Coq

Based on Software Foundations

Joohee Jeong

2024년 3월 26일

차 례

제 1 장 들어가기 1

1.1 Coq이란 . . . 1

1.2 Coq의 작업환경 . . . 1

1.3 Software Foundations . . . 4

제 2 장 Basics 7

2.1 Data and Functions . . . 7

2.2 증명 기법 . . . 22

제 3 장 Induction 31

3.1 Separate Compilation . . . 31

3.2 Proof by Induction . . . 32

3.3 Exercises . . . 36

제 4 장 Lists 45

4.1 Pairs of Numbers . . . 45

4.2 Lists of Numbers . . . 47

4.3 Partial Maps . . . 59

제 5 장 Poly 63

5.1 Polymorhpism . . . 63

5.2 Functions as Data . . . 68

5.3 Church Numerals . . . 75

제 6 장 Tactics 79

6.1 Part 1 . . . 79

6.2 Part 2 . . . 90

제 7 장 Logic 101

7.1 Propositions and Predicates . . . 101

7.2 Logical Connectives . . . 102

7.3 Existential Quantification and More . . . 110

7.4 Working with Decidable Properties . . . 120

7.5 The Logic of Coq . . . 127

제 8 장 Inductively Defined Propositions 135
8.1 Inductively Defined Predicates 135
8.2 Using Evidence in Proofs 148
8.3 Inductive Relations 154
8.4 Case Study: Regular Expressions 168
8.5 Case Study: Improving Reflection 192
8.6 More Exercises 195

제 9 장 The Curry-Howard Correspondence 205
9.1 Proof Scripts and Proof Terms 205
9.2 Logical Connectives as Inductive Types 211
9.3 Equality 219
9.4 More Stuff 223

제 10 장 Induction Principles 229

제 11 장 Properties of Relations 239

제 12 장 Total and Partial Maps 251

제 13 장 IMP, Simple Imperative Programs 259
13.1 Arithemetic and Boolean Expressions 259
13.2 Coq Automation 263
13.3 Evaluation as a Relation 268
13.4 Expressions with Variables 274
13.5 Commands 277
13.6 Evaluating Commands 279
13.7 Reasoning About Imp Programs 285

참고문헌 293

Index 294

1 들어가기

1.1 Coq이란

Coq은 수학적 증명을 돕는 소프트웨어Proof Assistant이며 1989년 첫 버전이 공개되었다. 이 소프트웨어가 제공하는 Coq 언어는 수학의 명제를 엄격하고 명확하게 표현할 수 있으며, 이 언어로 쓰인 수학적 증명은 오류가 있는지의 여부를 정확히 확인할 수 있다. 또한 Coq은 주어진 명제에 대한 증명을 자동으로 찾아내지는 못하지만 증명을 찾고자 하는 사람에게 적지않은 도움을 줄 수 있는 자동증명 책략tactics들을 제공한다.

현대 수학의 기반을 제르멜로-프랑켈 집합론Zermelo-Fraenkel Set Theory(ZF or ZFC)이라고 한다면 Coq은 *Calculus of Inductive Constructions*(CoIC)에 기반을 두고 있다.

CoIC는 *Calculus of Constructions*(CoC)에 *inductive type*을 추가하여 얻은, 타입이론type theory의 일종이다. CoC의 창시자는 프랑스의 수학자/전산학자인 Gérard Huet과 Thierry Coquand이다. Coq은 CoIC를 컴퓨터에 구현한 것이라고 말할 수 있다. CoC와 CoIC는 수학의 구성적 기반constructive foundations of mathematics인 한편 프로그래밍 언어인 것으로도 볼 수 있다. 이들은 Alonzo Church가 창시한 *Lambda Calculus*, 조금 더 구체적으로는 Henk Barendregt가 정립한 *Lambda Cube*를 바탕으로 이보다 더 풍부한 표현력을 가지며 또한 컴퓨터 구현에 적합하도록 발전된 것이다.

Coq의 발전 과정에서 지대한 역할을 한 인물로 스웨덴의 철학자/통계학자인 Per Martin-Löf가 있다. 그가 창시한 *Intuitionistic Type Theory*는 논리학과 컴퓨터 과학을 연결해 주는 역할을 하였다. [1]

CoC의 이론을 공부하기 위한 좋은 책으로 [3]을 꼽을 수 있다. CoC의 실제 구현, 즉 Coq에 대한 책으로는 [2]와 [4]가 있다. [2]는 Coq의 이론적인 측면을 다루고, [4]는 Coq의 실용적인 측면을 다룬다. [4]는 6개의 권Volume으로 되어 있는데, 이 책은 이들 중 첫 두 권, *Logical Foundations*와 *Programming Language Foundations*의 내용을 정리한 것이다.

1.2 Coq의 작업환경

Coq을 설치하고 사용하기 위한 모든 정보는 공식 웹사이트 `https://coq.inria.fr` [5]에서 얻을 수 있다. Coq의 작업환경은 상당히 다양하게 많이 있는데 이들 중에 가장 널리 알려진 것들 몇

개를 소개하면 다음과 같다.

- coqtop: 이것은 CLI, 즉 텍스트 터미널 환경에서 명령을 내려 실행하는 형식의 인터페이스를 제공한다.
- coqide: Coq의 공식 IDE이다. 좀 오래 되었지만 나무랄 데 없는 환경을 제공한다. 윈도즈 PC 사용자는, 특히 컴퓨터 전공자가 아니라면 아마도 이것이 가장 좋은 선택일 것이다. https://coq.inria.fr/download에서 Windows Installer 파일을 받아 설치하면 된다.
- Proof General: Emacs를 기반으로 한다.
- Coqtail: Vim을 기반으로 한다.
- jsCoq: 비교적 근래에 나온 것으로 Web에서 동작하므로 아무런 소프트웨어의 설치가 필요없다. https://coq.vercel.app/
- VsCoq: VS Code를 기반으로 한다. 속도가 빠르며 사용하기에 편하다. 하지만 조금 불안정한 면이 있어 때때로 VS Code를 재시작 해야 한다.
- coq-lsp: 또 하나의 VS Code extension이다.

다음은 VsCoq을 맥(Mac)에 설치하는 방법이다. 다른 시스템에서 설치하는 방법은 [5]를 참고하기 바란다.[1]

VsCoq는 Coq을 VS Code 에디터에서 사용할 수 있도록 하는 익스텐션$_{\text{extension}}$이다. VsCoq를 사용하기 위하여는 먼저 Coq을 설치해야 하는데 여기서는 opam을 통해서 설치하는 방법을 설명하겠다. Docker를 사용하는 방법도 있는데 이렇게 하면 Coq를 따로 설치하지 않고도 Coq를 사용할 수 있다고 한다. Docker 방법에 대해서는 이 책에서 다루지 않는다.

https://coq.inria.fr/opam-using.html에 보면 다음과 같이 나와 있다.

> Opam is the package manager for the OCaml programming language, the language in which Coq is implemented. Opam 2.1 is the recommended version, and is assumed below. Instructions on how to install opam itself are available on the opam website.

opam이 설치되어 있지 않다면 설치해야 한다. 설치방법은 opam.ocaml.org/doc/Install.html에 나와 있다. 여러 방법들 중에 간편히 다음과 같이 하는 것을 추천한다. 터미널에서 다음 명령을 실행한다.

```
brew install opam
```

이렇게 하면 opam이 설치되지만 아직은 사용할 수 없다. 다음의 명령을 실행한다.

```
opam init
```

좀 지루하게 기다리면 다음의 메시지가 나온다.

```
Done.
# Run eval $(opam env --switch=default) to update the current shell environment
```

[1] VsCoq을 윈도즈 PC에 설치하는 방법은 [5]에도 잘 나와 있지 않다. VsCoq을 사용하려면 opam이 설치되어야 하는데 현재 버전인 Opam 2.1은 윈도즈를 지원하지 않는다.

시키는 대로 eval $(opam env --switch=default) 명령을 실행한다. 화면에 별 변화는 없지만 이제 opam을 사용할 수 있다.

opam이 설치되었으니 이제 다음과 같이 하여 Coq을 설치한다. (이 설명은 Coq의 공식 웹사이트에서 [User Interfaces – VsCoq (VS Code)]를 클릭하여 접속되는 웹사이트에 나와 있다. 이 사이트의 웹주소는 github.com/coq-community/vscoq이다.)

```
opam pin add coq 8.18.0
```

좀 지루하게 기다렸다가 설치가 끝나면 같은 요령으로 다음의 명령을 실행한다.

```
opam install vscoq-language-server
```

설치가 끝나고 Done이 나타나면 다음의 명령으로써 설치가 되었음을 확인한다.

```
which vscoqtop
```

다음에는 VsCoq을 설치한다. 이것은 VS Code의 익스텐션이므로 VS Code가 미리 설치되어 있어야 함은 당연하다. VS Code가 실행되는 상태에서 [Shift-Command-X]를 쳐서 익스텐션 모음 화면을 열고 'vscoq'를 검색하면 3개의 항목이 뜨는데 이들 중에 VsCoq를 [Install]하면 된다. 설치가 완료되면 다음과 같은 메시지가 나타난다.

> Finally, go to the extension settings and enter the vscoqtop full path from above in the field "Vscoq: Path".

이 지시를 따르려면 [Cmd ,]를 쳐서 settings 화면으로 간다. 왼쪽 패널의 맨 아래 VsCoq가 보일 것이다. 그걸 클릭하면 Path 입력난이 보인다. 여기에 넣을 것은 which vscoqtop의 출력 문자열이다. 대략 다음과 같이 보일 것이다.

```
/Users/myusername/.opam/default/bin/vscoqtop
```

VsCoq의 설정을 마무리 한 후에 위의 디렉토리에 가 보면 coqc, coqtop, ocaml 등의 실행 파일들이 많이 보인다. 이 디렉토리를 PATH에 추가하는 등의 환경설정을 하려면 아까 보았던 명령

```
eval $(opam env --switch=default)
```

를 터미널에서 실행하면 된다. 터미널을 열 때마다 이 명령을 실행하기 귀찮다면 자동실행되도록 적당한 방법을 써서 만들어 놓도록 한다. CLI를 사용하지 않고 VS Code에서만 작업할 것이라면 굳이 이럴 필요는 없을 것이다.

혹시 coqide를 사용하고 싶다면 다음과 같은 명령을 내려 설치한다.

```
opam install coqide
```

설치 후에 터미널에서 coqide 명령을 내리면 Coq의 GUI가 나타난다.

VsCoq을 사용하다 보면 때때로 vscoq-language-server를 업그레이드 해야 한다는 메시지가 나타날 것이다. 이럴 때는 터미널에서 다음과 같이 하면 된다. 작업을 마치려면 시간이 좀 걸릴 것이니 참을성을 가지고 기다려야 한다. 그리고 이 작업을 마친 후에는 §3.1에서 설명한 make 작업(커맨드 2개 실행)을 해야 한다.

```
eval $(opam env --switch=default)
opam update
opam upgrade
```

설치가 끝난 뒤에 VsCoq를 사용할 때는 Coq 소스 파일들이 들어 있는 디렉토리를 열어야 한다. 소스파일은 .v 확장자를 가진다. 이런 확장자를 가지는 파일을 열고 코드를 넣으면 VsCoq가 자동으로 활성화 된다. 그리고 Coq Goals라는 이름의 윈도우가 자동으로 열릴 것이다. 이 윈도우에는 현재 증명하고자 하는 고울$_{\text{goal}}$과 메시지$_{\text{message}}$가 나타난다.

