가 스택의 꼭대기에 있다고 가정한다. 모든 표현식이 값매김 될때 이 성질이 성립하도록 컴파일된 코드가 만들어져 있다. 게다가, 이 코드는 스택의 나머지 부분은 건드리지 않고, 스택의 꼭대기에 있는 환경을 계산된 표현식의 값으로 바꾸는 전역적인 효과를 가진다.

이전에 보았던 값매김 방법에서와 같이 함수는 함수 묶음으로 표현된다. 즉, 환경과 코드의 묶음이다. 차이점은 환경과 코드가 표현되는 방식에 있다. 이제부터 "코드"는 코드 영역의 주소이고, 환경은 단순한 값의 리스트이다.

우리는 환경 $[x_1,\cdots,x_n]$에서 컴파일된 표현식 e의 코드를 $[[e]]_{(x_1,\cdots,x_n)}$으로 표시한다. 그러므로 자유변수가 없는 표현식 e에 대해서, 컴파일된 코드는 $[[e]]_{[]}$이 될 것이다.

상수 컴파일하기

$[[c]]_E$ = LOAD c

환경 E에서 상수 c의 코드는 단순히 스택의 꼭대기에 있는 환경 E를 상수 c로 바꿀 것이다.

변수 컴파일하기

$[[x]]_{(x::E)}$ = FST $[[x]]_{(y::E)}$ = SND; $[[X]]_E$

환경 $[x_1,\cdots,x_n]$에서 변수 x_i에 해당하는 코드는 $\text{SND}^{(i-1)}$; FST이다. 이것은 실행환경에서 변수의 값에 접근하도록 한다.

쌍 컴파일하기

`[[(e1,e2)]`$_E$ `= DUPL; [[e2]]`$_E$`; SWAP; [[e1]]`$_E$`; CONS`

환경 E에서 표현식 (e_1,e_2)에 해당하는 코드는 환경을 두번 이용하기 위해서(한번은 e_2를 계산할 때, 다른 한번은 e_1을 계산할 때 사용한다.) 환경(의 주소)를 복사한다. e_2에 해당하는 코드를 실행한 후에 e_2의 값 v_2는 스택의 꼭대기에, 환경 E(의 주소)의 두번째 복사본은 그 아래에 저장되어 있을 것이다. 그리고 나서 명령어 SWAP은 v_2와 E를 바꾸어 e_1에 해당하는 코드에 올바르게 실행되도록 한다. 이것은 스택의 꼭대기에 값 v_1이, 그 아래에 값 v_2이 저장되도록 한다. 남은 할일은 명령어 CONS를 이용하여 두개의 값이 쌍을 이루도록 하는 것이다.

산술 표현식 컴파일하기

`[[`e_1`+`e_2`]]`$_E$ `= DUPL; [[e2]]`$_E$`; SWAP; [[`e_1`]]`$_E$`; PLUS`

환경 E에서 표현식 (e_1+e_2)에 해당하는 코드는 e_2계산에서 한번, e_1계산에서 한번, 이렇게 두번 환경을 이용하기 위해서 환경(의 주소)를 복사한다. e_2에 해당하는 코드가 실행하고 난 후에 e_2의 값 v_2가 스택의 꼭대기에, 환경 E(의 주소)의 두번째 복사본은 그 아래에 저장되어 있을 것이다. 그리고 나서 명령어 SWAP은 e_1에 해당하는 코드가 바르게 실행되도록 v_2와 E를 바꾼다. 그리고 나서 값 v_1이 계산되어 스택의 꼭대기에 저장된다. 값 v_2는 그 아래에 있다. 남은 할일은 명령어 PLUS를 이 두 값에 적용하는 것이다.

함수 컴파일하기

`[[fn x ⇒e]]`$_E$ `= PUSH a; SWAP; CONS`
여기서 a는 함수 코드, 즉, `[[e]]`$_{(x::E)}$`; RETURN`의 주소이다.

단, a는 함수 코드의 주소이다. 즉, `[[e]]`$_{(x::E)}$`; RETURN`이다. 형식 환경 E에서 함수 (fn x ⇒e)의 몸체에 대해 생성된 코드는 환경 E에 변수 x를 추가한 형식 환경 (x::E)에서 표현식 e에 대한 코드이다. 이 코드를 실행하고 나면, 함수의 값매김 결과가 스택의 꼭대기에 저장되어 있을 것이고 리턴 주소는 그 아래 있을 것이다. 그러므로 명령어 RETURN이 수행될 수 있다. RETURN이 실행되고 난 후에, 함수의 값매김 결과가 다시 스택의 꼭대기에 있을 것이고 나머지 계산과정에서 사용될 수 있다.

표현식 (fn x ⇒e)로 만들어진 코드 그 자체는 스택의 꼭대기에 함수의 몸체에 대한 코드의 주소를 PUSH하고 코드 주소와 환경으로 이루어진 함수 묶음을 만들기 위해 명령어 CONS로 되어 있다.

함수 적용 컴파일하기

```
[[(e1 e2)]]E = DUPL; [[e2]]E; SWAP; [[e1]]E; SPLIT;
               IROT3; CONS; SWAP; CALL
```

환경 E에서 표현식 e_1 e_2에 해당하는 코드는 e_2를 계산하는 데에 한번, e_1을 계산하는 데에 한번, 이렇게 환경을 두번 이용하기 위해서 환경(의 주소)를 복사한다. e_2에 해당하는 코드가 실행된 후에 e_2의 값 v_2는 스택의 꼭대기에, 환

경 E(의 주소)의 두번째 복사본은 그 아래에 저장되어 있을 것이다. 그리고 나서 명령어 SWAP이 e_1에 해당하는 코드가 올바르게 수행되기 위해서 v_2와 E의 순서를 바꾼다. 이것이 함수 묶음인 값 v_1, 즉, 첫 부분은 함수 묶음의 환경이고 두번째 부분은 함수 코드의 주소인 쌍을 만든다. 이 쌍은 SPLIT에 의해 두 부분으로 나누어진다. 그리고 나서 스택의 꼭대기에 있는 세개의 원소가 명령어 IROT3에 의해 순서가 조정되고 그 결과 스택의 꼭대기에는 v_2가, 다음 원소에는 적용할 함수 묶음이, 그 다음 원소에는 함수 묶음에 대한 코드의 주소가 위치하게 된다. 명령어 CONS는 이 코드를 부르는 올바른 환경을 만든다. 그리고 나서 명령어 SWAP은 코드의 주소와 환경의 순서를 바꾸어 CALL이 실행되도록 한다.

조건문 컴파일 하기

$[[\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]]_E$ = DUPL; $[[e_1]]_E$; BRANCH a_2 a_3; CALL
여기서 a_2는 코드 $[[e_2]]_E$; RETURN의 주소이고,
여기서 a_3는 코드 $[[e_3]]_E$; RETURN의 주소이다.

표현식 (if e_1 then e_2 else e_3)에 해당하는 코드는 e_1을 계산하는 데에 한번, e_2나 e_3를 계산하는 데에 한번, 이렇게 환경을 두번 이용하기 위해 환경(의 주소)를 복사한다. e_1의 코드를 실행하면, 명령어 BRANCH에 의해 사용될 불 값이 나온다. 여기서 명령어 BRANCH는 CALL을 수행하기 전에 스택에 e_2나 e_3(경우에 따라 다름)에 대한 코드의 주소를 추가한다.

let 컴파일하기

$[[\texttt{let val x=}e_1\texttt{ in }e_2\texttt{ end}]]_E$ = DUPL; $[[e_1]]_E$; CONS; $[[e_2]]_{(x::E)}$;

표현식 let val x=e_1 in e_2 end에 해당하는 코드는 환경을 복사하고, e_1의 값을 계산한 후에 CONS를 이용하여 e_2의 값을 계산하기 전에 환경의 복사본에 이것을 추가한다. e_2는 x가 추가된 환경에서 컴파일됨을 주목하라.

