07장 자료 유형

【7.2.4】 ML의 유형 시스템

예 7.104

ML에서의 피보나치 수열

다음은 6.6.1절에서 소개한 꼬리 재귀 피보나치 수열의 ML 버전이다.

```
1. fun fib (n) =
2.    let fun fib_helper (f1, f2, i) =
3.       if i = n then f2
4.       else fib_helper (f2, f1+f2, i+1)
5.    in
6.       fib_helper (0, 1, 0)
7.    end;
```

let 구성소는 내포된 유효 범위를 도입한다. 함수 fib_helper는 fib 내에 내포되어 있다. fib의 몸체는 수식 fib_helper (0, 1, 0)이다. fib_helper의 몸체는 if...then...else 수식이다. 그래서 fib_helper의 3번째 인자가 n인지 아닌지에 따라 f2나 fib_helper(f2, f1+f2, i+1)로 계산된다.

이 함수 정의가 주어지면 ML 컴파일러는 대략 아래와 같이 추론할 것이다. fib_helper의 매개변수는 4행에 1을 더하기 때문에 유형 int를 가져야 한다. 이와 유사하게 fib의 매개변수 n은 3행에서 그것을 i와 비교하기 때문에 유형 int를 가져야 한다. 6행에서 fib_helper의 호출에서 3개의 인자 모두의 유형은 int다. 그래서 이 문맥에서 적어도 f1과 f2의 유형은 int다. 게다가 i의 유형은 앞선 추론에서의 그것, 즉 int와 모순이 아니고 4행에 재귀 호출에 대한 인자의 유형은 비슷하게 모순이 아니다. fib_helper가 3행에 f2를 반환하기 때문에 6행에 호출의 결과는 int일 것이다. fib가 자신의 결과로 이 결과를 즉시 반환하기 때문에 fib의 반환형도 int다.

예 7.105

수식의 유형

ML이 함수형 언어이기 때문에 ML의 모든 구성소는 수식이다. ML 유형 시스템은 모든 객체와 모든 수식에 대한 유형을 추론한다. 함수가 제1종 값이기 때문에 역시 유형을 가지고 있다. `fib`의 유형은 `int->int`, 즉 정수에서 정수로의 함수다. `fib_helper`의 유형은 `int * int * int -> int`, 즉 정수 3개에서 정수로의 함수다. 상징적 용어에서 `int * int * int`는 3방향 카테시안 곱이다.

ML에서의 유형 정확성은 유형 무모순성이라고 부르는 것에 해당된다. 유형 검사 알고리즘이 어떤 모순과 다중 정의된 이름의 어떤 모호한 출현도 없이 모든 수식에 대해 유일한 유형을 도출해낼 수 있으면 프로그램은 유형적으로 정확하다(고유 산술과 비교 연산자에 대해 ML은 인자가 다른 것으로 결정하지 못한다면 인자가 정수임을 가정한다. 하스켈은 약간 더 일반적이다. 그것은 인자가 요구되는 연산을 지원하는 어떤 유형도 가질 수 있게 한다). 프로그래머가 모순되게 객체를 사용한다면 컴파일러는 이를 알려줄 것이다. 다음 수식을 포함하는 프로그램에서 컴파일러는 `circum`의 매개변수가 `real` 유형을 가짐을 추론하고 그래서 정수 인자를 전달하려고 할 때 이를 알려줄 것이다.

예 7.106

유형의 모순성

```
fun circum (r) = r * 2.0 * 3.14159;
...
circum (7)
```

ML은 일반적으로 인터프리터(해석)하는 대신 컴파일하지만 상호작용적 사용을 위해 만들어졌다. 프로그래머는 ML 시스템과 "온라인"으로 상호작용한다. 그래서 특정 시간에 행을 입력하게 한다. 시스템은 이 입력을 증분적으로 컴파일해 기계어를 함수 이름에 바인딩하고 적절한 컴파일 시 오류 메시지를 발생한다. 이런 류의 상호작용은 해석과 컴파일 간의 전통적인 차이를 흐리게 한다. 하지만 후자를 더 선호한다. 언어 구현은 실행 시간 동안 활성화된 채로 남아있다. 하지만 그것은 프로그램 일부의 실행을 적극적으로 관리하지는 않는다. 그것은 제어를 프로그램에 넘기고 복귀하기를 기다리는 것일 뿐이다.