Coq에 경험이 있다면 이제 VsCoq를 어느 정도 자유롭게 사용할 수 있을 것이다. 그렇지 않다면 계속 이 책을 읽어 사용법을 익히도록 한다.

1.3 Software Foundations

Software Foundations (SF) 는 이 책의 주된 참고자료이다[4]. 이것은 6 권으로 이루어진 온라인 책이다. 앞서 말했듯이 이 책에서는 SF의 제1권 Logical Foundations와 제2권 Programming Language Foundations를 공부할 것이다.

SF 웹페이지에서 [Volume 1]을 클릭하고 이어서 [READ]를 클릭하면 제1권의 목차가 나타난다. Preface부터 Bibliography (Bib)까지 총 20개의 장으로 구성되어 있다. 각 장의 제목을 클릭하면 내용을 읽을 수 있다.

SF를 본격적으로 읽기 전에 [DOWNLOAD]를 클릭하여 책의 소스파일과 HTML 파일을 다운로드 받아야 한다. 다운로드 받은 파일의 이름은 lf.tgz이다.(Volume 2의 경우에는 plf.tgz이다.) 압축을 풀면 lf라는 디렉토리가 나타날 것이다. 이 디렉토리 lf를 내가 원하는 적당한 디렉토리에 넣어 두고 VS Code로 lf를 열어보면 여러 개의 파일들이 보이는데 이들의 확장자는 주로 .v 아니면 .html이다.

VS Code에 Live Preview 등의 익스텐션이 설치되어 있다면 index.html을 열어서 VS Code 내에서 SF의 모든 내용을 볼 수 있다.

Preface는 비교적 짧은 장인데, 이것을 다 읽은 후에는 다음 장인 Functional Programming in Coq (Basics)를 열어 본다. 그리고 이 장에 대응되는 Coq 소스 파일인 Basics.v를 VS Code에서 열도록 한다.

Basics.v의 본격적 공부는 다음 장에서 시작할 것이며 여기서는 VsCoq의 인터페이스를 둘러보기만 하겠다. Basics.v의 코드 부분만 보도록 한다. 첫 두 코드블럭을 보면 다음과 같다.

```
1  Inductive day : Type :=
2    | monday
3    | tuesday
4    | wednesday
5    | thursday
6    | friday
7    | saturday
8    | sunday.
9
10 Definition next_weekday (d:day) : day :=
11   match d with
12   | monday    => tuesday
13   | tuesday   => wednesday
14   | wednesday => thursday
15   | thursday  => friday
16   | friday    => monday
```

```
17    | saturday  => monday
18    | sunday    => monday
19    end.
```

라인 1의 day를 dy로 바꿔 보라. 그러면 라인 10의 day에 빨간 밑줄이 나타날 것이다. 여기에 마우스를 올리면 day는 정의된 적이 없다는 내용의 메시지가 나타난다. 다시 라인 1의 dy를 day로 바꾸면 빨간 밑줄이 사라질 것이다.

이번에는 라인 6의 friday 뒤에 마침표를 넣어 friday.로 바꿔 보라. 그러면 라인 7의 맨 왼쪽 |에 빨간 밑줄이 그어지고, 저 아래 있는 라인 17에도 빨간 밑줄이 그어진다. 왜냐하면 라인 1에서 시작된 Inductive day : Type의 정의는 마침표로써 끝나게 되어 있으므로 friday.에서 정의가 끝나게 된다. 따라서 라인 7과 라인 8의 saturday와 sunday는 제대로 정의가 되지 않는다. 라인 17에는 정의되지 않은 saturday가 나타나므로 빨간 밑줄이 그어져 오류를 알려주는 것이다. 그런데 라인 8의 sunday에 대해서는 오류가 지적되지 않는다. 왜냐하면 같은 종류의 오류가 여러 번 나타나면 맨 처음 것에만 경고가 나타나기 때문이다.

코드를 임의로 조금씩 바꿔가면서 어디에 빨간 밑줄이 나타나는지 관찰하고, 그 밑줄 부분에 마우스를 올리면 나타나는 메시지를 읽어 보라. 물론 바꾼 코드는 원래 상태로 되돌려 놓아야 한다.

지금까지는 Coq Goals 윈도에[2] 아무런 변화가 없었을 것이다. 이제 next_weekday 함수를 사용하여 계산을 해 보자. 그 계산 결과는 Coq Goals 윈도의 Messages 영역에 나타나게 된다. 다음의 코드에서 각 라인을 끝맺는 마침표의 오른쪽에 캐럿caret, 즉 키보드 커서를 두면 이 값을 볼 수 있다.

```
Compute (next_weekday friday).
Compute (next_weekday (next_weekday saturday)).
```

Coq Goals의 Proof 영역에는 Not in proof mode만 계속 덩그러니 나타나 있었을 것이다 이제 이 영역에 변화를 주어 보자.

```
1 Example test_next_weekday:
2   (next_weekday (next_weekday saturday)) = tuesday.
3 Proof. simpl. reflexivity.  Qed.
```

라인 2의 마침표 오른쪽에 캐럿을 두면 Proof 영역에 Goal 1이 보이게 되고 그 밑의 수평선 아래에 라인 2의 내용이 나타난다. 이 명제를 증명하는 것이 현재의 목표goal라는 뜻이다. 현재는 목표가 하나뿐이므로 Goal 1만 보이지만 목표가 여러 개라면 Goal 1, Goal 2, Goal 3, ... 이런 식으로 나타나게 된다.

라인 3에는 마침표가 4개 보이는데, 오른쪽/왼쪽 화살표 키를 이용하여 캐럿이 마침표를 좌우로 지나면서 Proof 영역에 어떤 변화가 나타나는지 관찰하여라.

Coq의 작업 환경 설정을 위하여 아직 할 일이 남아 있는데, 이에 대해서는 Functional Programming in Coq (Basics)의 다음 장인 Proof by Induction (Induction)의 첫 섹션인 §3.1

[2] Coq Goals 윈도는 자동으로 열려 있을 것임.

Separate Compilation에 상세하게 설명이 나와 있다. 우리도 그때 가서 이 설정 작업을 할 것이다.

다만 `Basics.v` 파일에서 직접 코딩 실습을 한다면 이 과정에서 파일이 망가져서 환경 설정 작업이 잘못될 위험이 있다. 이를 방지하기 위하여 `Basics.v` 파일의 복사본 `Basics_copy.v`를 만들어 여기서 실습을 하는 것도 하나의 방법이 될 것이다.

2

Basics

2.1 Data and Functions

Defining Types and their Elements

Coq에서 데이터 타입을 만들 때는 Inductive 키워드를 사용한다. 유한 개의 원소를 가지는 데이터 타입, 혹은 유한집합은 단순한 나열enumeration로써 정의할 수 있다. 예를 들어 다음과 같이 day라는 데이터 타입을 정의할 수 있다.

```
1 Inductive day : Type :=
2   | monday
3   | tuesday
4   | wednesday
5   | thursday
6   | friday
7   | saturday
8   | sunday.
```

day는 위에 나열하여 보인 7개의 원소로 구성된다. 식별자identifier a와 b에 각각 day 타입의 값을 하나씩 배정해 보자. Definition 키워드를 사용하여 다음과 같이 한다.

```
Definition a: day := monday.
Definition b: day := friday.
```

타입(*type*)은 집합set의 동의어라고 생각하면 된다. 다만 우리는 a ∈ day 대신 a : day 라고 쓴다.[1] 참고로 이렇게 정의된 a 등은 변수가 아니라 상수이다. 다른 값을 a에 배정하면, 혹은 같은 값을 배정해도 마찬가지지만, 에러가 난다.[2]

a가 무엇인지 알아보는 데 사용하는 키워드, 혹은 명령command으로는 Check, Compute, Print의 3개가 있다. 이 명령들은 a 뿐만 아니라 day, monday 등에도 사용할 수 있으며 이에 대해서는 조금 후에 다시 설명하겠다. 먼저 다음의 코드를 실행해 보자.

---

[1]실은 이렇게 단순하지는 않다. 타입과 집합이 같은 것이라면 왜 타입이론과 집합론이 따로 존재하겠는가? 아주 짧게 말하자면 타입은 *intensional*하고 집합은 *extensional*하다. 집합은 주어진 원소가 그것에 속하는지 여부에만 관심이 있다. 타입은 주어진 원소가 어떻게 만들어졌는지를 따진다. 따라서 주어진 원소를 포함하는 집합은 얼마든지 있다. 하지만 주어진 원소의 타입은 단 하나뿐이다. (적어도 이 책에서 다루는 타입이론에서는 이렇다.)

[2]Coq와 같은 함수형 프로그래밍 언어에서는 변수에 대한 배정assignment이라는 개념이 없다.

```
Check a.
Compute a.
Print a.
```

각 라인의 끝에 캐럿을 가져다 놓고 `Messages` 영역에 무엇이 나타나는지 관찰하여라.

① `Check`는 `a`의 타입을 보여주고,

② `Compute`는 `a`의 값을 보여주며,

③ `Print`는 `a`의 정의를(정확히 말하자면 정의할 때 사용되었던 코드를) 보여준다.

`Compute`의 결과 값은 항상 `=` 뒤에 보여준다는 점에 주목하여라.

우리는 `a` 뿐만 아니라 임의의 표현항(*term*) *t*에 대해서도 `Check` *t*, `Compute` *t*, `Print` *t*를 사용할 수 있다.[3] *t*가 문법에 맞는 표현well-formed expression이라면 이들 명령은 각각 타입, 값 및 정의를 보여주고, 만일 문법에 맞지 않는 표현ill-typed expression이라면 에러가 발생한 부분에 붉은 밑줄이 그어진다. 에러의 내용을 알려면 마우스를 그 부분에 올리면 된다.
다음과 같은 구문도 가능하다. 다만 라인 3에서 보듯이 `Print`는 안 된다.

```
1 Check a: day.
2 Compute a: day.
3 Print a: day. (* error *)
```

라인 3에서 사용된 `(* ... *)`는 코멘트(*comment*)를 나타내는 구문이다. 코멘트는 여러 줄에 걸쳐 쓸 수 있다.