함수형 **let val rec** 컴파일하기

$[[\texttt{let val rec f=}e_1\texttt{ in }e_2\texttt{ end}]]_E$ = PUSH nil; $[[e_1]]_{(f::E)}$; DUPL; ROT3; CONS;
SETFST; FST; $[[e_2]]_{(f::E)}$;

표현식 let val rec f = e_1 in e_2 end에 해당하는 코드는 let에 대한 것과 비슷하다. 차이는 다음의 두가지이다. e_1이 f가 포함된 환경에서 컴파일된다는 점과, 명령어 DUPL; ROT3; CONS; SETFST을 사용해 함수 묶음을 만들고, 이것의 환경 부분(초기에는 빈 환경인)을 f를 포함하는 전체 환경의 주소로 바꾼다는 점이다. 마지막으로 FST를 통해서 이 함수 묶음의 환경에 접근할 때, e_2에 대한 코드를 실행할 올바른 "순환" 환경을 얻게 된다.

비함수형 **let val rec** 컴파일하기

$[[\texttt{let val rec x=}e_1\texttt{ in }e_2\texttt{ end}]]_E$ = DUPL; PUSH nil; DUPL; CONS; CUPL; ROT3;
CONS; $[[e_1]]_{(x::E)}$; DUPL; ROT3; FST; SETFST;
SWAP; SND; SETSND; CONS; $[[e_2]]_{(x::E)}$;

표현식 `let val rec x = e1 in e2 end`에 해당하는 코드는 표현식 e_1의 값이 이중항으로 표현되고, 표현식이 이중항 생성자로만 만들어졌다는 가정을 한다. 처음에 환경(의 주소)를 복사한다. 그리고 나서 e_1의 임시 값인 이중항(`nil,nil`)을 만든다. 명령어 `DUPL; ROT3; CONS;` $[[e_1]]_{(x::E)}$는 임시값인 이중항 (`nil,nil`)의 주소를 바로 아래에 놓여있는 상태에서 스택의 꼭대기에 e_1의 새로운 값을 넣는다.

이제 이 이중항 (`nil,nil`)의 첫칸과 두번째 칸을 e_1의 값을 표현하는 이중항의 첫칸과 두번째 칸으로 연속적으로 바꾸어 적절한 "순환" 구조를 만드는 것만이 남았다. 이것은 명령어 `DUPL; ROT3; FST; SETFST; SWAP; SND; SETSND`로 수행된다. 그리고 나서 이 방식으로 얻게 되는 순환 구조 값은 `CONS`에 의해 초기 환경에 추가되고 e_2를 계산하는 올바른 환경이 만들어 진다.

12.3.3 컴파일 계획에 대하여

우리는 단순하다는 이유로 이 컴파일 계획을 선택하였다. 특히 이 모델에서는 이전에 나왔던 변수값의 리스트를 통해 직접 계산 환경에 접근할 수 있기 때문에, 단지 이 환경과 코드 주소로 쌍을 만들면 되므로 매우 간단히 함수 묶음을 만들 수 있게 한다. 덧붙여 말하자면 환경에 원소를 추가할 때 단순히 이중항을 할당하면 된다. 그래서 이런 방식으로 쉽게 부 환경(subenvironments)를 이끌어낸다.

그렇지만 효율성이라는 측면에서 이 컴파일 계획은 비판받아 마땅하다. 우선, 리스트를 사용하므로 일정 시간안에 해결할 수 있는 것을 리스트의 길이만큼의 시간을 필요로 하게 된다. 게다가 함수 묶음를 위한 환경에다 묶음이 생성된 때에 존재하는 모든 변수 값들을 넣어둘 필요는 없다. 실제로 이 환경들은 오직 함수의 몸체에서 사용되는 (자유)변수만을 저장하면 된다. 함수 묶음을 위한 환경이 너무 많은 값을 가지고 있다면 접근 시간이 길어지게 되고, 좀 더 심각한 점은 실제로 사용되지 않지만 함수 묶음의환경에 속하는 구조에 의해 차지된 공간을 메모리 재활용이 복구하는 것이 어렵다는 것이다. 마지막으로 환경을 관리해야하고 함수 묶음을 만들어야 하기 때문에 다른 전통적인 언어는 계산과정에서 필요한 값들을 위한 스택이 충분히 있다는 것을 생각해 볼 때 함수형 언어의 값매김은 불리한 입장에 있다. 함수형 언어로 쓰여진 모든 프로그램이 실제로 함수형 특성의 능력을 사용하는 것은 아니기 때문에, 효율성을 목표로 하는 경우 이런 프로그램을 효율적으로 컴파일하는 것은 매우 중요하다. 이 목적을 위해 스택을 최대 용량까지 사용한다는 점과 반드시 필요할 때만 환경과 함수 묶음을 만드는 것이 요구된다.

이 모든 이유들 때문에, 앞서 소개한 컴파일 계획은 가능성 여부를 실험해 보는 수준에서 고려되어야 할 것이다.

12.4 구현

먼저 코드 시뮬레이션을 간략히 살펴보고, 이어서 코드 생성을 알아본다.

12.4.1 코드 시뮬레이션

여기서는 "논리적"인 수준에서 기계가 동작하는 방식을 시뮬레이션하는 것까

지만 알아본다. 명령어들은 주소로 접근하는 코드영역 메모리에 있지 않고, 대신 명령어들을 타입 list의 객체로 다룰 것이다. 값과 환경은 메모리 영역에서 있지 않을 것이고, 대신 nML 값으로 다루어 질 것이다. 이렇다 하더라도 실행 함수가 이전 절에서 정의된 기계의 행동을 시뮬레이션하는 것은 변하지 않는다.

```
# type instruction =
        STOP
      | LOAD of value
      | PUSH of value
      | DUPL | SWAP | ROT3 | IROT3
      | FST | SND | SETFST | SETSND
      | CONS | SPLIT
      | ADD | SUB | MULT | EGAL
      | CALL | RETURN
      | BRANCH of value * value
and value =
        Int_Const of int
      | Bool_Const of bool
      | Clo of value ref * instruction list
      | NilList
      | Pair of value ref * value ref
      | Adr of instruction list;;
```

타입 value는 계산과정, 환경, 주소 위치에 사용되는 명령어들로부터 만들어지는 값을 나타낸다. 함수 묶음은 쌍으로 표현된다. 이 쌍의 요소는 참조이기 때문에, 필요하다면 재귀적 용법을 다루기 위해 값을 실제로 수정할 수 있다.

함수 exec는 명령어 리스트와 인자로 스택을 받아, 하나씩 명령어들을 수행한다. 일반적인 실행은 스택에 NilList의 환경 한개만을 저장한 상태에서 시작하여 명령어 Stop에서 종료한다. 마지막 명령어가 실행될 때, 스택에는 계산 결과 값만이 들어있어야 한다.

```
# exception Exec_error;;
exception Exec_error

# val rec exec = fn
    ([STOP],[v]) => v
  | ((LOAD v)::code,v'::stack) => exec (code,v::stack)
  | ((PUSH v)::code,stack) => exec (code,v::stack)
  | (DUPL::code,v::stack) => exec (code,v::v::stack)
  | (SWAP::code,v::v'::stack) => exec(code,v'::v::stack)
  | (ROT3::code,v1::v2::v3::stack) => exec(code,v2::v3::v1::stack)
  | (IROT3::code,v1::v2::v3::stack) => exec(code,v3::v1::v2::stack)
  | (FST::code,(Pair(v1,_))::stack)
        => exec(code,!v1::stack)
  | (SND::code,(Pair(_,v2))::stack)
        => exec(code, !v2::stack)
```

```
  | (SETFST::code, v::(Pair(v1,v2))::stack)
        => v1:=v;exec (code,(Pair(v1,v2))::stack)
  | (SETSND::code, v::(Pair(v1,v2))::stack)
        => v2:=v;exec (code,(Pair(v1,v2))::stack)
  | (CONS::code, v1::  v2::stack)
        => exec(code,(Pair(ref v1,ref v2))::stack)
  | (SPLIT::code, (Pair(v1,v2))::stack)
        => exec(code, !v1::!v2::stack)
  | (ADD::code,(Int_Const v1)::(Int_Const v2)::stack)
        => exec(code, (Int_Const(v1+v2))::stack)
  | (SUB::code,(Int_Const v1)::(Int_Const v2)::stack)
        => exec(code,(Int_Const(v1-v2))::stack)
  | (MULT::code,(Int_Const v1)::(Int_Const v2)::stack)
        => exec(code, (Int_Const(v1*v2))::stack)
  | (EGAL::code,(Int_Const v1)::(Int_Const v2)::stack)
        => exec(code, (Bool_Const(v1=v2))::stack)
  | (CALL::code,(Adr code')::v::stack)
        => exec(code', v::(Adr code)::stack)
  | (RETURN::code,v::(Adr code')::stack)
        => exec(code',v::stack)
  | (BRANCH(adr1,adr2)::code,(Bool_Const b)::stack)
        => if b then exec(code,adr1::stack)
                else exec(code,adr2::stack)
  | _ => raise Exec_error;;
val exec:  instruction list * value list -> value = <fun>
```