예 7.107

다형적 함수

프로그래머가 모든 유형을 명시적으로 선언해야 하는 언어와 비교할 때 ML의 유형 추론 시스템은 간결성과 상호작용적 이용에 대한 편리함의 이점을 가지고 있다. 더 중요한 것은 거의 무상으로 묵시적 매개변수 다형성의 강력한 형태를 제공한다는 것이다. 한 ML 프로그램 내 객체의 모든 사용이 모순이 없어야 하지만 완전히 지정될 필요는 없다.

```
fun compare (x, p, q) =
    if x = p then
       if x = q then "both"
       else "first"
    else
       if x = q then "second"
       else "neither";
```

여기서 등가 검사(=)는 'a * 'a -> bool 유형의 고유 다형적 함수, 즉 동일한 유형의 인자 한 쌍을 받아서 논리형 결과를 만드는 함수다. 토큰 'a는 **유형 변수**라고 부른다. 모든 유형을 의미하며 묵시적으로 제네릭 구성소(8.4절과 9.4.4절)에서 명시적인 유형 매개변수의 역할을 맡는다. 주어진 =의 호출에서 'a의 모든 인스턴스는 동일한 유형을 나타내야 한다. 하지만 별개의 호출에서 'a의 인스턴스는 다를 수 있다. =의 유형에서 시작해 ML 컴파일러는 compare의 유형이 'a * 'a * 'a -> string이라고 추론할 수 있다. 그리해 compare는 다형적이다. x, p와 q가 모두 동일한 동안 그것들의 유형에 의존적이지 않다. 관찰할 주요 사항은 프로그래머가 compare를 다형적으로 만들기 위해 어떤 것도 할 필요가 없었다는 것이다. 다형성은 ML 양식의 유형 추론의 자연스러운 결과다.

유형 검사

ML 컴파일러가 잘 정의된 제약 조건 집합에 대해 유형 무모순성을 증명한다. 특히 다음과 같다.

- (유효 범위 내에서의) 동일한 식별자의 모든 출현은 동일한 유형이어야 한다.
- if...then...else 수식에서 조건은 반드시 bool 유형이어야 하고 then과 else 절은 반드시 동일한 유형이어야 한다.
- 프로그래머 정의 함수는, 'a는 함수의 매개변수의 유형이고 'b는 결과의 유형이라고 하면 유형 'a -> 'b를 갖는다. 간략히 볼 것처럼 모든 함수는 한 개의 매개변수를 가진다. 인자로 튜플을 전달해 여러 개의 매개변수의 출현을 가능하게 한다.
- 함수를 적용(호출)하면 전달되는 인자의 유형은 함수의 정의에서 매개변수의 유형과 동일해야 한다. 적용(호출)의 유형은 함수 정의에서 결과의 유형과 동일해야 한다.

두 유형 A와 B가 "동일"해야 하는 모든 경우에서 ML 컴파일러는 공통 유형의 (잠재적으로 좀 더 자세한) 설명을 만들어내기 위해 A와 B에 관해 아는 것을 동일화해야 한다. 예를 들면 컴파일러가 E1이 유형 'a * int(즉, 둘째 원소가 정수로 알려진 2개의 원소를 갖는 튜플)의 수식이고 E2가 유형 string * 'b의 수식임을 결정했었다면 수식 if x then E1 else E2에서 컴파일러는 'a가 string이고 'b가 int임을 추론할 것이다. 그래서 x는 bool이고, E1과 E2는 string * int 유형을 갖는다.

설계와 구현

동일화

동일화는 강력한 기법이다. (C++의 템플릿[제네릭]에서도 발생하는) 유형 추론에서의 역할과 더불어 동일화는 프롤로그와 다른 논리형 언어의 계산 모델에서 중심적인 역할을 한다. 11.1절에서 후자의 역할에 대해 논할 것이다. 일반적인 경우에서 두 수식의 유형을 동일화하는 것의 비용은 지수적이다[Mai90]. 하지만 그렇게 심각한 경우는 실제로 잘 발생하지 않는다.