`day`의 원소인 `monday`, `tuesday` 등을 생성자(*constructor*)라고 한다. 생성자라는 용어는 원소를 생성한다는 뜻이다. 그런데 `monday`, `tuesday` 등의 생성자는 상수, 즉 nullary이기 때문에 생성자는 곧 원소이다. 따라서 굳이 생성자라고 부를 필요가 없는 것으로 보이겠지만, unary, binary 등의 생성자를 사용하게 되면 생성자라는 이름이 의미를 가지게 된다. 함수(상수가 아닌) 생성자의 예는 조금 있다가 들어 보기로 하고, 생성자에 `Check`, `Compute`, `Print`를 사용하면 어떻게 되는지 보자.

```
1 Check monday.
2 Compute monday.
3 Print monday.
```

라인 3의 출력은 `monday`가 속하는 타입 `day` 전체의 정의에 사용된 코드이므로 꽤 크다는 것에 주목하여라. `Check`를 가지고 좀 더 실험해 보자.

```
Check day. (* day : Set *)
Check Set. (* Set : Type *)
Check Type. (* Type : Type *)
```

`Set`는 뭔가? 타입 이론에서 `Type`이면 충분한 것 아닌가 하는 의문이 들 수 있다. `Type`들 중에 계산의 대상이 되는 데이터 타입을 `Set`라고 한다고 말할 수 있다. `Set`는 `Type`의 부분집합으로 보아도 되고 원소로 보아도 된다.

p7에서 보았던 `day`의 정의

---

[3] 표현항은 표현expression들 중에 어떤 조건을 만족하는 것들을 뜻한다. '표현'은 형식언어formal language에서 널리 사용되는 용어인데 언어에 따라 의미가 다르다. `Coq`에서는 *evaluate* 했을 때 어떤 타입의 원소가 되는 표현을 표현항이라고 한다.

```
Inductive day : Type :=
  | monday
  ..
```

는 다음과 같이 바꿔 써도 된다.

```
Inductive day : Set :=
  | monday
  ..
```

Type들 중에 특수한 것이 Prop이다. 이것은 명제proposition를 나타내는 타입이다. Prop의 *inhabitant*, 즉 원소는 명제이며, 명제는 커리-하워드 대응에 의하여 다시 타입인 것으로 취급된다.[4] 명제의 inhabitant는 그 명제의 증명proof, or evidence(들)이다. 이 토픽에 대해서는 나중에 자세히 공부하게 될 것이다.

Function Types

앞서 상수를 정의할 때 사용한 Definition 키워드는 함수의 정의에도 사용된다. 다음의 정의에서 함수의 이름은 next_weedkay이고 day 타입 인수를 받아서 day 타입 값을 리턴한다.

```
1  Definition next_weekday (d: day) : day :=
2    match d with
3    | monday    => tuesday
4    | tuesday   => wednesday
5    | wednesday => thursday
6    | thursday  => friday
7    | friday    => monday
8    | saturday  => monday
9    | sunday    => monday
10   end.
11
12 Check next_weekday. (* next_weekday : day -> day *)
```

라인 12의 코멘트에 등장하는 화살표 ->는 함수 타입을 나타낸다. 일반적으로 T: Type에서 S: Type으로 가는 함수의 타입은 T -> S: Type이다.

여기서 새로 등장한 키워드는 match, with, end 이다. 함수의 인수 d의 값에 따라 함수의 리턴값이 결정되는데, d의 값은 유한집합 day의 원소들 중 하나이므로 가능한 모든 경우를 나열하여 각각의 경우에 리턴값을 지정해 줄 수 있다.

이 예에서는 match가 인수를 그 인수 타입의 원소에 매치시켰지만, 일반적으로는, 생성자가 nullary가 아닐 수 있으므로, 인수를 생성자에 매치시킨다고 하는 것이 옳은 표현이다. 함수의 리턴값은 Compute를 사용하여 얻을 수 있다. Compute는 앞서 보았듯이 함수 표현이 아닌 상수 표현에도 적용할 수 있다. 리턴값은 표현항의 계산값evaluated value이라고 해도 된다.

```
1 Compute (next_weekday friday). (* = monday : day *)
2 Eval compute in (next_weekday friday).
3 (* `Eval compute in exp` is the same as `Compute exp`. *)
4
5 Compute (next_weekday (next_weekday saturday)). (* = tuesday : day *)
```

[4] Inhabitant는 원소element의 동의어이다. 타입이론에서는 원소 대신 inhabitant를 쓰는 것이 일반적이다.

Coq에서는 함수의 인수에 괄호를 사용하지 않는 것이 원칙이다. 이는 라인 1을 보면 알 수 있다. 라인 5에서는 next_weekday 함수를 두 번 적용했는데, 첫 번째 적용에서의 인수에는 괄호를 사용하여야 한다. 그 이유는 next_weekday는 인수로 day 타입 표현을 받아야 하기 때문이다.

라인 1과 라인 5에서 Compute를 Check로 바꾸면 어떤 결과가 나오는가? 그리고 만일 Print로 바꾸면?

함수 표현(함수와 인수들로 이루어진)을 계산한evaluate한 값을 보려면 Compute를 함수 표현에 사용하고, 함수의 정의를 보려면 Print를 함수 이름에 사용한다고 알고 있으면 된다.

Proposition, Theorem, Proof

이 절은, 실습을 통해서 공부하는 것을 염두에 두고 쓴 이 책의 성격과 거리가 있다. 내용이 좀 추상적이며, 원하지 않는 독자라면 이 절을 건너 뛰어도 좋다. 혹은 이 책을 읽는 도중에 이 절로 돌아와서 읽어도 좋다. 다만 Coq의 증명의 인터페이스를 익히기 위하여 이 절의 내용 중에 Theorem my0, Definition n_eq_n_prop 및 Theorem n_eq_n_thrm은 꼭 읽도록 한다.

증명의 대상이 되는 명제를 Coq에서는 *proposition*이라고 부른다. 이러한 명제를 나타내는 표현이 proposition이라고 말해도 될 것이다. Proposition을 우리말로 프랍이라고 하겠다. Prop 타입의 inhabitant가 곧 프랍이다.

논리학의 핵심 개념인 명제, 정리, 증명에 대한 Coq의 정의는 전통적classical 1계논리(수리논리학이라고 부르기로 한다)에서의 정의와 다른 점이 많다. 수리논리학의 기본 지식을 갖춘 사람이 이 부분을 확실히 해 두지 않고 Coq 공부를 시작하면 어려움을 겪게 될 수 있다.[5]

20세기 초반에 이루어진 현대 수리논리학의 진보는 구문form, syntax과 내용content, semantics의 분리에서 시작되었다고 본다. 구문론적 대상은 자연수만큼이나 수학적으로 명확하게 정의되므로 수학적 방법을 통하여 얼마든지 명확하고 엄격하게 분석하고 연구할 수 있다. 어디선가 수리논리학은 명제의 form과 content의 상호작용interplay을 연구하는 학문이라고 하는 것을 보았는데 나는 여기에 크게 동의한다.

Per Martin-Löf의 직관주의 타입이론intuitionistic type theory은 이와 반대 되는 입장을 취한다.[6] 그의 저서 [1]의 일부를 발췌하여 아래에 보였다.

> In standard textbook presentations of first order logic, we can distinguish three quite separate steps:
>
> (1) inductive definition of terms and formulas,
> (2) definition of axioms and rules of inference,
> (3) semantical interpretation.
>
> Formulas and deductions are given meaning only through semantics, which is usually done following Tarski and assuming set theory.

[5] 오히려 수리논리학을 정식으로 배우지 않은 사람보다 더 큰 혼란을 겪을 수 있다. 여기서 수리논리학을 정식으로 배웠다 함은 괴델의 완전성 정리(불완전성 정리가 아님에 유의)를 이해하고, 그것을 남에게 설명할 수 있는 수준을 의미한다.

[6] 형식과 내용을 분리하면 명료하고 엄격하다는 장점이 있고, 분리하지 않으면 더 풍부한 표현력을 누릴 수 있고 또한 모든 것을 구성construct할 수 있다는 장점이 있다. 수학적으로 훈련 받은 사람이라면 후자의 체계가 전자의 체계보다 이해하기에 더 어렵게 느껴질 수 있다고 본다.

> What we do here is meant to be closer to ordinary mathematical practice. *We will avoid keeping form and meaning (content) apart.* Instead we will at the same time display certain forms of judgement and inference that are used in mathematical proofs and explain them semantically. Thus we make explicit what is usually implicitly taken for granted. ... snip ...

수리논리학에서는 명제를 일반적으로 사람들이 말하는 대로 '참/거짓을 판별할 수 있는 문장'이라고 본다. 명제를 기호로 이루어진 수학적 대상으로 구현한 것이 논리식formula이다. 논리식 중에 자유변수를 가지지 않은 것, 즉 closed formula를 (논리)문장sentence이라고 한다. 프랍은 논리식에 대응하는가 아니면 문장에 대응하는가? 이 질문에 대한 답은 neither이다. 여기에 대해서는 p23에 보충 설명이 있다.

수리논리학에서 논리식의 진위는 모델model, 혹은 해석interpretation이 주어진 다음에만 의미를 가진다. 모델이 결정되지 않은 상태에서는 참truth 대신 항진logical validity이라는 개념을 사용할 수 있다. 항진은 모든 모델에서 참이라는 뜻이다.

Coq에서는 의미론semantics이 따로 없다. 프랍들의 집합에 대한 모델model의 개념을 사용하지 않는다. 프랍의 참, 거짓은 그것의 증명, 즉 inhabitant의 존재에 의하여 결정된다.

아직까지 정리에 대해서 말하지 않았다. 정리는 수학/수리논리학에서는 증명된 명제/논리문장을 뜻하는데 일단은 Coq에서도 이와 같이 생각하기로 하고, 상세한 부분은 뒤로 미룬다. 우선 프랍과 증명이 무엇인지를 명확하게 이해하는 것이 중요하며 예를 통해서 이를 이해하기로 한다.

지금까지 우리가 다루었던 대상들의 타입은 모두 Set 타입이다. 예를 들면 monday : day이고 day : Set이다. 이제 프랍의 첫 예를 보자.

```
1 Check 0 = 0. (* 0 = 0 : Prop *)
2 Check 0 = 1. (* 0 = 1 : Prop *)
3 Check forall n: nat, n = n. (* forall n : nat, n = n : Prop *)
4
5 Check n = n. (* Error, n is undefined *)
6
7 Check Prop. (* Prop : Type *)
```

라인 1–3에서 3개의 표현 0 = 0, 0 = 1, forall n: nat, n = n이 각각 프랍임을 Check 명령을 통하여 확인하였다. 라인 5의 n = n은 현재의 컨텍스트에서 ill-typed expression이다.

라인 7을 보면 Prop은 Set와 마찬가지로 Type의 원소인 동시에 부분집합임을 알 수 있다.

라인 1과 라인 3의 프랍에 대한 증명은 곧 보게 될 것이며, 라인 2의 증명은 존재하지 않는다.

```
1 Theorem my0 : 0 = 0.
2 Proof.
3   reflexivity.
4   Show Proof. (* eq_refl *)
5 Qed.
```

캐럿이 라인 1 혹은 라인 2의 마침표period 뒤에 있을 때 Coq Goals 화면은 다음과 같다.

```
Goal 1
(1 / 1) ----------------------------
0 = 0
```

증명해야 할 고울이 하나 뿐이다. 나중에 다른 증명에서 고울이 여럿 있는 경우도 보게 될 것이다. 수평선 (1 / 1) ---------의 윗부분은 원래 컨텍스트(*context*)가 들어가도록 되어 있는데 이 증명은 극도로 단순하므로 현재 비어 있다. 수평선의 아랫부분에는 증명해야 할 프랍인 0 = 0가 있다. 이 프랍을 고울이라고도 하는데, 고울 화면 전체와 구별하기 위하여 '고울 프랍'이라고 부르는 것도 좋겠다.

증명(증명 스크립트)은 Proof.로 시작하고 Qed.로 맺는다. 이 예에서는 증명이 실질적으로 라인 3 한 줄만으로 이루어져 있다. 캐럿을 라인 3의 마침표 뒤에 두면 There are no more subgoals가 나타나며 증명이 완료된다. 라인 3의 reflexivity는 등식의 좌변과 우변의 값이 같을 때 증명을 마쳐 주는 증명 책략proof tactic이다. 나중에 더 설명할 예정이다. 라인 4는 이 증명이 Coq에서 어떻게 저장되어 있는지를 보여주기 위한 것이며 통상적으로는 생략하지만 여기서는 증명에 대해서 상세히 설명하기 위하여 넣어 둔 것이다.