12.4.2 코드 생성하기

nML프로그램의 타입은 우리가 11.2절에서 값매김 장치를 공부하면서 사용한 것과 동일하다.

```
# type ml_unop = Ml_fst | Ml_snd;;

# type ml_binop = Ml_add | Ml_sub | Ml_mult | Ml_eq | Ml_less;;

# type ml_exp =
   Ml_int_const of int                    (* 정수 상수 *)
 | Ml_bool_const of bool                  (* bool 상수 *)
 | Ml_pair of ml_exp * ml_exp                   (* 쌍 *)
 | Ml_unop of ml_unop * ml_exp            (* 단항 연산 *)
 | Ml_binop of ml_binop * ml_exp * ml_exp (* 이항 연산 *)
 | Ml_var of string                          (* 변수 *)
 | Ml_if of ml_exp * ml_exp * ml_exp       (* 조건문 *)
 | Ml_fun of string * ml_exp                 (* 함수 *)
 | Ml_app of ml_exp * ml_exp                 (* 적용 *)
 | Ml_let of string * ml_exp * ml_exp        (* 선언 *)
 | Ml_letrec of string * ml_exp * ml_exp  (* 재귀 선언 *)
```

```
;;
```

컴파일 함수 compile은 세개의 보조 함수, compile_unop, compile_binop, compile_var을 사용하여 기본 연산자를 명령어로 변환하고, 변수에 접근하는 것을 환경을 이용해 컴파일한다.

```
# val compile_unop = fn
    Ml_fst => FST
  | Ml_snd => SND;;
val compile_unop:  ml_unop -> instruction = <fun>

# val compile_binop = fn
    Ml_add => ADD
  | Ml_sub => SUB
  | Ml_mult => MULT
  | Ml_eq => EGAL
  | _ => failwith "compile_binop:  not implemented";;
val compile_binop:  ml_binop -> instruction = <fun>

# exception Compile_error of string;;
exception Compile_error of string

# fun compile_var env v =
  case env of
        [] => raise(Compile_error "unbound variable")
      | (x::env) => if x=v then [FST]
              else SND::(compile_var env v);;
val compile_var:  'a list -> 'a -> instruction list = <fun>
```

이제 12.3절에서 알아보았던 것과 정확히 일치하는 컴파일 함수를 알아보자.

```
# fun compile e =
  let fun comp env x = case x of
    (Ml_int_const n) => [LOAD (Int_Const n)]
  | (Ml_bool_const b) => [LOAD (Bool_Const b)]
  | (Ml_pair(e1,e2)) => [DUPL]@(comp env e2)@[SWAP]
              @(comp env e1)@[CONS]
  | (Ml_unop(op,e)) => (comp env e)@[compile_unop op]
  | (Ml_binop(op,e1,e2))
      => [DUPL]@(comp env e2)@[SWAP]@(comp env e1)@
         [compile_binop op]
  | (Ml_var v) => compile_var env v
  | (Ml_if(e1,e2,e3))
      => [DUPL]@(comp env e1)@
         [BRANCH(Adr(comp env e2@[RETURN]),
              Adr(comp env e3@[RETURN])),
```

```
          CALL]
  | (Ml_fun(x,e))
      => [PUSH(Adr(comp (x::env) e @[RETURN])),SWAP,CONS]
  | (Ml_app(e1,e2))
      => [DUPL]@(comp env e2)@[SWAP]@(comp env e1)
         @[SPLIT,IROT3,CONS,SWAP,CALL]
  | (Ml_let(x,e1,e2))
      => [DUPL]@(comp env e1)@[CONS]@(comp (x::env) e2)
  | (Ml_letrec(f,(Ml_fun k),e2))
      => [PUSH NilList]@(comp (f::env) (Ml_fun k))@[DUPL,ROT3,CONS,SETFST,FST]
         @(comp (f::env) e2)
  | (Ml_letrec(f,e1,e2))
      => [DUPL,PUSH NilList,DUPL,CONS,DUPL,ROT3,CONS]
         @(comp (f::env) e1)
         @[DUPL,ROT3,FST,SETFST,SWAP,SND,SETSND,CONS]
         @(comp (f::env) e2)
  in (comp [] e)@[STOP]
  end;;
val compile:  ml_exp -> instruction list = <fun>
```

이제 우리는 함수 compile과 exec를 이용하여 표현식을 값매김 할 수 있다.

```
# val init_stack= [NilList];;
val init_stack:  value list = [NilList]

# fun eval e = exec(compile e,init_stack);;
val eval:  ml_exp -> value = <fun>
```

12.5 예제

우리가 만든 컴파일 코드의 간단한 예제들이다.

```
# compile(ml_exp_of_string "2+3");;
val it:  instruction list
      = [DUPL, LOAD (Int_Const 3), SWAP, LOAD (Int_Const 2), ADD, STOP]

# compile(ml_exp_of_string "fn x => x*2");;
val it:  instruction list
= [PUSH (Adr [DUPL, LOAD (Int_Const 2), SWAP, FST, MULT, RETURN]),
SWAP, CONS, STOP]

# compile(ml_exp_of_string "(fn x => x*2) 3");;
val it:  instruction list
= [DUPL, LOAD (Int_Const 3), SWAP,
PUSH (Adr [DUPL, LOAD (Int_Const 2), SWAP, FST, MULT, RETURN]),
SWAP, CONS, SPLIT, IROT3, CONS, SWAP, CALL, STOP]
```

이제 값매김에 관한 11.2절의 두개의 예제를 살펴보자.

```
# val P1=
  (ml_exp_of_string
  ("let val double = fn f => fn x => f(f x)" ^
  "in let val sq = fn x => x*x" ^
  "   in (double sq) 5 end end"));;

# compile P1;;
val it:  instruction list
= [DUPL,
        PUSH
         (Adr
          [PUSH
             (Adr
               [DUPL, DUPL, FST, SWAP, SND, FST, SPLIT, IROT3, CONS, SWAP,
                CALL, SWAP, SND, FST, SPLIT, IROT3, CONS, SWAP, CALL, RETURN]),
            SWAP, CONS, RETURN]),
        SWAP, CONS, CONS, DUPL,
        PUSH (Adr [DUPL, FST, SWAP, FST, MULT, RETURN]), SWAP, CONS, CONS,
        DUPL, LOAD (Int_Const 5), SWAP, DUPL, FST, SWAP, SND, FST, SPLIT,
        IROT3, CONS, SWAP, CALL, SPLIT, IROT3, CONS, SWAP, CALL, STOP]

# eval P1;;
val it:  value = Int_Const 625

# val P2 =
 (ml_exp_of_string
  ("let val rec fact=" ^
   "fn n => if n=0 then 1 else n*(fact(n-1)) in" ^
   "fact 10 end"));;

# compile P2;;
val it:  instruction list
      = [PUSH NilList,
         PUSH
          (Adr
            [DUPL, DUPL, LOAD (Int_Const 0), SWAP, FST, EGAL,
             BRANCH
              (Adr [LOAD (Int_Const 1), RETURN],
               Adr
                [DUPL, DUPL, DUPL, LOAD (Int_Const 1), SWAP, FST, SUB, SWAP,
                 SND, FST, SPLIT, IROT3, CONS, SWAP, CALL, SWAP, FST, MULT,
                 RETURN]),
             CALL, RETURN]),
         SWAP, CONS, DUPL, ROT3, CONS, SETFST, FST, DUPL,
         LOAD (Int_Const 10), SWAP, FST, SPLIT, IROT3, CONS, SWAP, CALL,
         STOP]
```

```
# eval P2;;
val it:  value = Int_Const 3628800
```

마지막 예제는 어떻게 재귀적인 값을 만드는 프로그램이 실행하는지 보여준다.

```
# val P3 =
  (ml_exp_of_string
  "let rec p = (1,(2,p)) in fst(snd(snd p)) end");;

# compile P3;;
val it:  instruction list
= [DUPL, PUSH NilList, DUPL, CONS, DUPL, ROT3, CONS, DUPL, DUPL, FST,
   SWAP, LOAD (Int_Const 2), CONS, SWAP, LOAD (Int_Const 1), CONS,
   DUPL, ROT3, FST, SETFST, SWAP, SND, SETSND, CONS, FST, SND, SND,
   FST, STOP]

# eval P3;;
val it:  value = Int_Const 1
```

연습 문제
12.3 필요에 의한 값매김을 이용한 방식으로 컴파일러를 수정하라.