리스트

대부분의 함수형 언어에서처럼 ML 프로그래머는 리스트를 상당히 많이 이용하는 경향이 있다. 동적으로 유형을 지정하는 (그리고 묵시적으로 다형적인) 리스프와 스킴과 같은 언어에서 리스트는 임의의 유형의 객체를 포함할 수 있다. ML에서 주어진 리스트의 모든 원소는 동일한 유형이어야 하지만 원소에 대한 연산을 수행하지 않고 리스트를 조작하는 함수는 인자로 어떤 종류의 리스트를 취할 수 있다.

예 7.108
다형적 리스트 연산

```
fun append (l1, l2) =
    if l1 = nil then l2
    else hd (l1) :: append (tl (l1), l2);
fun member (x, l) =
    if l = nil then false
    else if x = hd (l) then true
    else member (x, tl (l));
```

여기서 append는 유형 'a list * 'a list -> 'a list를 가진다. 그리고 member의 유형은 'a * 'a list -> bool이다. 예약어 nil은 빈 리스트를 나타낸다. 고유 :: 생성자는 리스프의 cons와 유사하다. 원소와 리스트를 받아서 전자를 후자의 앞부분에 붙인다. 이것의 유형은 'a * 'a list -> 'a list다. hd와 tl 함수는 리스프의 car과 cdr과 유사하다. 각기 ::가 생성하는 리스트의 헤드와 나머지를 반환한다.

예 7.109
리스트 표기법

ML에서는 거의 대부분 "각괄호" 표기법을 사용해 리스트를 작성한다. 토큰 []는 nil과 같다. [A, B, C]는 A::B::C::nil과 같다. 엄밀한 리스트, 즉 nil로 끝나는 것만이 각괄호를 사용해 표현할 수 있다. 실제로 ML에서는 고유 중위 생성자 위에 정의한 append 함수를 @로 제공한다. 수식 [a, b, c] @ [d, e, f, g]는 [a, b, c, d, e, f, g]로 계산된다.

ML 리스트가 동질적(모든 원소가 동일한 유형을 가짐)이기 때문에 nil의 유형에 대해 궁

금할지도 모른다. 이를 어떤 리스트의 유형이라도 가지게 하기 위해 nil을 객체가 아닌 유형 unit->'a list를 가지는 고유 다형적 함수로 정의한다. 고유 유형 unit은 단순히 C의 void와 유사한 위치 지정자다. 인자를 가지지 않는 함수를 유형 unit의 매개변수를 가진다고 말할 수 있다. 부수효과를 위해서만 실행하는 함수(ML은 순수하게 함수적이 아님)를 유형 unit의 결과를 반환한다고 말한다.

다중 정의

등가 검사(=)가 고유 다형적 연산자임을 이미 확인했었다. 이는 크기 비교 검사(<, <=, >=, >)나 산술 연산자(+, -, *)에서는 사실이 아니다. 어떤 유형의 인자라도 받을 수 있기 때문에 등가 검사를 다형적 함수로 정의할 수 있다. 관계 연산자와 산술 연산자는 특정 유형에서만 동작한다. 단일한 유형(예를 들어 인자)의 인자로 한정하는 것을 피하기 위해 ML은 고유 함수의 집합에 대해 이를 다중 정의된 이름으로 정의한다. 이들은 각기 다른 유형(정수, 부동소수점 숫자, 문자열 등)의 객체를 연산한다. 프로그래머는 새로운 유형에 대해 추가적인 함수를 정의할 수 있다.

예 7.110

명시적 유형을 이용한 모호성의 해결

불행하게도 다중 정의는 때때로 유형 추론을 방해한다. 다른 유효한 프로그램 내에서 어떤 함수가 다중 정의된 연산자에 의해 이름이 붙이는지를 해결하기에 충분한 정보가 있지 않을지도 모른다.

```
fun square (x) = x * x;
```

여기서 ML 컴파일러는 *가 정수나 부동소수점의 곱셈 중 어느 것을 참조하기로 되어 있는지를 알 수 없다. 기본적으로 전자를 가정한다. 이것이 프로그래머가 원하는 것이 아니면 명시적으로 대안을 지정해야 한다.

```
fun square (x : real) = x * x;
```