증명에는 증명 스크립트(*proof script*)와 증거항(*proof term, or proof object, or evidence term*)이 있다. Proof.에서 시작하여 Qed.로 끝나는 텍스트가 증명 스크립트이고, 라인 4에 있는 Show Proof의 결과로 보이는 것이 증거항이다.

통상 증명 스크립트는 사람이 손으로 작성하며 증거항은 Show Proof를 통하여 얻는다. 그런데 증거항 eq_refl은 도데체 무엇인가? 그리고 정리 my0의 타입은 무엇인가? 이 질문에 대한 답을 얻기 위하여 Check를 사용하여 보자. 먼저 my0부터.

```
1 Check my0. (* : 0 = 0 *)
2 Print my0. (* my0 = eq_refl : 0 = 0 *)
```

라인 1을 보면 정리의 타입은 그것이 증명하는 프랍임을 알 수 있다. 그리고 라인 2를 보면 정리는 곧 그것의 증거항이며 이 증거항의 타입은 증명 대상 프랍으로 되어 있다. 원래 Print 명령은 그 대상을 정의하는 표현식을 보여주게 되어 있으므로, 이것은 정리가 증거항으로 정의되었음을 말하고 있다. 조금 이해하기 힘들 수 있는데 다음의 정의를 보라.

```
3 Definition my1 : 0 = 0 := eq_refl.
4 Check my1. (* my1 : 0 = 0 *)
5 Print my1. (* my1 = eq_refl : 0 = 0 *)
```

라인 3은 앞서 보았던 Definition a : day := monday와 같은 형식을 취하고 있다. 이것이 상수 표현항의 Definition의 형식이며, 이것은 중요한 암기사항이므로 아래에 정확히 보였다.

Definition *name* : *type* := *value*.

그러니까 프랍 0 = 0는 my1의 타입이다. 이 사실은 라인 3 뿐만 아니라 라인 4와 라인 5에서도 확인된다. (라인 5에서는 type과 value의 위치가 라인 3에서의 위치와 뒤바뀌어 있음을 볼 수 있다. 실은 라인 3을 Definition my1 := eq_refl : 0 = 0로 써도 된다.) 다시 강조하여 말하면

프랍은 타입이다. 그리고 이 타입의 inhabitant는 그 프랍의 증거항이다.

그러니까 *proof_term* : *prop*이고 *prop* : Prop이다. 이것은 마치 monday : day이고 day : Set인 것과 같다.

라인 4와 라인 5를 각각 라인 1과 라인 2에 비교해 보면 my1은 my0와 이름만 다를 뿐 동일한 존재임을 알 수 있다. 요약하면, 정리는 Theorem과 증명 스크립트를 써서 정의할 수도 있고 Definition과 증거항을 써서 정의할 수도 있다.

이제 증거항 eq_refl에 대해서 알아볼 차례인데, 이것은 현재까지 공부하여 얻은 지식으로는 충분히 설명하기 힘들므로 아래의 코드를 보고 대략 넘어가기로 하자.

```
6  Check @eq_refl. (* : forall (A : Type) (x : A), x = x *)
7  Print eq_refl.
8  (* Inductive eq (A : Type) (x : A) : A -> Prop := eq_refl : x = x. *)
```

라인 6에서 @를 앞붙인prefix 것은 (나중에 상세히 설명할 것인데) 다형성polymorphism 때문이며, 이것의 내용은 대략 등호equality의 반사성 공리axiom of reflexivity로 생각하면 되겠다. 라인 7의 결과물인 라인 8을 보면 eq_refl은 2항 술어 eq의 Inductive 정의에서 사용하는 생성자임을 알 수 있다.

이 정의는 등호 eq를 정의하고 있는데, 수학/수리논리학에서는 기호를 도입하고 공리를 써서 그 기호를 정의하는 반면에, Coq에서는 기호에 이름을 주고 그 기호를 사용한 명제의 성립조건(의 증거항)을 생성자를 써서 construct 하며, 이것으로써 기호가 갖추어야 할 성질을 일일히 증명한다는 차이가 있다. Constructive logic에서는 공리를 사용하지 않고 등호, 결합자logical connective, True, False 등 거의 모든 것을 구성하고 그것의 성질을 증명한다. 이 부분은 SF의 Volume 1에서 가장 중요하고 깊은 내용이며 앞으로 여러 예를 보면서 계속 설명하게 될 것이다.

흔히 정리에는 이름이 있고 프랍에는 이름이 없는 것으로 잘못 알고 있는 경우가 있는데, 이제 프랍에 이름을 주어 사용하는 예를 보이겠다. 정리에는 반드시 이름을 주어야 하며 프랍에는 이름을 줄 수도, 주지 않을 수도 있다.

```
1  Definition my2_prop : Prop := 0 = 0.
2
3  Theorem my2 : my2_prop.
4  Proof.
5    unfold my2_prop. (* can be omitted *)
6    apply eq_refl. (* same as reflexivity. *)
7    Show Proof. (* eq_refl : my2_prop *)
8  Qed.
9
10 Check my2. (* my2 : my2_prop *)
```

라인 5의 unfold는 프랍의 이름 my2_prop을 '풀어서' 프랍의 표현식 전체로 바꿔준다. 라인 6의 apply는 증명 스크립트에서 특히 많이 사용되는 책략tactic중 하나이다. 다양한 상황에서 사용되므로 나중에 상세히 설명하겠다. my1 때와 비교하여 어떤 차이가 있는지 살펴 보기 바란다.

이제 0 = 0보다 조금 더 복잡한 프랍 forall n: nat, n = n을 증명해 보자.

```
1  Definition n_eq_n_prop : Prop := forall n : nat, n = n.
2
3  Theorem n_eq_n_thrm : n_eq_n_prop.
4  Proof.
5    unfold n_eq_n_prop.
6    intro n. (* `intros` instead of `intro` works too *)
7    Check n = n. (* n = n : Prop *)
8    apply eq_refl. (* or reflexivity. *)
9    Show Proof. (fun n : nat => eq_refl) : n_eq_n_prop
10 Qed.
```

```
11
12 Definition n_eq_n_thrm' : forall n : nat, n = n := fun n => eq_refl.
```

라인 5를 실행한 후에 Coq Goals 화면을 보면 다음과 같다.

```
1 (1 / 1) ---------------------------
2 forall n : nat, n = n
```

라인 5에는 apply에 필적할 만큼 자주 쓰이는 책략인 intro가 나타난다. 이것의 사용처는 다양한데, 이 경우에는 고울 프랍의 전칭한정 변수를 컨텍스트로 도입intro해 주는 역할을 한다. intro에 의하여 화면이 다음과 같이 바뀐다.

```
3   n : nat
4   (1 / 1) ---------------------------
5   n = n
```

이제 고울 프랍이 n = n이므로 지금까지 많이 사용했던 apply eq_refl로써 증명을 마칠 수 있다.

라인 7은 n = n이 현재의 컨텍스트에서 프랍임을 확인해 주는데 이것은 p11에서 n = n이 ill-typed expression이었던 것과 비교된다.

참고로 intro와 유사한 intros라는 책략도 있는데 이것은 여러 개의 *hypothetical judgement*(a:*type* 혹은 H:*prop* 형태)를 컨텍스트에 도입할 때 사용된다. 1개만 도입할 때는 intro를 쓰도록 되어 있지만 intros를 쓰는 것도 허용된다. Judgement는 직관주의 타입이론에서 프랍과 더불어 가장 중심이 되는 개념이며 수리논리학도에게는 생소할 수 있다. 이에 대해서는 나중에, Coq에 대하여 충분히 경험을 쌓은 후에 더 설명할 것이다.

이러한 증명 기법은 수리논리학에서 사용하는 *fresh variable*에 대한 *generalization*이라고 보면 될 것이다. 직관적으로 보면 $\forall n, n = n$을 증명하기 위하여 n이 주어졌다고 하자. n은 앞에서 언급된 적이 없는 fresh variable이므로, 즉 n은 우리가 아무런 조건도 요구하지 않는 임의의 자연수이므로, 이 특정한 n에 대해서 어떤 논리식이 성립한다면 모든 n에 대해서도 그 논리식이 성립할 것이라는 논리이다.

———— ∘ ————

이상은 SF에는 없는 내용을 저자가 보충한 것이다. 이제부터 다시 SF의 내용을 따라가기로 한다.

간단한 프랍의 예를 다음에 보였다. 둘 다 등식이다. 당분간은 프랍에 사용하는 술어predicate는 등호밖에 없을 것이다.

```
1 Check (next_weekday friday) = monday. (* .. : Prop *)
2 Check (next_weekday monday) = saturday. (* .. : Prop *)
```

라인 1의 명제, 즉 프랍proposition은 참이고 라인 2의 프랍는 거짓이다. 다만 두 경우 모두 타입은 Prop이다. 흔히 Prop 타입의 원소는 참이나 거짓이라는 두 가지 값을 가질 수 있다고 하는데 이 말은 엄격히 말하자면 틀리다. Coq에서 어떤 표현의 값이란 Compute의 결과로 얻어지는 것이다. 다음의 계산을 보면 프랍 표현의 값은, 그 표현을 간략하게 환원*simplify*한 표현이며, 참이나 거짓이 아님을 알 수 있다.

```
1 Compute (next_weekday friday) = monday.
2 (* ==> = monday = monday: Prop *)
3
4 Compute (next_weekday monday) = saturday.
5 (* ==> = tuedsay = saturday: Prop *)
```

Coq에서는 정리를 정의할 때 사용하는 키워드가 유일하지 않다. Theorem, Lemma, Fact, Remark, Corollary, Proposition, Property, Example 등은 모두 theorem을 정의하는 데 사용될 수 있다.

간단한 하나의 정리와 그 증명을 보자.

```
1 Example test_next_weekday:
2   (next_weekday (next_weekday saturday)) = tuesday.
3 Proof. simpl. reflexivity. Qed.
```

이 정리에 나타나는 모든 마침표를 전후하여 캐럿이 놓였을 때 Coq Goals 화면이 어떻게 변화하는지 관찰하여라.

simpl은 간략화 계산(simplification)을 하는 책략tactic인데 이것이 정확히 어떤 의미를 가지는지는 차차 설명할 것이다.

reflexivity는 등식equational proposition의 양변이 동일할 때 Goal을 제거하는 책략이다. 양변이 동일하다는 것은, 표현이 동일한 경우에는 당연히 성립하는 것이고, 표현은 다르지만 값이 동일해도 성립하는데 이때 어느 정도의 간략화 계산이 이루어진다. simpl보다 더 잘 계산해준다. reflexivity는 simpl을 포함하므로 reflexivity 직전에 나오는 모든 simpl은 생략해도 된다. 증명에서 simpl의 용도는 증명의 확인이 아니라 증명 과정의 이해인 것으로 보면 된다.

Booleans

Coq에서 참과 거짓을 다루는 데는 2가지가 있다고 볼 수 있다. 첫 번째 방법은 부울값과 등식을 이용하는 것이다.

부울리언(*Boolean*) 타입은 다음과 같이 정의된다.

```
Inductive bool : Type :=
  | true
  | false.
```

그런데 실은 이 타입은 이미 Coq의 표준 라이브러리에 포함되어 있다.