12.6 요약

지금까지 메모리에 함수형 언어의 값을 표현하는 원리를 살펴보고, 자료 구조의 생성자를 묵시적으로 지원하기 위해 이중항을 할당하는 간단한 방법을 공부해보았다. 또한 값을 만드는 방법에서 유도된 공유(sharing)를 강조하였다.

이후에, nML의 함수형 특성을 구현하는 저수준의 명령어 집합을 살펴보았다. 또한 이 명령어 집합으로 nML을 컴파일하는 알고리즘을 정의하였다. 알고리즘은 세개의 메모리 영역 - 코드, 힙, 그리고 스택 - 에서 동작한다. 이번에 사용한 실행 모델은 11장에서 살펴본 값매김 장치에서 유도된 환경에 기초하고 있지만, 여기서 환경이라는 아이디어는 형식 환경(컴파일 과정에서 사용되는 환경)과 실행 환경으로 나뉘어진다.

마지막으로 컴파일 함수와 nML에서 코드 실행을 모의로 실험해보는 함수를 작성해 보았다.

12.7 더 배울 내용들

함수형 언어를 구현하기 위한 방법을 설명한 책이 많지는 않다. 주로 찾기 어려운 연구 기사나 일반적으로 구현에 대한 자세한 사항보다는 이론적인 모델을 설명하는 논문에서 이런 것을 설명한다.

CAML LIGHT에 사용된 구현 방법에 관심이 있는 독자는 비록 [23]이 이 구현의 좀 오래된 버전에 대해 다루지만 봐야할 필요가 있다.

컨티뉴에이션에 근거한 컴파일 방식이나 SML에서 구현된 방식은 [4]를 참조한다.

마지막으로 늦은 값매김이나 지연된 값매김을 사용한 언어를 구현하는 방법을 설명한 것들은 많이 있다. 특히 [31]과 [32]를 볼 필요가 있다.

13 장

타입 유추

이전 장들에서 nML의 몇가지 타입 유추를 살펴보았다. 5장에서 다형성을 다루는 것을 제외하고는 거의 대부분 살펴보았다. 그리고 5.3.1절에서 프로그램의 타입을 유추할 수 있는 조건을 간략하게 알아보았다. nML은 다형적 타입 체계(polymorphic type system)을 가지고 있기 때문에, `let val x = e1 in e2 end`에서 나오는 e_1같은 값은 e_2에서 `x`가 나타나는 방식에 따라 서로는 모순될지라도 여러가지 다른 타입을 가질 수 있다.

그렇지만 5.3.1절에서 우리는 `let val x = e1 in e2 end`에서 `x`의 위치에 표현식 e_1을 치환하는 방식으로, 즉 e_2 `[x` $\leftarrow$ e_1`]`를 사용하는 것으로 표현식 e_2의 타입을 결정함으로써, 같은 효과를 얻을 수 있음을 공부하였다. 이 방식의 중요한 문제는 e_1의 타입을 여러번 유추해야 한다는 것이다. 불편함을 없애기 위해 몇몇 자연언어에서 문맥에 따라 명사나 형용사를 어형변화(decline)하는 것과 같은 방식으로, e_1과 타입사이의 관계를 맺어 `x`가 나올 때마다 여러 방식으로 어형변화를 시킬 수 있다.

좀더 정식으로 이야기하자면, 이번 장에서 nML 타입 유추를 다형성까지 고려해서 살펴볼 것이고 실제로 사용되는 유추 알고리즘을 공부할 것이다.

13.1 타입 규칙

nML에서 타입 유추는 값매김처럼 삼항 연산자로 정확히 정의할 수 있다.

$$E \vdash e : t$$

이것은 다음과 같이 읽는다.

> 타입 환경 `E`에서 표현식 `e`는 타입 `t`를 가진다.

값 환경이 변수과 값의 관계 리스트였던 것 처럼, 타입 환경은 변수와 타입의 관계 리스트이다.

13.1.1 치환에 의한 다형성

다형성은 타입 유추를 어렵게 한다. `(fn x ⇒ x)`와 같은 표현식은 타입 `int -> int`이 될 수도 있지만 또한 타입 `bool -> bool`일 수도, 더 나아가 다형적 타입 `'a -> 'a`일 수도 있다. 복잡해 보이지만 이것을 추론 규칙으로 정의하는 것은 가능하다.

$$\frac{\mathtt{x} : \mathtt{t} \in \mathtt{E}}{\mathtt{E} \vdash \mathtt{x} \ : \ \mathtt{t}}(\text{변수})$$

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \ : \ \mathtt{t}_1 \quad \ldots \quad \mathtt{E} \vdash \mathtt{e}_n \ : \ \mathtt{t}_n}{\mathtt{E} \vdash (\mathtt{e}_1, \ldots, \mathtt{e}_n) \ : \ (\mathtt{t}_1 * \ldots * \mathtt{t}_n)}(\text{N-튜플})$$

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \ : \ \mathtt{bool} \quad \mathtt{E} \vdash \mathtt{e}_2 \ : \ \mathtt{t} \quad \mathtt{E} \vdash \mathtt{e}_3 \ : \ \mathtt{t}}{\mathtt{E} \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \ : \ \mathtt{t}}(\text{조건문})$$

$$\frac{(\mathtt{x} : \mathtt{t}_1) :: \mathtt{E} \vdash \mathtt{e} \ : \ \mathtt{t}_2}{\mathtt{E} \vdash \mathtt{fn}\ \mathtt{x} \Rightarrow \mathtt{e} \ : \ \mathtt{t}_1 \rightarrow \mathtt{t}_2}(\text{함수})$$

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \ : \ \mathtt{t} \rightarrow \mathtt{t}' \quad \mathtt{E} \vdash \mathtt{e}_2 \ : \ \mathtt{t}}{\mathtt{E} \vdash (\mathtt{e1}\ \mathtt{e2}) \ : \ \mathtt{t}'}(\text{적용})$$

이제 생성자 let이 문제이다. 왜냐하면 이 생성자는 같은 변수를 두개의 다른 타입으로 사용하게 만들기 때문이다. (예를 들자면, `let val id = (fn x ⇒ x) in id id end`와 같은 경우가 있다.) 5장에서는 `let val id = (fn x ⇒ x) in id id end`을 `(fn x ⇒ x) (fn x ⇒ x)`와 같은 식으로 치환함으로써 이 문제를 다룰 수 있다고 제안하였다. 이런 방식으로 충분히 두 표현식 `(fn x ⇒ x)`의 타입을 알 수 있다.

만약 이런 방식으로 다형성을 다루려고 한다면, 생성자 `let`에 대한 규칙을 다음과 같이 쓸 것이다.

$$\frac{\mathtt{E} \vdash \mathtt{e}_2[\mathtt{x} \leftarrow \mathtt{e}_1] \ : \ \mathtt{t}}{\mathtt{E} \vdash \mathtt{let}\ \mathtt{val}\ \mathtt{x} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2\ \mathtt{end} \ : \ \mathtt{t}}(\text{Let})$$

그렇지만 x가 $\mathtt{e}_2$에 나타나지도 않는 경우에도 $\mathtt{e}_1$의 타입이 결정되어야 하고, 따라서 다음과 같이 규칙을 쓸 수 있다.