다중 정의된 기호의 해결을 허용하는 것과 함께 명시적인 유형 선언은 ML 프로그램에서 "증명된 문서화"의 역할을 한다. ML 프로그래머는 필요하지 않을 때도 종종 변수에 대한 유형을 선언한다. 왜냐하면 이 선언은 프로그램을 읽고 이해하는 것을 더 쉽게 하기 때문이다. 물론 주석도 가독성을 향상시킬 수 있지만 프로그래머가 지정한 유형은 매우 중요한 이점을 가지고 있다. 즉, 컴파일러는 그들의 의미를 이해하고 객체의 모든 사용을 선언된 유형과 모순되지 않게 함을 보장한다.

패턴 매칭

지금까지의 논의에서 ML과 그것의 유형의 다른 주요 특징, 즉 패턴 매칭을 "간과"하고 있었다. 패턴 매칭의 가장 간단한 형태 중 하나는 2개 이상의 매개변수의 함수에서 나타난다. 엄격히 말하면 그런 함수는 존재하지 않는다. ML에서 모든 함수는 한 개의

인자만을 받아들이지만 이 인자는 튜플일 것이다. 튜플은 그것의 멤버를 이름이 아닌 위치로 식별하는 것을 제외하고 알골계 언어의 레코드와 닮았다. 예를 들면 위에서 정의한 함수 compare는 원소가 3개인 튜플을 인자로 취한다. 아래의 모든 것은 유효하다.

예 7.111

인자 튜플의 패턴 매칭

```
compare (1, 2, 3);
let val t = ("larry", "moe", "curly") in compare (t) end;
let val d = (2, 3) in
    let val (a, b) = d in
       compare (1, a, b)
    end
end;
```

여기서 패턴 매칭은 호출의 매개변수와 인자 사이에서뿐만 아니라 val 구성소의 왼편과 오른편 사이에서도 발생한다(예약어 val은 이름을 선언하는 역할을 한다. 구성소 fun inc (n) = n+1;는 val inc = (fn n => n+1);에 대한 문법적 편의성이다).

예 7.112

ML에서의 swap

좀 더 그럴듯한 예로 원소가 2개인 튜플을 뒤집는 정말 유용한 함수를 정의할 수 있다.

```
fun swap (a, b) = (b, a);
```

ML이 (거의) 함수형이기 때문에 객체의 값을 교환하게 swap을 의도하지 않는다. 오히려 인자로 원소가 2개인 튜플을 받아 그와 대칭인 원소가 2개인 튜플을 결과로서 만들어낸다.

ML에서의 패턴 매칭은 튜플뿐만 아니라 합성형의 어떤 고유 생성자나 사용자 정의 생성자에 대해서도 동작한다. 생성자는 튜플에 대해 사용하는 괄호, 리스트에 대해 사용하는 각괄호, 고유 연산자(::, @ 등) 중 일부와 datatype의 사용자 정의 생성자(다음 페이지에서 볼 것임)를 포함한다. 리터럴 상수를 생성자라고 생각할 수 있고 그래서 튜플 t를 패턴 (1, x)에 대해 일치시킬 수 있다. 이 일치는 t의 첫 원소가 1일 때만 성공할 것이다.

compare (t)나 swap(2, 3)과 같은 호출에서 ML 구현은 컴파일 시에 패턴 매치가 성공할 것인지를 말할 수 있다. 즉, 그것은 그 패턴에 대해 일치되는 값의 구조에 관한 모든 필요한 정보를 안다. 다른 경우에 구현은 어떤 일치가 실패할 것인지를 말할 수 있다. 일반적으로 패턴의 유형과 값을 동일화할 수 없기 때문이다. 좀 더 흥미로운 경우는 패턴과 값이 동일한 값을 갖지만(예를 들어 동일화할 수 있음) 일치의 성공을 실행 시간까지 결정할 수 없는 것이다. l의 유형이 int list이라면 l을 헤더나 꼬리로 "해체"하려는 시도는 l의 값에 따라 성공하거나그렇지 않을 수도 있다.

예 7.113

실행 시간 패턴 매칭

```
let val head :: rest = l in ...
```

l이 nil이면 시도된 일치는 실행시간에 예외를 발생시킬 것이다(8.5절에서 예외에 대해 더 다룰 것이다).