사용자가 정의하는 타입은 ① 그것을 정의하기 전에는 사용할 수 없고, ② 동일한 이름의 타입을 다시 정의하면 에러가 난다. 하지만 bool과 같이 Coq의 표준 라이브러리에 들어 있는 타입은 ① 그것을 정의하지 않고도 사용할 수 있고, ② 동일한 이름의 타입을 다시 정의해도 에러가 나지 않는다.

이 사실 ①, ②는 타입뿐만 아니라 함수에도 적용된다. 예를 들어 negb라는 함수는 다음과 같이 정의된다.

```
Definition negb (b:bool) : bool :=
  match b with
  | true => false
  | false => true
  end.
```

이 함수도 표준 라이브러리에 들어 있으므로 실은 Basics.v에서 정의하지 않고도 사용할 수 있었다. 아무튼 계속 부울리언 함수를 2개 더 정의하자. 이들은 2항binary 함수이다.

```
Definition andb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => b2
  | false => false
  end.

Definition orb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => true
  | false => b2
  end.
```

이 함수들이 제대로 작동하는지는 다음과 같은 *unit test*를 통해 확인할 수 있다.

```
Example test_orb1:  (orb true  false) = true.
Proof. simpl. reflexivity. Qed.
(* 3 more such theorems *)
Example test_and1:  (andb true false) = true.
Proof. simpl. reflexivity. Qed.
(* 3 more such theorems *)
```

그런데 이런 부울함수들은 (다른 많은 함수들도 그렇지만) 다음과 같이 if-then-else 구문을 사용하여 정의할 수도 있다.

```
Definition negb' (b:bool) : bool :=
  if b then false else true.

Definition andb' (b1:bool) (b2:bool) : bool :=
  if b1 then b2 else false.

Definition orb' (b1:bool) (b2:bool) : bool :=
  if b1 then true else b2.
```

bool은 표준 라이브러리에 들어있기는 하지만 붙박이built-in는 아니다. true, false는 우리가 임의로 선택하여 bool: Type의 정의에 사용한 것이며 특별한 의미를 가지지 않는다. if-then-else 구문은 bool뿐만이 아니라 다른 어떤 타입에 대해서도, 그것의 원소가 딱 2개이기만 하면 작동한다. 첫 번째 원소를 true, 두 번째 원소를 false로 취급하면 된다.

```
Inductive mybool : Type :=
  | a_my
  | b_my.

Definition negb_my (b:mybool) : mybool :=
  if b then b_my
  else a_my.

Definition andb_my (b1:mybool) (b2:mybool) : mybool :=
  if b1 then b2
  else b_my.
```

```
Definition orb_my (b1:mybool) (b2:mybool) : mybool :=
  if b1 then a_my
  else b2.
```

이렇게 if-then-else 구문을 사용한 정의가 제대로 작동함을 unit test를 써서 확인해 보라. negb_my부터 orb_my까지 모두 3개의 함수가 있는데 이들 각각에 대해서 2개 혹은 4개의 unit test를 작성하면 된다.

simpl에 대하여

simpl이 작동하지 않는 예를 들어 보겠다. nandb 함수를 다음과 같이 3가지 서로 다른 방법으로 정의해 보았다.

```
Definition nandb1 (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => negb b2
  | false => true
  end.

Definition nandb2 (b1:bool) (b2:bool) : bool :=
  if b1 then (negb b2) else true.

Definition nandb3 (b1:bool) (b2:bool) : bool :=
  negb (andb b1 b2).
```

이 정의들에 대하여 각각 unit test를 해 보면, nandb1과 nandb2에서는 simpl이 작동하지만 nandb3에서는 작동하지 않는다. 하지만 nandb3에서도 reflexivity는 잘 작동한다.

정리하면, simpl은 match .. end, 또는 if-then-else를 써서 정의한 함수에는 작동하지만 함수의 합성composition을 써서 정의한 함수에는 듣지 않는다. reflexivity는 simpl보다 더 강력하게 간략화 계산을 한다. reflexivity를 쓰기 전에 simpl을 사용하지 않아도 증명에는 지장이 없다. 다만 simpl을 쓰면 증명 과정을 이해하는 데 도움이 된다.

Coq 소스 파일에서 작업 중에 증명을 마무리 하지 못했지만 진도를 더 나가고 싶을 때는 일단 Admitted로 가마무리(가짜 마무리, 임시 마무리) 하고 진행할 수 있다. Admitted는 나중에 증명을 채워 넣은 다음에 삭제하면 된다.

Admitted와 비슷한 것으로 Abort가 있다. 이것도 일단 증명을 가마무리 하고 진행하는 데 사용된다. 다만 정리의 증명을 Admitted 해 놓으면 그 정리를 뒤에 다른 정리의 증명에 사용할 수 있지만 Abort 해 놓은 정리는 다음에 사용할 수 없다는 차이가 있다.

Infix Notation

논리합disjunction과 논리곱conjunction 연산은 가운데 쓰기infix 표기법으로 쓰는 경우가 많다.

```
Notation "x && y" := (andb x y).
Notation "x || y" := (orb x y).
```

New Types from Old

이 섹션의 제목은 실은 non-nullary constructor가 더 적당할 수 있다고 본다. rgb는 평범한 enumeration type이다. 그리고 color는 rgb 타입을 인수로 하는 1항unary 생성자 primary를 사용한다. color는 상수 생성자가 아닌 함수 생성자, 즉 *constructor with parameter*로서는 이 책에서 처음 등장한다.

```
Inductive rgb : Type :=
  | red
  | green
  | blue.

Inductive color : Type :=
  | black
  | white
  | primary (p : rgb).
```

color 타입의 원소의 개수는 5개임을 확인해 보라. 다음에 정의하는 monochrome 함수는 color 타입의 원소를 인수로 받아서 bool 타입의 값을 리턴한다.

```
1 Definition monochrome (c : color) : bool :=
2   match c with
3   | black => true
4   | white => true
5   | primary _ => false (* p in place of _ is ok *)
6   end.
```

라인 5가 특이하다. 1항 생성자와 매치를 하는데, 리턴값은 그 생성자의 인수의 값과 무관하게 일정하다. 이럴 때는 _를 사용하여 인수를 무시하는 것이 좋다. (인수를 사용해도 상관은 없다. 다만 이것은 혼동을 주기 때문에 좋지 않다.)

Module

모듈(*module*)은 Coq에서 코드를 구조화하는 데 사용된다. 모듈은 Module 키워드를 사용하여 정의한다.

```
Module Playground.
  Definition foo: rgb := red.
End Playground.
```

모듈 내에서 정의된 타입, 함수 등은 모듈 외부에서 사용할 수는 있지만 이때는 그냥 foo를 써서 사용할 수 없고 Playground.foo와 같은 표현을 써서 사용해야 한다.

Numbers

이제야 처음으로 무한개의 원소를 가지는 타입을 소개하게 되었다. 다음의 정의가 그것이다. 이것은 일종의 1진법이다.

```
Inductive nat : Type :=
  | O
  | S (n : nat). (* _ instead of n is ok *)
```

1진법은 원래 1개의 기호만 사용하여야 하는 것인데, 이렇게 할 경우에는 0을 나타내려면 기호를 하나도 사용하지 않아야 한다는 문제가 발생하므로 일종의 end-marker로 O를 사용한다. 숫자 0이 아니라 대문자 O를 사용하는 것에 주목하라.

O는 0항$_{\text{nullary}}$ 생성자, S는 1항$_{\text{unary}}$ 생성자로 사용되었다.

지금까지의 Inductive 정의는 실은 재귀적이지 않았으며 따라서 유한집합만 정의할 수 있었다. nat는 처음으로 Inductive 정의의 위력을 사용하여 무한집합을 정의한 예이다. 타입 t의 정의가 재귀적$_{\text{recursive}}$이라 함은 생성자 중에 t 타입의 인수를 받는 것이 있다는 것을 의미한다. nat의 경우에는 S 생성자가 nat 타입의 인수를 받는다.

nat 타입은 표준 라이브러리에 포함되어 있지만, 앞서 말했듯이 이렇게 새로 정의해도 된다.

무한집합을 도메인(*domain!of discourse*)으로 가지는 타입에 대한 함수를 정의할 때, 아주 간단한 함수라면 재귀를 쓰지 않고도 정의가 가능하다. 바로전 원소$_{\text{predecessor}}$ 함수 pred를 다음과 같이 정의한다.

```
Definition pred (n : nat) : nat :=
  match n with
  | O => O
  | S n' => n'
  end.
```

크기가 좀 되는 자연수를 나타낼 때 S를 여러 개 사용하는 것은 불편하므로 Coq은 통상 사용하는 표기법 $n \stackrel{\text{def}}{\equiv} \overbrace{\texttt{S}\cdots\texttt{S}}^{n\text{개}}\ \texttt{O}$를 제공한다. 이 계산은 내부적으로$_{\text{internally}}$ 이루어지며 사용자가 개입할 여지가 없다.

다음은 even과 odd의 정의이다. 이들은 pred 경우와 달리 재귀를 사용한다. 함수의 재귀적 정의를 위한 Coq의 키워드는 Fixpoint이다. 지금까지 함수의 정의에 사용되었던 Definition은 재귀적 함수의 정의에는 쓸 수 없다.

```
Fixpoint even (n:nat) : bool :=
  match n with
  | O        => true
  | S O      => false
  | S (S n') => even n'
  end.

Fixpoint odd (n:nat) : bool :=
  match n with
  | O        => false
  | S O      => true
  | S (S n') => odd n'
  end.
```

만일 odd를 다음과 같이 정의하면

```
Definition odd (n:nat) : bool :=
  negb (even n).
```

함수의 사용에는 문제가 없지만, odd가 나타나는 Proof에서 simpl이 작동하지 않는다. 물론 이 경우에도 reflexivity는 제대로 작동한다.

덧셈plus, 곱셈mult, 뺄셈minus, 거듭제곱exp 등의 2항함수도 재귀를 사용하여 정의할 수 있다. 이들 중 plus와 minus의 정의만 아래에 보였다.

minus의 정의에서 배울 것이 2가지 있다.

① 2개의 인수가 연이어 나올 때 이들이 동일한 타입을 가진다면, 이 두 인수들을 하나로 묶어서 쓸 수 있다.

② 2개의 인수를 동시에 match시킬 수 있다.

```
Fixpoint plus (n: nat) (m: nat): nat :=
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.

Fixpoint minus (n m: nat): nat :=
  match n, m with
  | O   , _    => O
  | S _ , O    => n
  | S n', S m' => minus n' m'
  end.
```

plus의 타입을 확인해 보자.

```
Check plus.
(* ===> nat -> nat -> nat *)
```

plus의 타입이 nat * nat -> nat가 되어야 할 것 같은데 Check의 결과는 좀 다르다. 이것은 Coq가 다변수 함수를 *Currying*을 통하여 처리하고 있기 때문이다. Currying을 알지 못하는 독자라면 인터넷 검색을 통하여 이 개념을 쉽게 이해할 수 있을 것이다.

Coq에서는 함수의 인수에 괄호를 쓰지 않는 것이 원칙이라고 했는데, 이를 두고 괄호를 써도 된다고 이해하면 곤란하다. 예를 들어 plus 2 3은 plus (2 3), 혹은 plus(2, 3) 등으로 쓰면 안 된다. plus 2 3은 (plus 2) 3에서 function application의 left associativity에 의하여 괄호를 생략한 것이다.