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \ : \ \mathtt{t}_1 \quad \mathtt{E} \vdash \mathtt{e}_2[\mathtt{x} \leftarrow \mathtt{e}_1] \ : \ \mathtt{t}}{\mathtt{E} \vdash \mathtt{let}\ \mathtt{val}\ \mathtt{x} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2\ \mathtt{end} \ : \ \mathtt{t}}(\text{Let})$$

재귀적 용법을 고려하기 위해 다시쓰기에 의한 값매김을 정의하기 위해 사용했던 상수 Rec와 let에서 사용한 것과 같은 종류의 치환을 이용할 것이다.

$$\frac{\mathtt{E} \vdash \mathtt{e}_2[\mathtt{f} \leftarrow \mathtt{Rec}(\mathtt{fn}\ \mathtt{f} \Rightarrow \mathtt{e}_1)] \ : \ \mathtt{t}}{\mathtt{E} \vdash \mathtt{let}\ \mathtt{val}\ \mathtt{rec}\ \mathtt{f} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2\ \mathtt{end} \ : \ \mathtt{t}}(\text{Letrec})$$

역시 여기서도 전체 프로그램이 올바르게 타입을 유추하는 것을 보장하기 위해서 표현식 $\mathtt{e}_1$의 타입을 매길 수 있어야 함을 보장해야 한다. 그리고 이것은 f가 반드시 $\mathtt{e}_2$에 나타날 필요는 없기 때문에 위 규칙에 의해서는 확실히 알 수 없다.

이 체계가 잘 동작한다면 언어에서 미리 정의된 상수의 타입을 유추하는 규칙도 만들 수 있을 것이다. 상수 Rec가 $((\mathtt{t}_1 \rightarrow \mathtt{t}_2) \rightarrow (\mathtt{t}_1 \rightarrow \mathtt{t}_2)) \rightarrow (\mathtt{t}_1 \rightarrow \mathtt{t}_2)$의 형태를 갖는 어떤 타입도 가질 수 있게 할 것이다.

불행히도 부 프로그램을 치환함으로서 다형성을 다루려는 것은 현실적이지 못하다. 그래서 새로운 아이디어를 제시할 것이다. 그것은 주어진 표현식의 가능한 모든 타입을 표현할 수 있는 타입 개요이다.

13.1.2 매개변수를 이용한 다형성

좀더 현실적인 방법은 **타입 개요**(type scheme)의 개념을 이용하는 것이다. 타입 개요는 어떤 변수를 특화시킨 타입에 해당한다. 이 특별 변수를 **포괄적**(generic)이라고 할 것이다. 만약 우리가 포괄 변수의 타입을 치환함으로써 타입 개요는 여러 종류의 다른 타입(타입 개요의 개체)로 변화될 수 있다.

$\alpha_1, \cdots, \alpha_n$이 포괄 변수인 타입 개요는 t가 타입일 때 $\forall\alpha_1 \cdots \alpha_n.\mathtt{t}$로 표현될 것이다. t에서 변수 $\alpha_1 \cdots \alpha_n$을 타입 $\mathtt{t}_1, \cdots, \mathtt{t}_n$으로 치환함으로써 이 타입 개요의 개체를 얻을 수 있다. 따라서 주어진 타입들 $\mathtt{t}_i$에 대해서 $\mathtt{t}[\alpha_i \leftarrow \mathtt{t}_i]_{i=1,n}$은 이러한 타입 개요 개체이다.

그러므로 모든 t에 대해서 타입 t->t를 가지는 (fn x ⇒ x)에 대해서 이 표현식이 타입 개요 $\forall\alpha\,.\,\alpha \rightarrow \alpha$를 가진다고 할 것이다.

타입의 포괄 변수의 집합이 공집합인 경우, 우리는 타입과 타입 개요의 두 가지 아이디어를 동일하다고 간주할 것이다. 그러므로 타입은 또한 타입 개요가 되며 이것은 **명백**(trivial)하다.

13.1.3 타입 개요를 포괄화 하기

let val x = $\mathtt{e}_1$ in $\mathtt{e}_2$ end의 타입을 유추할 때, $\mathtt{e}_1$이 타입 $\mathtt{t}_1$라고 생각할 것이다. 그리고 나서 이것의 포괄 변수의 집합, $\alpha_1, \cdots, \alpha_n$을 구하고, x와 타입 개요 $\forall\alpha_1 \cdots \alpha_n\,.\,t_1$사이의 묶음을 추가한 타입 환경에서 $\mathtt{e}_2$의 타입을 유추할 것이다.

이제 타입에서 나타나는 모든 변수중에서 어떤 타입 변수가 포괄화 될 수 있는 지 결정해야만 한다.

타입 제한 조건에 대해 살펴본 5.3.1절에서, 타입 유추는 방정식 집합의 해를 구하는 것으로 표현되었다. 이런 관점에서, 방정식이 해결되면(즉, 동일화(unification)가 일어나면), 정확한 값을 갖게 될 미지수로써 변수들을 바라볼 수 있다. 이런 미지수 가운데 방정식을 끝까지 풀고난 후에도 여전히 값을 갖지 못한 것은 포괄화될 수 있다. 사실 그것들은 정확한 값을 갖지 않기 때문에 임의의 타입으로 표현한다.

추론 규칙에서 사용하는 어구를 이용해 다음과 같은 형태로 규칙을 바꾸려고 한다.

$$\frac{\mathtt{E} \vdash \mathtt{e}_2[\mathtt{x} \leftarrow \mathtt{e}_1] \;:\; \mathtt{t}}{\mathtt{E} \vdash \mathtt{let\ val\ x} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2\ \mathtt{end} \;:\; \mathtt{t}}(\mathrm{Let})$$

이것을 다음과 같이 바꾼다.

$$\frac{\mathtt{E} \vdash \mathtt{e_1} \ : \ \mathtt{t_1} \qquad (\mathtt{x} : \forall \alpha_1 \cdots \alpha_n \, . \, \mathtt{t_1}) :: \mathtt{E} \vdash \mathtt{e_2} \ : \ \mathtt{t}}{\mathtt{E} \vdash \mathtt{let\ val\ x = e_1\ in\ e_2\ end} \ : \ \mathtt{t}} (\text{Let})$$

그리고 α_i의 집합을 구한다. 결과적으로 $\mathtt{e_1}$의 타입을 유추하면서 얻은 제한 조건을 완전히 해결해야만 한다. 그리고 나서 $\mathtt{t_1}$에 나타나는 변수 중에서 정확한 값을 갖지 않는 타입 변수를 포괄화해야한다. 앞서 살펴보았듯이, 이것들은 $\mathtt{e_1}$에 관련된 문맥에 의존하지 않는 것들이다. 문맥은 타입 환경 E로 표현되고, 따라서 포괄화된 변수들은 E에 나타나지 않는 $\mathtt{t_1}$의 변수들이다. (E가 이제부터 타입 개요를 가지고 있음을 기억하라.)

따라서 포괄화된 변수의 집합은 $\mathtt{vars(t_1)} \backslash \mathtt{vars(E)}$이다.

13.1.4 개체화(Instantiation)

(let의 의해 만들어지는) 변수가 여러번 나오면, 다형성에 의해 같은 변수가 다른 타입(개체)를 가질 수 있다. 각각의 타입이 나오는 것과 타입들이 속한 타입 개요의 관계를 **개체화 관계**(instantiation relation)라고 한다. 이것은 $\leq$라고 쓴다. 만약 $\mathtt{t'} = \mathtt{t}[\alpha_i \leq \mathtt{t}_i]_{i=1,n}$인 타입 $\mathtt{t}_1, \cdots, \mathtt{t}_n$이 존재한다면, 타입 t'은 타입 개요 $\forall \alpha_1 \cdots \alpha_n \, . \, t$의 개체이다.