예 7.114
ML의 case 수식

함수 호출과 val 구성소에서 패턴 매칭이 어떻게 동작하는 지를 확인했다. case문도 이를 지원한다. case를 이용해 위 append 함수를 아래와 같이 작성할 수 있다.

```
fun append (l1, l2) =
    case l1 of
       nil => l2
    | h :: t => h :: append (t, l2);
```

여기서 case에 대해 생성된 코드는 먼저 l1을 nil에 대해 패턴 매칭을 하고 나서 h :: t에 대해 할 것이다. 이 case문은 일치하는 패턴을 가지고 있는 첫 가지 내의 => 다음에 나오는 하위 수식으로 계산된다. 가지의 패턴이 고갈되지 않거나 이전의 가지의 패턴이 나중에 나오는 가지의 패턴을 완전히 포함하면(나중에 나오는 것을 전혀 선택하지 못할 것을 의미함) 컴파일러는 컴파일 시에 경고 메시지를 발생시킬 것이다.

예 7.115
case 레이블의 범위

무용한 가지는 일종의 오류겠지만 동적 의미 오류 메시지를 발생시키지 않는다는 면에서 해롭지는 않다. 프로그래머가 실행 시간에 그 패턴이 항상 동작할 것이라고 예측 가능하면 비고갈(nonexhaustive)인 경우는 고의적일지도 모른다. append 함수를 아래와 같이 작성한다면 그런 경고를 발생시킬 것이다.

```
fun append (l1, l2) =
    if l1 = nil then l2
       else let val h::t = l1 in h :: append (t, l2) end;
```

여기서 컴파일러는 l1이 비어 있지 않으면 else 절 내의 let 구성소를 정교화할 것이라는 것을 깨닫지 못할 가능성이 크다(이 예는 이해하기에 충분히 쉬워 보인다. 하지만 일반적인 경우 이를 계산할 수 없다. 그리고 대부분의 컴파일러는 쉬운 경우를 인지하기 위한 특별한 코드를 가지고 있지 않을 것이다).

예 7.116
대안으로서의 함수

함수의 몸체가 전부 case문으로 이뤄져 있을 때 그것을 간단한 대안으로 작성할 수 있다.

```
fun append (nil, l2) = l2
| append (h::t, l2) = h :: append (t, l2);
```

패턴 매칭은 다른 언어, 특히 문자열에 중점을 두는 언어(스노볼, 아이콘과 펄)에서 아주 중요하게 취급된다. ML 양식의 패턴 매칭은 정적 유형 지정과 유형 추론의 통합한다는 점에서 문자열 지향 언어의 그것과 다르다. 스노볼, 아이콘과 펄을 모두 동적으로 지정한다.

다중 인자 함수를 튜플로 변환해 ML은 대다수의 다른 언어에서의 함수의 인자와 반환 값 간의 비대칭성을 제거한다. swap에서 보여줬듯이 어떤 함수는 정확히 튜플 인자를

취하는 것만큼 쉽게 튜플을 반환할 수 있다. 패턴 매칭은 튜플의 원소를 호출자가 추출하게 한다.

예 7.117
반환 튜플의 패턴 매칭

```
let val (a, b) = swap (c, d) in ...
```

여기서 a는 d가 주는 값을 가질 것이다. 그리고 b는 c가 주는 값을 가질 것이다.

datatype 생성자

리스트와 튜플에 더불어 ML은 프로그래머가 다른 종류의 합성형을 도입하게 하는 datatype 메커니즘과 함께 레코드에 대한 고유 생성자를 제공한다. 레코드는 원소가 이름을 갖지만 특정한 순서를 가지지 않는 합성 객체이다(언어 구현은 내부적인 표현을 위해 어떤 순서를 선택해야 하지만 이 순서는 프로그래머에게 보이지 않는다). 레코드를 {name= "Abraham Lincoln", elected =1860}과 같이 "중괄호" 생성자를 이용해 지정한다(동일한 값을 {elected =1860, name= "Abraham Lincoln"}로 표기할 수 있다).

예 7.118
ML의 레코드

ML의 datatype 기법은 유형 이름과 그 유형에 대한 한 무리의 생성자를 도입한다. 가장 간단한 경우에서 생성자는 모두 인자가 없는 함수고 그 유형은 본질적으로 열거형이다.