사칙연산에 대한 가운데 쓰기는 다음과 같이 정의한다.

```
Notation "x + y" := (plus x y)
                      (at level 50, left associativity)
                      : nat_scope.
Notation "x - y" := (minus x y)
                      (at level 50, left associativity)
                      : nat_scope.
Notation "x * y" := (mult x y)
                      (at level 40, left associativity)
                      : nat_scope.
```

이 정의에서

① [at level 50], [left associativity], [nat_scope]는 모두 생략가능 하다.

② [at level n]는 우선순위를 말한다. n은 0 이상 100 이하의 정수이며 작을수록 상대적 우선순위가 높다.

③ [left associativity]는 왼쪽결합성을 말한다.

④ [nat_scope]는 이 표기법이 nat 타입에 적용된다는 것을 말한다. 생략해도 되지만 쓰는 것이 좋다.

Boolean functions on nat

자연수에 대한 중요한 부울값 함수들을 몇 개 정의해 보겠다.

```
Fixpoint eqb (n m : nat) : bool :=
  match n with
  | O => match m with
    | O => true
    | S m' => false
    end
  | S n' => match m with
    | O => false
    | S m' => eqb n' m'
    end
  end.

Fixpoint leb (n m : nat) : bool :=
  match n with
  | O => true
  | S n' =>
      match m with
      | O => false
      | S m' => leb n' m'
      end
  end.

Fixpoint ltb (n m : nat) : bool :=
  match n with
  | O => match m with
    | O => false
    | S m' => true
    end
  | S n' => match m with
    | O => false
    | S m' => ltb n' m'
    end
  end.

(* Definition ltb (n m : nat) : bool :=
     andb (n <=? m) (negb (n =? m)). *)

Notation "x =? y" := (eqb x y) (at level 70) : nat_scope.
Notation "x <=? y" := (leb x y) (at level 70) : nat_scope.
Notation "x <? y" := (ltb x y) (at level 70) : nat_scope.
```

위의 정의는 어려울 것 없다. 하지만 '=?'와 '='의 차이점을 정확하게 말하기는 쉽지 않을 수 있다. 전자는 nat 타입 표현항$_{\text{term}}$들에 대한 부울값 표현이고, 후자는 Prop이다.

```
Check  (2 =? 2). (* 2 =? 2: bool *)
Compute (2 =? 2). (* = true: bool *)
Check 2 = 2. (* 2 = 2: Prop *)
Compute 2 = 2. (* = 2 = 2: Prop *)
```

<=?와 <?에 대응하는 <=와 <도 있다. 이와 관련하여 나중에 더 심도있게 논하기로 한다.

2.2 증명 기법

증명에 사용되는 명령, 혹은 키워드들을 책략(*tactics*)이라고 한다. 지금까지 우리가 사용했던 책략은 simpl, reflexivity, intros, intro, apply 등이 있다. 이 절에서 몇 개의 책략을 더 공부하기로 한다.

증명 관련 용어들을 복습하자. Coq Goals 윈도의 윗부분에는 Proof 영역이, 아랫부분에는 Messages 영역이 있다. Proof 영역은 1개 이상의 고울들로 이루어지고, 각 고울은 수평선 위의 컨텍스트와 수평선 아래의 고울 프랍으로 이루어진다.

intros

intros 책략은 고울 프랍이 전칭문(*universal proposition*)이거나 조건문(*conditional*)일 때 사용할 수 있다.다음의 코드를 Basics.v에 넣고 실행해 보자.

```
1 Fact theorem0 : forall n m: nat, n = m -> n = m.
2 Proof.
3   intros n m.
4   intro H.
5   exact H. Qed.
```

라인 1은 다음과 같이 두어도 원래와 똑같다.

```
Fact theorem0 : forall n: nat, forall m: nat, n = m -> n = m.
```

라인 3은 intro n. intro m.으로 두어도 원래와 것과 똑같은 효과를 가진다. 라인 3과 4를 합쳐서 intros n m H.로 두는 것도 마찬가지다. intro와 intros는 기본적으로 동일한 책략이지만, 후자는 여러 개의 인수를 동시에 처리할 수 있다는 점이 다르다.

라인 3은 intros를 고울 프랍이 전칭문일 때 실행한 것이며, 이때 Coq Goals 화면은 다음과 같게 된다.

```
n, m : nat
(1 / 1) ------------------------
n = m -> n = m
```

전칭문이던 고울 프랍은 한정사가 사라져 조건문이 되었고, 한정사에서 사용되었던 묶인 변수는 컨텍스트으로 이동하여 타입 선언type declaration 되었다.

라인 4는 intro를 고울 프랍이 조건문일 때 실행한 것인데, 이런 경우 조건문의 전건antecedent가 컨텍스트로 이동하여 가설hypothesis이 된다. 그리고 고울 프랍은 원래 고울 프랍의 후건

succedent, consequent으로 대체된다.[7] 이때 `Coq` `Goals` 화면은 다음과 같게 된다.

```
n, m : nat
H : n = m
(1 / 1) -----------------------
n = m
```

라인 5의 `exact H`는 가설 `H`와 고울 프랍이 정확히 일치할 때 사용하여 증명을 마무리하는 책략이다.

이상으로써 증명과정의 설명은 되었지만 보충할 것이 많다. 다음은 아주 기본적이고 중요한 사실들이며, 좀 길지만 반드시 정확히 이해해 두어야 할 내용이다.

(1) 정리에서 증명하고자 하는 프랍에는 자유변수free variable가 나타나지 않아야 한다. 하지만 증명이 시작되고 나면 컨텍스트에 타입 선언된 변수들은 고울 프랍에 자유변수로 나타날 수 있다.

앞서 프랍은 논리식이라고 할 수도 없고 문장이라고 할 수도 없다고 하였는데, 이제 그 이유를 알 수 있을 것이다 — 증명하려는 프랍은 처음에는 closed formula, 즉 문장이어야 한다. 일단 증명이 시작되어 컨텍스트에 타입 선언된 변수들이 나타나면, 이 변수들은 고울 프랍이나 가설 프랍에 자유변수로 나타날 수 있게 된다.

(2) 한정사(*determiner*) `forall n: nat,`의 범위(*scope*)는 콤마 기호 뒤에 나오는 논리식의 끝까지이다.

`forall n: nat`는 한정사이고 `forall`은 한정기호(*quantifier*)이다. 한정기호에는 전칭한정기호universal quantifier(`forall`, ∀)와 존재한정기호existential quantifier(`exists`, ∃)가 있다.

전칭한정사, 존재한정사, 전칭한정문, 존재문, 존재한정문 등의 용어의 의미는 잘 알고 있을 것으로 알고 따로 설명하지 않겠다.

한정사의 바로 전에 왼편 괄호가 있는 경우에는 그 괄호에 대응하는 오른편 괄호까지가 그 한정사의 범위이다.[8]

(3) `intros` 책략이 실행되면 증명의 고울의 컨텍스트(*context*)에 *hypothetical judgement*가 추가(또는 도입introduce)된다.

컨텍스트에는 0개 이상의 라인이 있으며, 각 라인은 `H:` *prop* 형태의 가설hypothesis, 혹은 `x:` *type* 형태의 타입 선언type declaration이다. 이들을 가설 저지먼트hypothetical judgement라고 한다.

(4) `n: nat`과 같은 타입 선언은 타입을 명시했다는 것 이상의 의미가 있다. 이것으로써 `n`은 임의의 값을 배정할 수 없는 자유변수가 되었으며, 그 값이 무엇인지는 알 수 없지만 특정한 값으로 고정되었다는, 즉 특정(*particularize*), 또는 *instantiate* 되었다는 의미이다. (자유변수는 값을 배정할 자유가 없고, 묶인변수는 값이 묶여있지 않아 자유롭게 배정할 수 있으므로 이 용어들은 좀 혼동스러운 면이 있는 것이 사실이다.)

---

[7] 흔히 조건문의 전건antecedent을 가설hypothesis, 후건consequent, succedent을 결론conclusion이라고 하기도 하는데, 이것은 hypothesis와 conclusion의 원래 용법에서 벗어난다. `Coq`에서 hypothesis는 컨텍스트 내에 존재하여 고울을 증명하는 데 사용되는 프랍이고, conclusion은 곧 goal proposition이다.

[8] 이 경우에는 이 한정문은 부분프랍sub-proposition이며 원래 증명의 대상이 아닐 것이다.

증명의 두 번째 예로서 다음을 보라.

```
1 Fact theorem1 : forall n m: nat, n = m -> n = m.
2 Proof.
3   intros.
4   exact H. Qed.
```

theorem1은 theorem0와 거의 같으나 intros에 인수를 사용하지 않았다. 그러나 증명과정에서 Coq Goals 화면은 theorem0에서와 완전히 동일할 것이다. Coq가 알아서 변수와 가설의 이름을 지어주었기 때문이다.

이번에는 라인 3을 intros k l H1.으로 바꿔서 실행해 보라. 증명과정에서의 Coq Goals 화면은 내용적으로는 이전과 동일하며 변수와 가설의 이름만 다를 것이다. 증명스크립트에서는 당연히 라인 4의 exact H.를 exact H1.으로 바꿔야 한다.

전칭문에 intros를 적용할 때 변수 이름을 원래의 묶인 변수와 다른 것으로 지정하는 것은 많은 경우 필요가 없으나 때로는 필요한 경우도 있다. 예를 들어 고울이 forall n: nat, .. 형태인데, 증명 도중에 n이라는 자유변수가 컨택스트 내에 이미 들어있다면, intros를 실행 시 변수 이름의 충돌을 피하기 위하여 n', m 등의 새로운fresh 이름을 intros의 인수로 사용해야 할 것이다.

rewrite

새로운 책략 rewrite의 사용법을 다음의 예를 통하여 알아보자.

```
1 Theorem plus_id_example : forall n m:nat,
2   n = m ->
3   n + n = m + m.
4 Proof.
5   intros n m.
6   intros H.
7   rewrite -> H.
8   reflexivity.  Qed.
```

다음은 라인 6까지 실행한 직후의 Coq Goals 화면이다.

```
n, m : nat
H : n = m
______________________________
n + n = m + m
```

이 상태에서 라인 7을 실행하면 고울 프랍에서 n의 모든 나타남이 m으로 바뀌어 다음과 같이 된다.

```
m + m = m + m
```

라인 7의 의미는 H: n = m을 이용하여 고울 프랍를 다시 쓰라rewrite는 것이다. 방향을 ->로 지정했으니 n을 m으로 rewrite한다. 만일 방향을 <-로 지정했더라면 m을 n으로 rewrite 했을 것이다. 방향을 지정하지 않고 그냥 rewrite H로 쓰면 디폴트 방향인 ->가 적용된다.

이제 고울 프랍의 양변은 동일하므로 reflexivity를 실행하여 증명을 마칠 수 있다.

intros를 요약하면 다음과 같다.

(1) 고울이 전칭한정문 $\forall x : X, P(x)$일 때 intros를 실행하면 $x : X$가 컨텍스트로 올라가고 고울은 $P(x)$로 바뀐다. intros x'도 가능하며 이때는 $x' : X$가 컨텍스트로 올라가고 고울은 $P(x')$이 된다.