13.1.5 매개변수를 이용한 다형성에서의 타입 규칙

이제 우리의 작은 언어에서 타입에 관한 새로운 추론 규칙을 명료하게 표현할 수 있다. 이 규칙들은 이름과 let 생성자에 대한 규칙을 제외하면 13.1.1절에서 소개된 체계와 같다. 여기서 t, $\mathtt{t}_1$ 등으로 표시된 객체는 타입 개요가 아닌 타입을 나타낸다. 타입에 관한 추론 규칙에서 타입 개요는 오직 타입 환경에서만 나타날 지도 모른다.

두개의 새로운 규칙은 다음과 같다.

$$\frac{\mathtt{x} : \forall \alpha_1 \cdots \alpha_n \, . \, \mathtt{t_1} \in \mathtt{E} \qquad \mathtt{t'} \leq \forall \alpha_1 \cdots \alpha_n \, . \, \mathtt{t}}{\mathtt{E} \vdash \mathtt{x} \ : \ \mathtt{t'}} (\text{변수})$$

$$\frac{\mathtt{E} \vdash \mathtt{e_1} \ : \ \mathtt{t_1} \qquad \{\alpha_i\}_{i=1,n} = \mathtt{vars}(\mathtt{t_1}) \setminus \mathtt{vars}(\mathtt{E}) \qquad (\mathtt{x} : \forall \alpha_1 \cdots \alpha_n \, . \, \mathtt{t_1}) :: \mathtt{E} \vdash \mathtt{e_2} \ : \ \mathtt{t_2}}{\mathtt{E} \vdash \mathtt{let\ val\ x = e_1\ in\ e_2\ end} \ : \ \mathtt{t_2}} (\text{Let})$$

Let에 대한 타입 규칙이 타입 환경에 타입 개요를 추가한다는 점에 주목하라. (이 타입 개요는 만약 어떤 타입 변수도 포괄화하지 않았다면 명백한 타입 개요일 수도 있다.) 이에 비해, fn에 대한 규칙은 명백한 타입 개요(즉, 타입)를 환경에 추가하지 않는다.

간단히 하기 위해 단항 연산자와 이항 연산자에 대한 규칙은 생략했다.

여기서 하나의 타입 규칙이 언어의 각 생성자에 해당함에 주목하라. 따라서 표현식의 타입을 유추하기 위한 프로그램은 인자의 구조에 해당하는 경우에 따라 정의된 함수일 것이다.

반대로, 이 추론 규칙을 알고리즘으로 변환하는 것은 값매김 프로그램의 경우만큼 간단하지 않다. 값매김의 경우, 환경 E와 표현식 e를 받아서 값 v를 반환

하는 함수 eval에서 함수의 정의에 나타나는 하나씩의 경우로 추론규칙을 이해할 수 있었다. 각 규칙에 대해서 표현식(직접적인 e의 부 표현식이거나 중간값에서 얻은 표현식)에 대한 (규칙의 전제에 해당하는) 재귀 호출을 이용하여 반환할 값 v를 계산할 수 있었다. 각 호출을 하는 동안, 결과를 위해 저장해야 하거나 다음 재귀 호출을 하기 위해 전달해야할 중간값을 계산한다. 마지막 재귀 호출을 하면 최종 결과의 전체 또는 일부분이 만들어진다.

타입 유추의 경우 상황이 조금 복잡하다. 실제로 다음의 두가지 문제에 직면해 있다.

1. 타입 t가 e_2의 타입이면서 동시에 e_1의 타입의 인자 부분이어야 하는 함수 적용에 대한 규칙에서 나타나는 일반적인 동등 조건을 어떻게 고려해야 하는가?
2. 추상화(함수)의 타입을 정할때, 형식 인자의 타입에 대한 가정(t_1)을 세워야 한다. 어떻게 올바른 가정을 할 것인가? 다른 말로 하면 어떻게 이전에 언급한 제한 조건의 결과를 프로그램의 나머지 부분까지 도달하도록 전파시킬 것인가?

동등 제한 조건을 전달하는 문제는 특히 어려워 보인다. 이 제한 조건은 함수 적용때문에 나온 것이고, 따라서 제한 조건들을 뒤쪽으로, 예를 들어 함수 인자의 타입으로, 전달해야 한다. 치환에 의한 다형성이든, 매개 변수에 의한 다형성이든 어떤 방법을 선택하든지 간에 이 어려운 문제에 계속 직면하게 된다.

물론 우리의 답은 동일화를 이용하는 것이다. (동일화에 대한 소개는 5.2.4절에 있다.) 타입에 대한 동등 조건과 이 조건을 전달하는 것은 두개의 결정적인 상황에서 고려된다.

1. 두 개의 타입이 같아지기 전에 그들을 동일화하기 위해 치환을 할 때
2. 치환한 것을 타입 환경에 적용하기 위해 결과의 일부분으로써 해당 치환을 반환할 때

사실상 동등 조건이 전달되는 것은 타입 환경을 통해 이루어진다. 이것은 동등 조건을 이용해 조금 더 개체화한 타입 개요들을 타입 환경이 가지고 있기 때문이다.

13.2 타입 유추 프로그램 작성하기

어떻게 타입 유추 프로그램을 작성할지를 고민하기 전에 언어, 타입 표현방식, 치환, 그리고 개체화와 포괄화를 위한 방법을 생각해야 한다.

언어의 표현식, 타입, 그리고 치환에 대해서 5.3.1절에서 소개한 것을 다시 살펴볼 것이다. 언어의 타입은 변수가 정수인 어구로 만들 것이다. 예를 들어 다음의 함수는 단항과 이항 연산자의 입력 타입과 결과 타입을 제공한다.

```
# val unop_type = fn
    Ml_fst ⇒ let val a = Var(new_int()) val b = Var(new_int())
```

```
          in (pair(a,b),a)
          end
  | Ml_snd ⇒ let val a = Var(new_int()) val b = Var(new_int())
          in (pair(a,b),b)
          end;;
val unop_type:  ml_unop -> (string, int) term * ('a, int) term = <fun>

# val binop_type = fn
    Ml_add ⇒ (const "int", const "int",const "int")
  | Ml_sub ⇒ (const "int", const "int",const "int")
  | Ml_mult ⇒ (const "int", const "int",const "int")
  | Ml_eq ⇒ (const "int", const "int",const "bool")
  | Ml_less ⇒ (const "int", const "int",const "bool");;
val binop_type:  ml_binop -> (string, 'a) term * (string, 'b) term *
          (string, 'c) term = <fun>
```

또한 새로운 개념인 타입 개요를 고려한다. 이것을 다음의 타입으로 표현한다.

```
# type 'a scheme = Forall of int list * 'a;;
```

예를 들어

```
# Forall([1], arrow(Var 1, Var 1));;
val it:  (string, int) term scheme
      = Forall ([1], Term (``arrow'', [Var 1, Var 1]))
```

는 함수 (fn x ⇒ x)와 관련된 타입 개요 $\forall\alpha\,.\,\alpha \to \alpha$을 표현한다.

13.2.1 타입 포괄화하기

(Let)규칙에서 표현했듯이, 타입의 포괄화는 현재 타입 환경에 관계되어 있다. 따라서 다음의 두개의 함수가 필요하다.

```
subtract :  'a list -> 'a list -> 'a list
unique :  'a list -> 'a list
```

subtract l_1 l_2의 결과는 l_2의 가능한 원소들을 모두 제외한 리스트 l_1이다. 함수 subtract는 미리 정의되었다.

unique xs의 결과는 중복을 제거한 리스트 xs이다. 따라서 결과로 얻은 xs'의 원소는 리스트 xs'에서 유일하다. unique의 정의는 연습문제로 남겨두기로 한다.