예 7.119
ML의 datatype

```
datatype weekday = sun | mon | tue | wed | thu | fri | sat;
```

좀 더 복잡한 예에서 생성자는 인자를 갖는다. 그리고 유형은 본질적으로 공용체(베리언트 레코드)다.

```
datatype yearday = mmdd of int * int | ddd of int;
```

이 코드 mmdd를 한 쌍의 정수를 인자로 취하는 생성자로, 하나의 정수를 인자로 취하는 생성자로 ddd를 정의한다. 그 의도는 해당년도의 날짜를 (월, 일)로나 1에서 366 사이의 정수로 지정하게 하는 것이다. yearday를 그것 자신의 특별한 등가 연산을 이용해 (3.3.4절의 유클리드 모듈 유형과 유사한) 추상 유형으로 만들지 않았다면 등가 검사는 mmdd (7, 4) = ddd (188)을 실패할 것이지만 7월 4일을 mmdd (7, 4)나 ddd (188)로서 표현할 수 있다.

예 7.120
재귀 datatype

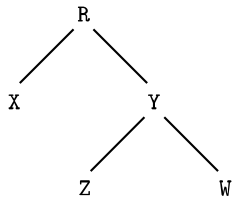
포인터를 사용하지 않고 재귀형을 정의하기 위해서 ML의 datatype을 사용할 수도 있다. 특징적인 ML의 예는 이진 트리다.

```
datatype int_tree = empty | node of int * int_tree * int_tree;
```

정의에 명시적인 유형 변수를 도입하면 임의의 동일 유형 엘리먼트로 구성된 제네릭 트리를 생성할 수도 있다.

```
datatype 'a tree = empty | node of 'a * 'a tree * 'a tree;
```

이런 정의가 주어지면 다음과 같은 트리를



`node (#"R", node (#"X", empty, empty), node (#"Y", node (#"Z", empty, empty), node (#"W", empty, empty)))`으로 작성할 수 있다. 재귀 유형도 리스프, 클루, 자바, C#과 변수의 참조 모델을 사용하는 다른 언어에도 나타난다. 7.7절에서 이에 대해 더 자세히 논한다.

예 7.121

ML에서의 유형 등가

유형 추론의 사용으로 인해 ML은 보통 구조적인 유형 등가의 효과를 제공한다. 이름 등가의 효과를 얻기 위해 datatype의 정의를 사용한다.

```
datatype celsius_temp = ct of int;
datatype fahrenheit_temp = ft of int;
```

그러면 `ct` 생성자를 사용해 유형 `celsius_temp`의 어떤 값을 얻을 수 있을 것이다.

```
val freezing = ct (0);
```

불행하게도 `celsius_tmp`는 자동으로 `int`의 산술 연산자와 관계를 상속하지는 않는다. 프로그래머가 이 연산자를 명시적으로 정의하지 않으면 수식 `ct (0) < ct (20)`은 "해당 유형에 대해 정의되지 않은 연산자"의 행과 함께 에러 메시지를 발생시킬 것이다.

✓ 확인문제

55. ML 컴파일러는 어떤 상황에서 유형 충돌을 선언하는가?
56. ML의 유형 추론이 어떤 방법으로 자연스럽게 다형성을 이끌어내는지를 설명하라.
57. 유형 변수란 무엇인가? ML 프로그래머가 이런 변수를 명시적으로 사용할 수도 있는 예를 제시하라.
58. ML의 리스트가 리스프와 스킴의 리스트와 다른 점은?
59. 그럴 필요가 없음에도 불구하고 ML 프로그래머가 종종 변수의 유형을 선언하는 이유는?

60. 동일화는 무엇인가? ML에서의 역할은?

61. ML가 패턴 매칭을 수행하는 3가지 문맥을 나열하라.

62. ML에서의 튜플과 레코드 간의 차이점을 설명하라. ML 레코드는 알골계 언어에서의 레코드(구조체)와 어떻게 다른가?

63. ML의 `datatype`은 무엇인가? 이것은 C와 파스칼과 같은 명령형 언어에서의 어떤 특징을 포함하는가?