(2) 고울이 조건문 $P \to Q$일 때 intros 하면 P가 컨텍스트로 올라가고 고울은 Q로 바뀐다. intros H와 같이 가설의 이름을 지정하는 것도 가능하다. (실은 여기서 H는 가설이 아니라 이 가설의 증거항이다. 나중에 더 설명할 것이다.)

(3) 고울이 $P \to Q \to R$일 때 intros H1 H2를 실행하면 컨텍스트에 두 개의 가설 H1: P와 H2: Q가 추가되고 고울은 R로 바뀐다. 이러한 예는 나중에 보게 될 것이다.

rewrite 책략은 가설뿐만 아니라 기존에 증명해 놓은 정리도 사용할 수 있다. (책략의 디폴트 타겟$_{\text{target}}$은 고울이다. 우리는 rewrite 책략을 사용할 때 가설이나 정리를 '인수'로 하여 고울에 적용할 수 있다.)

다음의 정리를 보라.

```
1 Theorem mult_n_0_m_0 : forall p q : nat,
2   (p * 0) + (q * 0) = 0.
3 Proof.
4   intros p q.
5   rewrite <- mult_n_O. (* 0 + q * 0 = 0 *)
6   rewrite <- mult_n_O. (* 0 = 0 *)
7   simpl. reflexivity. Qed.
```

라인 5와 라인 6에서 rewrite 책략을 사용하고 있는데 이때 mult_n_O가 사용되었다.[9]

```
Check mult_n_O. (* mult_n_O : forall n : nat, 0 = n * 0 *)
```

라인 5는 p * 0을 0으로 바꾸어 주는 것이므로 정리 mult_n_O에서 n := p로 치환하여 사용하였다. 라인 6은 n := q로 치환하여 사용한다. 이렇듯 rewrite 책략은 전칭문 정리를 인수로 받았을 때 전칭한정변수(이 경우에는 n)를 상황에 맞도록 적당한 값으로 치환해 준다.

요령 2.1 정리의 이름을 알되 내용을 정확히 모를 때 Check를 사용할 수 있다.

```
Check mult_n_O. (*forall n : nat, 0 = n * 0 *)
```

왜냐하면 Check는 원래 인수의 타입을 보여주도록 되어 있는데, 정리의 타입은 그것이 증명하는 프랍이기 때문이다. 프랍의 원소는 증거항, 혹은 이와 동일시 되는 그 정리이다. 하나의 정리에 서로 다른 증명들이 있을 수 있으며 이 증명들의 타입이 바로 그것이 증명하는 프랍이다. ⊣

Proof by Case Analysis

Coq로 $\forall n, n+1 \neq 0$를 증명해 보자. 이것은 단순히 intros, simpl, reflexivity로는 되지 않는다. 자연수 n은 0, 또는 S n'의 형태를 가지므로 각각의 경우에 대해서 증명을 구성하도록 한다. 이렇게 경우로 나누어 증명할 때 사용하는 것이 destruct 책략이다.

[9] 이 정리는 Coq의 표준 라이브러리에 포함되어 있으므로 우리는 이를 어떤 증명에서든 자유로이 사용할 수 있다.

```
1 Theorem plus_1_neq_0 : forall n : nat,
2   (n + 1) =? 0 = false.
3 Proof.
4   intros n.
5   destruct n as [| n'] eqn:E.
6   - (* n = 0 *) simpl. reflexivity.
7   - (* n = S n' *) simpl. reflexivity.   Qed.
```

라인 5에서 증명을 분할destruct한 이후의 Coq Goals 화면은 다음과 같다. 이것은 Goal이 2개 이상인 첫 예이다.

```
8   Goal 1
9   n : nat
10  E : n = 0
11  (1 / 2) -------------------
12  (0 + 1 =? 0) = false
13
14  Goal 2
15  n, n' : nat
16  E : n = S n'
17  (2 / 2) -------------------
18  (S n' + 1 =? 0) = false
```

두 개의 Goal 각각에 대한 증명을 라인 6과 라인 7에서 '-'로 시작하였다. 이러한 기호 '-'를 불릿(*bullet*)이라고 한다. 불릿을 쓰지 않고도 증명을 진행할 수 있으나 불릿을 쓰는 것이 증명의 구조를 파악하는 데 도움을 주므로 되도록이면 쓰는 것이 좋다.

자연수의 덧셈 plus : nat -> nat -> nat의 정의는 다음과 같다.

```
1 Fixpoint plus (n : nat) (m : nat) : nat :=
2   match n with
3   | O => m
4   | S n' => S (plus n' m)
5   end.
```

① n이 0와 매치되는 경우는 Goal 1에서 처리하고 있다. 이 경우에는 n = 0가 성립해야 하므로 이 사실이 컨텍스트의 라인 10에 가설 E : n = 0로 명시되어 있다. n = 0이므로 고울 프랍 (n + 1 =? 0) = false는 (0 + 1 =? 0) = false로 바뀐다. 이것은 simpl에 의하여 true = false로 바뀐다. 이는 두 단계의 계산을 하나의 simpl이 처리한 결과이다. 그 다음은 reflexivity에 의하여 Goal 1의 증명이 끝난다.

② n이 S n'과 매치되는 경우는 Goal 2에서 처리하고 있다. 이 경우에는 n = S n'이 성립해야 하므로 이 사실이 컨텍스트의 라인 15에 가설 E : n = S n'으로 명시되어 있다. n = S n'이므로 고울 프랍 (n + 1 =? 0) = false는 (S n'+ 1 =? 0) = false로 바뀐다. 이것은 simpl에 의하여 (S (n'+ 1) =? 0) = false로 되었다가 다시 false = false로 바뀐다. 그 다음은 reflexivity에 의하여 Goal 2의 증명이 끝난다.

이상으로써 destruct 책략이 작동하는 과정을 아주 세밀하게 기술하였다. 앞으로 destruct와 유사한, 그러나 더 고난도의 새로운 case analysis 책략들을 만나게 될 것이며, 그때마다 이러한 방식의 분석 과정을 확실하게 이해하고, 또 혼자서 스스로 확인해 보는 것이 Coq을 잘 이해하는 방법이다.

아직 라인 5 destruct n as [| n'] eqn:E에 대해서 조금 더 설명이 필요하다. as 이하는 꼭 필요한 것은 아니지만 생략하지 않는 것이 좋다. [| n']은 *intro pattern*이라고 부르는 것이다. 이것은 n이 두 constructor(O와 S) 중 어느 하나에 매치될 때, 즉 분해destruct될 때, 각각의 경우에 사용할 변수의 이름을 지정한다. |의 왼쪽에는 첫 번째 생성자에 대한 변수를 넣고, 오른쪽에는 두 번째 생성자에 대한 변수를 넣는다. Intro pattern을 생략하면 Coq이 임의로 지어낸 변수 이름을 사용하게 된다.

첫 번째 생성자는 상수 생성자 O이므로 매치시켜야 할 변수가 존재하지 않아서 변수 이름을 지정해 줄 필요가 없다. 그래서 |의 왼쪽은 비어 있다.

두 번째 생성자는 1개의 인수를 가지는 S이므로 하나의 변수가 필요하다. 이것을 n'으로 이름을 주어 |의 오른쪽에 넣어 두었다.

생성자가 3개 이상인 경우, 혹은 하나의 생성자가 2개 이상의 인수를 가지는 경우 등도 있는데 이런 경우는 나중에 다루게 된다.

eqn:E는 *annotation*이라고 부르는 것이다. 이것은 매치될 때 성립해야 하는 등식의 이름을 E로 지정한다. Annotation을 생략하면 Coq이 임의로 지은 이름을 사용하는 것이 아니라 아예 등식이 나타나지 않는다.

생성자마다 따로 이름을 줄 필요는 없으며 하나의 이름을 모든 생성자가 공유한다. 이 예에서는 E를 증명에서 사용하지 않았으므로 생략해도 별 문제가 없었다. 하지만 때로는 이것이 꼭 필요한 경우도 있으며 나중에 이런 예를 보게 될 것이다.

n'을 m으로, E를 E1으로 바꾸어 놓고 destruct를 실행해 보면 destruct에서의 intro pattern의 의미를 잘 알 수 있을 것이다.

Intro pattern은 다른 여러 case analysis 책략에도 있다. 물론 그때마다 다른 방식으로 기능한다. 변수와 아울러 가설(귀납가설 등)을 as [...]에 넣는 경우도 있다. 앞으로 새로운 case analysis 책략을 소개할 때 상세하게 설명할 것이다. Case analysis 책략은 destruct 외에 induction, inversion 등이 있는데 이들 중에 destruct가 가장 간단하고 쉬운 편이다. 하나의 책략이라고 해도 적용 대상 변수의 데이터 타입에 따라 디테일이 많이 달라지므로 이러한 책략들을 익히는데 많은 시간을 할애해야 할 것이다. 어찌 어찌 증명에 성공했다고 해서 만족하지 말고 모든 디테일을, 다른 사람에게 자시있게 설명할 수 있을 정도로 이해해야 한다.

destruct 책략은 전칭문에 대하여 intros 바로 뒤에 사용하는 경우가 많다. 이 책략은 Inductive 키워드를 써서 정의된 임의의 타입 변수에 대하여 사용할 수 있다. 예를 들어 bool 타입 변수에도 destruct를 사용할 수 있다. 다음의 예를 보라.

```
Theorem negb_involutive : forall b : bool,
  negb (negb b) = b.
Proof.
  intros b. destruct b eqn:E. (* we don't need as [..|..] *)
  - simpl. reflexivity.
  - simpl. reflexivity.  Qed.
```

이 증명에서는 intro pattern을 사용하지 않았다. 그 이유는 두 생성자가 모두 nullary이기 때문이다.

andb의 결합법칙 같은 것의 증명은 destruct를 여러 번 사용해야만 한다. 이 증명의 구조는 subgoal 내부에 subgoal, 그 내부에 다시 subgoal이 존재하는 형태, 즉 다층구조이므로 불릿을

여러 종류 사용해야 하겠지만, 불릿 대신 이렇게 중괄호 쌍을 사용해도 된다. 일단 중괄호를 사용하면 그 내부의 불릿은 외부에서 이미 사용했던 불릿 기호를 또 사용해도 된다. 중괄호를 쓰지 않는 경우에는 불릿 기호로 -, +, * 등을 사용하면 된다.

```
  Theorem andb3_exchange :
  forall b c d, andb (andb b c) d = andb (andb b d) c.
Proof.
  intros b c d. destruct b eqn:Eb.
  - destruct c eqn:Ec.
    { destruct d eqn:Ed.
      - reflexivity.
      - reflexivity. }
    { destruct d eqn:Ed.
      - reflexivity.
      - reflexivity. }
  - destruct c eqn:Ec.
    { destruct d eqn:Ed.
      - reflexivity.
      - reflexivity. }
    { destruct d eqn:Ed.
      - reflexivity.
      - reflexivity. }
Qed.
```

정말 재미없는 증명이다. 이 증명은 본질적으로 진리표를 (일목요연하지 않은 방법으로) 그리는 것과 같다.