다음의 함수는 인자로 전달되는 타입환경에 나타나는 아직 정의되지 않은 변수를 찾는다.

```
# fun vars_of_tyenv env =
  flat_map (fn (_, Forall(gvars, t)) ⇒ subtract (vars t) gvars) env;;
val vars_of_tyenv:  ('a * ('b, int) term scheme) list -> int list = <fun>
```

포괄화 함수는 다음과 같이 간단히 정의된다.

```
# fun generalize env t =
  let val gvars = unique (subtract (vars t) (vars_of_tyenv env))
  in Forall(gvars, t)
  end;;
val generalize:  ('a * ('b, int) term scheme) list
          -> ('c, int) term -> ('c, int) term scheme = <fun>
```

13.2.2 개체화

주어진 타입 개요에 대해, 개체화 함수는 그 개요의 "가장 작은" 개체를 반환한다. 즉, 함수는 단순히 타입 개요의 포괄 변수의 이름을 다시 짓는다. 이런 방식으로 새로운 타입 변수가 좀더 정확한 값을 갖게 하는 것은 동일화에 달려있다.

```
# val instance = fn Forall(gvars,t) ⇒
  let renaming = map (fn n ⇒ (n, Var(new_int()))) gvars
  in apply_subst renaming t
  end;;
val instance:  ('a, int) term scheme -> ('a, int) term = <fun>
```

13.2.3 추가 함수들

이제 타입을 유추하는 주 알고리즘을 정의하기 전에 몇가지 부가 함수를 정의한다.

치환. 우선, 타입환경에 타입 개요를 가지고 있기 때문에 (그러므로 정량화된 변수를 가지고 있기 때문에), 치환을 적용할 때 이것들을 치환하지 않도록 조심해야 한다. 다음 함수는 치환의 정의역에서 변수를 제거한다.

```
# fun subst_but v x = case x of
    [] ⇒ []
  | (v1,t1)::subst ⇒
      if v1 = v then subst_but v subst
      else (v1,t1)::(subst_but v subst);;
val subst_but:  'a -> ('a * 'b) list -> ('a * 'b) list = <fun>
```

만약 변수 리스트에 대해 이 과정을 반복하면, 다음과 같이 치환의 정의역으로부터 변수의 집합을 제거할 수 있다.

```
# val rec subst_minus subst vars =
  case vars of
        [] ⇒ subst
      | v::vs ⇒ subst_minus (subst_but v subst) vs;;
val subst_minus:  ('a * 'b) list -> 'a list -> ('a * 'b) list = <fun>
```

이제 타입 환경에 치환을 적용할 수 있다.

```
# val subst_env subst env =
  map(fn (k, Forall(gvars,t)) ⇒
          (k,Forall(gvars, apply_subst (subst_minus subst gvars) t))) env;;
val subst_env:  (int * ('a, int) term) list
          -> ('b * ('a, int) term scheme) list
          -> ('b * ('a, int) term scheme) list = <fun>
```

타입에서 어구로의 변환. 5.2절에서 정의한 함수를 이용해 타입에서 어구로 변환하기 위해서는, 먼저 정수 타입의 변수를 문자열로 바꿀 필요가 있다. 이것은 함수 make_string_vars가 수행한다.

```
# val rec term_map fop fleaf x = case x of
    Term(oper, sons) ⇒ Term(fop oper, map (term_map fop fleaf) sons)
  | Var(n) ⇒ Var(fleaf n);;
val term_map:  ('a -> 'b) -> ('c -> 'd) -> ('a, 'c) term -> ('b, 'd) term = <fun>

# val make_string_vars t =
  let var_of_int n =
      ``v''^(string_of_int n)
  in term_map (fn x⇒x) var_of_int t end;;
val make_string_vars:  ('a, int) term -> ('a, string) term = <fun>
```

13.2.4 주 함수

이제 타입을 유추하는 주 함수를 정의해보자. 시작하기 전에, 그 속에 있는 원리를 명확히 하기위해 몇가지를 생각해보자.

함수 type_expr은 두개의 인자를 받는다. 타입 환경 E와 표현식 e이다. 그리고 타입 t와 치환 σ를 반환한다. 이 치환은 타입 환경 E에 대해 동등 조건이 나타내는 효과를 표현한다. 이 동등 조건은 e의 타입이 정해지는 동안에 만들어진다. 그러므로 이것을

$$(\sigma,\texttt{t}) = \texttt{type\_expr E e}$$

다음과 같이

$$\sigma(E) \vdash e : t$$

해석해야한다.

결론적으로 다음의 알고리즘에서 치환을 다루는 것은 매우 힘들다. 특히 현재 표현식이 몇개의 부 표현식(조건문에서와 같이)을 포함하는 경우에, 각 부 표현식에 대해

- 이 부표현식의 타입 t와 치환 σ를 얻어야 한다.
- 치환 σ에 의해 수정된 환경에서 이후의 부 프로그램의 타입을 결정해야 한다.

그리고 이것이 전부가 아니다. 현재 부 표현식의 단계에서 동일화 과정을 통해 만들어진 중간결과물인 부 치환들도 추가 되어야 하고, 따라서 결과로 반환할 치환을 만들기 위해 이것들을 모두 합성해야 한다.

따라서 함수의 전형적인(그리고 흥미로운) 경우들은 다음과 같다.

상수 해당하는 타입을 반환하고, 빈 치환을 생성한다.

변수 현재 환경에서 변수와 연관되어있는 타입 개요의 개체를 반환한다.

추상 새로운 타입 변수 α와 치환σ를 얻고, 함수의 몸체에서 타입 $\mathtt{t}_2$를 유추하기 위해 앞서 얻은 타입 변수 α와 형식 인자를 관계 짓는다.

Let let문 내에서 선언된 값의 타입 $\mathtt{t}_1$을 유추한다. 이를 통해 치환 σ_1를 얻는다. 환경 σ_1(E)에 대해 $\mathtt{t}_1$을 포괄화 하여 타입 개요 $\mathtt{ts}_1$을 계산한다. 마지막으로 let에 의해 선언된 변수 이름과 타입 개요 $\mathtt{ts}_1$의 묶음을 추가한 환경 σ_1(E)에서 let의 몸체의 타입을 계산한다.

적용 먼저 적용의 두개의 부 표현식의 타입을 얻는다. 이를 위해, 처음 부 표현식에서 계산된 치환을 사용하여 두번째 것의 타입을 만든다. 적용의 함수 부분의 타입은 반드시 함수형 타입 $\mathtt{t'}_1 \rightarrow \mathtt{t'}_2$여야 하고, $\mathtt{t'}_1$는 또한 인자의 타입이어야 한다. 만약 두개의 부 표현식의 타입 결과를 (σ_1,$\mathtt{t}_1$)과 (σ_2,$\mathtt{t}_2$)라고 하면, 다음을 수행해야 한다.

- $\mathtt{t}_2 \rightarrow\sigma$로 $\sigma_2(\mathtt{t}_1)$를 동일화한다. (α는 새로운 타입 변수이다.) 그리고 이것의 동일화 장치 μ를 호출한다.
- (μ o σ_2 o σ_1, $\mu(\alpha)$)를 결과로 반환한다.

재귀 기본 아이디어는 새로운 타입 변수와 재귀적으로 정의된 변수 이름을 관계짓고, 정의의 몸체부분의 타입을 계산하는 것이다. 이것은 동일화를 통해 타입 변수의 값을 좀더 많이 개체화한 형태로 만들 것이다. 그 후에는 let과 똑같이 수행한다.

(비록 남은 경우들이 이들보다 일반적으로 좀더 많지만) 남은 경우들은 단순히 적용에서 살펴본 것의 변형이다.

```
# val rec type_expr = fn tenv expr => case expr of
    Ml_int_const n => ([],const "int")
  | Ml_bool_const b => ([],const "bool")
  | Ml_var s => ([], instance (List.assoc s tenv)
                      handle Not_found => failwith "Unbound variable")
  | Ml_fun(s,e) =>
      let val alpha = Var(new_int())
      in let val (su,t) = type_expr ((s,Forall([],alpha))::tenv) e
          in (su,arrow(apply_subst su alpha,t))
          end
      end
  | Ml_let(s,e1,e2) =>
      let val (su1,t1) = type_expr tenv e1
```

```
      in let val ts1 = generalize (subst_env su1 tenv) t1
          in let val (su2,t2) = type_expr ((s,ts1)::(subst_env su1 tenv)) e2
              in (compsubst su2 su1, t2)
              end
          end
      end
  | Ml_app(e1,e2) =>
      let val (su1,t1) = type_expr tenv e1
      in let val (su2,t2) = type_expr (subst_env su1 tenv) e2
          in let val alpha = Var(new_int())
              in let val mu = unify (apply_subst su2 t1, arrow(t2,alpha))
                  in (compsubst mu (compsubst su2 su1),apply_subst mu alpha)
                  end
              end
          end
      end
  | Ml_unop(unop,e) =>
      let val (ti,t0) = unop_type unop
      in let val (su,t) = type_expr tenv e
          in let val mu = unify (ti,t)
              in (compsubst mu su, apply_subst mu t0)
              end
          end
      end
  | Ml_pair(e1,e2) =>
      let val (su1,t1) = type_expr tenv e1
      in let val (su2,t2) = type_expr (subst_env su1 tenv) e2
          in (compsubst su2 su1, pair(apply_subst su2 t1, t2))
          end
      end
  | Ml_binop(binop,e1,e2) =>
      let val (ta1,ta2,tr) = binop_type binop in
      let val (su1,t1) = type_expr tenv e1 in
      let val mu1 = unify (t1, ta1) in
      let val s1 = compsubst mu1 su1 in
      let val (su2,t2) = type_expr (subst_env s1 tenv) e2 in
      let val s2 = compsubst su2 s1 in
      let val mu2 = unify (t2,apply_subst s2 ta2) in
      let val s3 = compsubst mu2 s2 in
      (s3,apply_subst s3 tr) end end end end end end end end
  | Ml_if(e1,e2,e3) =>
      let val (su1,t1) = type_expr tenv e1 in
      let val mu1 = unify (t1, const "bool") in
      let val s1 = compsubst mu1 su1 in
      let val (su2,t2) = type_expr (subst_env s1 tenv) e2 in
      let val s2 = compsubst su1 s1 in
```

```
      let val (su3,t3) = type_expr (subst_env s2 tenv) e3 in
      let val s3 = compsubst su3 s2 in
      let val mu3 = unify (t3, apply_subst su3 t2) in
      (compsubst mu3 s3, apply_subst mu3 t3) end end end end end end end end
  | Ml_letrec(s,e1,e2) =>
      let val t1 = Var (new_int()) in
      let val ts1 = Forall([],t1) in
      let val (su1,t1) = type_expr ((s,ts1)::tenv) e1 in
      let val tss = generalize (subst_env su1 tenv) (apply_subst su1 t1) in
      let val (su2,t2) = type_expr ((s,tss)::(subst_env su1 tenv)) e2 in
      (compsubst su2 su1,t2) end end end end end;;
val type_expr:  (string * (string, int) term scheme) list ->
              ml_exp -> (int * (string, int) term) list * (string, int) term = <fun>
```

13.2.5 예제

다음 함수는 몇가지 예제를 통해 방금 만든 알고리즘을 테스트한다.

```
# val type_of = fn e =>
  reset_new_int();
  let val (su,t) = type_expr [] e
  in ml_type_of_term (make_string_vars t)
  end;;
val type_of:  ml_exp -> ml_type = <fun>
```

예제에서 타입 유추기의 결과로 얻는 값은 다음의 함수를 통해 출력한다.

```
print_ml_type:  ml_type -> unit
```

타입 `ml_type` 값의 기본 출력기이다.

```
# type_of (ml_exp_of_string ("let val id = fn x => x " ^
                      "in (id 3, id true) end"));;
val it:  ml_type = (int * bool)

# type_of (ml_exp_of_string ("fn x => x"));;
val it:  ml_type = (v0 -> v0)

# type_of (ml_exp_of_string "(fn x => x x)(fn x => x)");;
Uncaught exception:  <local exception>
```

13.3 파괴적 동일화를 이용하여 타입 유추하기

13.4 요약

타입을 유추하는 두가지 알고리즘을 통해 nML의 대표적인 핵심 부분인 타입에 관한 규칙을 알아보았다. 처음 알고리즘은 치환을 이용하여 일반적 어구에

대한 동일화를 이용하였다. 이 알고리즘은 논문에 나오는 것과 비슷하고, 다음의 성질을 증명할 수 있기 때문에 이론적으로 흥미롭다.

- 정확성: 알고리즘의 결과는 추론 체계의 올바른 연역과정에 해당한다.
- 완전성: 추론 체계의 모든 연역과정은 알고리즘의 결과에 치환을 적용해서 얻을 수 있다.

두번에 살펴본 알고리즘은 좀더 현실적이다. 이것은 동일화를 위해 물리적으로 어구를 확인하는 파괴적 동일화 알고리즘을 사용한다.

13.5 더 배울 내용들

ML 언어의 타입 체계를 처음으로 표현한 것은 Milner [29]이다. [24]에서 최신의 타입 체계 표현을 찾을 수 있으며 여기에는 완전성과 정확성에 대한 증명도 포함되어 있다.

두번째 알고리즘이 상대적으로 현실적이지만, 개체화와 포괄화의 두가지 면에 있어서 비효율적이다.

타입과 타입 개요가 공유 어구를 포함하는 그래프로 표현되었지만, 타입 개요를 개체화 할 때 치환 과정중에 공유가 깨진다. 공유를 보호하기 위해 함수를 다듬으면 효율성이 확실히향상된다. 이를 위해 함수는 개체화 과정 중에 기존 타입 개요에서 이미 전에 방문했었던 노드를 저장하면 된다. 그리고 나서 함수가 이 노드들 가운데 하나를 다시 방문하면 저장된 것을 반환해야 한다.

우연히 타입환경에서 포괄적이지 않은 변수를 계산하는 것은 전체 환경을 찾아봐야 하기 때문에 비싼 연산이다. 이 문제를 해결하기 위하여 타입 변수에 단계라는 개념을 추가해야 한다. 단계는 `let`의 왼쪽의 타입을 계산하는 동안에는 증가하고(즉, `let val x=`$\mathtt{e}_1$ `in` $\mathtt{e}_2$ `end`에서 $\mathtt{e}_1$을 계산하는 동안에는 증가하고), 오른쪽의 타입을 계산하는 동안에는 이전 값으로 돌아간다. 변수 `tv`에 대해 타입 `t`를 동일화 하는 동안에 필요하다면 동일화 알고리즘을 통해 `t`의 변수를 `tv`의 변수로 줄여야 한다. [42]에서 이런 방식으로 단계를 다루는 것을 찾을 수 있다.

$\mathtt{e}_1$을 통해 $\mathtt{t}_1$의 타입을 포괄화 하는 것은 현제 단계에서 $\mathtt{t}_1$에 나타나는 타입 변수의 단계들과 비교하는 것이 된다. 현재 단계나 그 이상의 단계에 있는 변수들은 $\mathtt{e}_1$에 대해 "지역적"이기 때문에 포괄화 될 수 있고, 반대로 더 낮은 단계의 것들은 그렇지 못하다.

아직 성능향상이 가능한 부분이 많다. 특히 동일화 알고리즘에 대해서 많은 부분을 향상시킬 수 있다. 경로를 시험하는 것도 자주 사용되기 때문에 비싼 연산이다. 이런 검사를 가능한 늦추는 것으로 성능을 향상 시킬 가능성이 생긴다. [33]에서 경로 검사를 늦춤으로서 타입 유추 알고리즘을 매우 효율적으로 만든 것을 찾을 수 있다. 또한 방정식의 해를 찾는 것으로 동일화 알고리즘을 바라보는 체계도 설명되어 있다.

참고 자료

[1] H.P. Barendregt. *The lambda-calculus: Its syntax and semantics*. North-Holland, 1987.

[2] R.S. Boyer and J.S. Moore. *A computational logic*. Academic Press, 1979.

[3] M.J. Gordon. *The denotational description of programming languages*. Springer Verlag, 1977.

[4] M.J. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF: a mechanized logic of computation*. Springer Verlag, 1979.

[5] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq proof assistant: A tutorial. Technical Report 204, INRIA-Rocquencourt, 1997.

[6] X. Leroy and P. Weis. *Manuel de référence du langage Caml*. InterÉditions, 1993.

[7] L.C. Paulson. *Logic and computation: Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

[8] J.E. Stoy. *Denotational semantics: the scott-strachey approach to programming languages theory*. MIT Press Series in Computer Science, 1979.

[9] J. van Leeuven. *Handbook of computer science*, volume B. North-Holland, 1990.

[10] P. Weis and X. Leroy. *Le langage Caml*. InterÉditions, 1993.