때로는 destruct 키워드를 생략하고도 destruct 책략을 사용할 수 있다. 다음의 예를 보라.

```
Theorem plus_1_neq_0' : forall n : nat,
  (n + 1) =? 0 = false.
Proof.
  intros [|n'].
  - reflexivity.
  - reflexivity.  Qed.
```

상수 생성자만으로 정의된 inductive type에 대해서는 변수지정이 필요 없으므로 다음과 같이 해도 destruct 책략이 사용된다.

```
  Theorem andb_commutative'' :
  forall b c, andb b c = andb c b.
Proof.
  intros [] [].
  - reflexivity.
  - reflexivity.
  - reflexivity.
  - reflexivity.
Qed.
```

나는 개인적으로 이런 단축형을 썩 좋아하지는 않지만, 사람들이 Coq 증명을 할 때 종종 이런 방식으로 destruct를 사용하므로 이를 읽고 이해할 수 있으려면 이런 것도 알아두는 것이 좋을 것이다.

More Exercises

SF는 연습문제의 해답을 공개하지 말아 달라고 말하고 있으므로 여기서는 연습문제의 해답 전체가 아니라 힌트만 제공할 것이다.

이 책은 어려운 문제에 대해서는 모두 충분한 힌트를 제공할 것이다. 그리고 쉬운 문제들에 대해서는 언급하지 않는 경우가 많을 것이다.

```
1 Theorem identity_fn_applied_twice :
2   forall (f : bool -> bool),
3   (forall (x : bool), f x = x) ->
4   forall (b : bool), f (f b) = b.
5 Proof.
6   intros f H b.
7   rewrite -> H.
8   (* ... *)
```

(해설). 이 문제에서 새로운 것은 라인 2에서 보듯이 전칭한정universal quantification을 함수에 대해서 적용했다는 것이다.

라인 6을 더 쉽게 이해하려면 intros f. intros H. intros b.로 나누어 쓰고 하나씩 실행해 보라. 그 다음에 Coq Goals를 보면 다음과 같을 것이다.

```
9  f : bool -> bool
10 H : forall x : bool, f x = x
11 b : bool
12 (1 / 1) ------------------------
13 f (f b) = b
```

라인 7에서 실행한 rewrite -> H를 눈여겨 보자. H는 라인 10에 나타나 있다.

화살표의 방향이 ->이므로 H의 좌변인 f x를 고울 프랍 f (f b) = b의 어떤 부분 문자열에 매치해야 한다. x는 전칭한정 되어 있으므로 임의의 값으로 치환할 수 있는데, 여기에는 두 가지 방법이 있다. 하나는 ① x ↦ b이고 다른 하나는 ② x ↦ f b이다.

경우 ①에는 f x에 매치되는 고울 프랍의 부분 문자열은 f b이며 이것이 H의 좌변인 x로 대체되어야 하므로 결국 고울 프랍은 f b = b로 바뀌게 된다.

경우 ②에는 f x에 매치되는 부분 문자열은 f (f b)이며 이것이 H의 좌변인 x로 바뀌어야 하므로 고울 프랍은 경우 ①에서와 동일하게 f b = b로 바뀌게 된다.

고울 프랍이 f b = b이 되었으므로 이제 어떻게 해야 할지는 자명하다. ✓

위의 예에서 보듯이, 전칭문을 인수로 삼아 rewrite 책략을 사용할 때, 매치의 방법이 유일하지 않은 경우가 발생할 수 있다. 위의 예에서는 두 매치의 결과가 동일했으므로 아무 문제가 없었지만, 경우에 따라서는 두 매치의 결과가 다를 수도 있다. Coq에서 실제로 어떤 매치를 선택하는지는 이 예를 아무리 들여다 보아도 알 길이 없다. 다음의 예를 보자.

```
1 Theorem dbl_fn_applied_twice :
2   forall (f : nat -> nat),
3   (forall (x : nat), f x = 2 * x) ->
4   forall (n : nat), f (f n) = 4 * n.
5 Proof.
6   intros f H n.
```

```
7   rewrite -> H.
8   (* need mult_assoc to finish the proof *)
9 Qed.
```

라인 7을 실행했을 때 고울 프랍이 f (2 * n)이 될지, 아니면 2 * f n이 될지 궁금하다. (실은 어느 쪽이 되어도 증명을 완료하는 데는 문제가 없다. 하지만 아무튼 둘 중 어느 경우인지를 묻는 것은 타당한 질문이다.)

실험을 통해 어느 쪽인지 확인해 보기 바란다. (힌트: outermost-leftmost)

```
1 Theorem andb_eq_orb :
2   forall (b c : bool),
3   (andb b c = orb b c) ->
4   b = c.
5 Proof.
6   intros b c.
7   destruct b eqn:Eb.
8   - destruct c eqn:Ec.
9     + simpl. intros H. reflexivity.
10    + simpl. (* ... *)
11 (* ... *)
```

(해설). 이 문제는 왜 별이 3개인지 모르겠다. 지극히 평이하게 풀린다. 바로 이전의 문제보다 더 쉬운 것으로 보인다. bool 타입 변수 b와 c가 취할 수 있는 경우의 수는 4밖에 안 된다. 그러므로 b를 destruct 하고, 각각의 경우에 다시 c를 destruct하면 된다. 나는 이것 외에 다른 방법을 생각하지도 못하겠다.

앞서도 보았듯이 bool은 상수 생성자만 사용하므로 intro pattern은 필요 없다. 이제 라인 8까지는 그냥 쓸 수 있을 것이다. 그 다음은 자명하다. ✓

Course Late Policies, Formalized

이건 꽤 길다. Coq 증명 작성 훈련용으로 활용할 수도 있겠다. 그러나 특별히 새로 설명할 기법은 없다. 무엇보다도 나는 이 문제는 Coq의 장점을 잘 보여주지 못하는 예라고 생각한다. Coq는 증명에 특화된 언어다. 그러나 이 문제는 본질적으로 계산 문제이다. 이런 문제는 Coq이 아니라 Python 같은 범용 언어로 해결하는 것이 더 빠르고 자연스럽다고 생각한다. 이 두 가지 이유로 인하여 이 문제에 대해서는 설명하지 않고 그냥 지나가기로 하였다.

Binary Numerals

이 문제는 쉽고 재미가 없다. 그런데 다음 장 Induction에서 아주 흥미로운 예제와 연결된다. 그래서 이 문제는 다음 장에서 다시 설명할 것이다.

3

Induction

3.1 Separate Compilation

이 섹션은 Coq의 언어 자체보다는 다음의 두 토픽에 대한 것이다.

① Export 시스템

② 소스파일 *.v를 컴파일 하여 byte code *.vo 파일 만들기. 이렇게 함으로써 export를 효율적으로 할 수 있다.

Induction.v의 첫 부분에 보면

```
From LF Require Export Basics.
```

가 보인다. 이 명령을 실행함으로써 Basics.v에서 작업한 결과를 모두 import 하여 사용할 수 있게 된다.[1] 그런데 이를 제대로 실행하기 위해서는 준비작업이 필요하다—Basics.v를 컴파일 하여 얻은 Basics.vo 파일이 LF 디렉토리 내에 존재해야 한다.

이 준비작업이 되어 있지 않다면 위의 From LF ..에 붉은 밑줄이 그어 있을 것이고 마우스를 올리면 다음과 같은 메시지가 나타날 것이다.

```
Cannot find a physical path bound to logical path basics with prefix lf.
```

Bascis.vo뿐만 아니라 lf 디렉토리 내의 모든 .v 파일들을 컴파일 하여 .vo 파일을 만들어 놓는 것이 좋으며 이를 위하여 다음과 같은 2개의 커맨드를 터미널에서 실행한다. 물론 이 작업은 lf 디렉토리 내에서 실행해야 한다.

```
coq_makefile -f _CoqProject *.v -o Makefile
make
```

만일 .vo 파일들을 얻었음에도 불구하고 문제가 해결되지 않는다면 그것은 필경 VS Code가 LF를 잘못 인식하고 있기 때문일 것이다. 다음의 명령을 Induction.v 내에서 실행하면 LF의 값이 현재 어떻게 인식되고 있는지 알 수 있으며 이를 토대로 문제를 해결할 수 있을 것이다.

```
Print LoadPath.
```

[1]여기서 import 하는 커맨드에 Export를 사용하는 것이 이상하게 보일 수 있다. 원한다면 From LF Require Import Basics.로 써도 된다. 그러나 이렇게 하면 다른 파일이 Induction.v를 import 하게 될 경우, Basics.v는 포함되지 않는다.

3.2 Proof by Induction

$\forall n : \mathbb{N}, n + 0 = n$을 증명하기는 $\forall n : \mathbb{N}, 0 + n = n$을 증명하기보다 까다롭다. 후자는 `plus: nat -> nat -> nat`의 정의로부터 쉽게 `simpl`되므로 간단히 증명할 수 있다.

전자를 증명할 때 `destruct`를 사용해 보면 `n = S n'`인 경우에 다음과 같은 고울 프랍을 증명해야 하는 상황을 만나게 된다.

```
1 S n' + 0 = S n'
2 S (n' + 0) = S n'
```

고울 프랍이 라인 1인 상태에서 `simpl`을 실행하면 고울 프랍이 라인 2로 바뀌게 된다. 여기서 더 이상 어떻게 할 수 없다. 따라서 `destruct` 책략으로는 이 증명을 마무리 할 수 없다.

그런데 라인 2를 잘 보면 `n'+ 0 = n'`을 가설로 사용할 수만 있다면 이 증명은 `rewrite`를 써서 해결된다! 바로 이를 위하여 `induction` 책략을 사용한다.

`induction` 책략도 `destruct` 책략에서와 마찬가지로 `n = 0`인 경우와 `n = S n'`인 경우로 나누어 증명한다. 우리는 전자를 기저단계(*base case*), 후자를 귀납단계(*inductive case*)라고 부른다. `induction` 책략이 `destruct` 책략보다 나은 점은 전자에서는 귀납단계에서 귀납가설(*induction hypothesis*)을 추가로 사용할 수 있다는 것이다.

$\forall n \in \mathbb{N}, P(n)$을 `induction`을 사용하여 증명하고자 할 때 다음의 두 가지를 증명하면 된다.[2]

① 기저단계에서 $P(0)$을 증명하고,

② 귀납단계에서 $P(n') \rightarrow P(S\, n')$을 증명한다.

다음의 정리를 보라.

```
1 Theorem add_0_r : forall n:nat, n + 0 = n.
2 Proof.
3   intros n. induction n as [| n' IHn'].
4   - (* n = 0 *) simpl. reflexivity.
5   - (* n = S n' *) simpl. rewrite -> IHn'. reflexivity.  Qed.
```

라인 3의 구문을 잘 익혀 두어야 한다. `destruct` 경우와 비슷하지만 intro pattern에 `IHn'`이 추가된 부분이 다르다. `IHn'`은 귀납가설에 주는 이름이다. `IH`는 Induction Hypothesis의 머리글자를 딴 것이며 귀납가설의 이름에 이것을 앞붙이는 것이 관례다.

Annotation `eqn:E`는 induction에서 써도 되지만 이 경우 귀납가설이 조금 불편한 형태로 나타나므로 쓰지 않는 경우가 많다.

라인 3까지 실행한 후의 `Coq Goals` 화면은 다음과 같다.

```
6  Goal 1
7  (1 / 2) ----------------------
8  0 + 0 = 0
9
10 Goal 2
11 n' : nat
12 IHn' : n' + 0 = n'
13 (2 / 2) ----------------------
14 S n' + 0 = S n'
```

[2] 이렇게 하면 된다는 것이 자연수에 대한 수학적 귀납법의 원리(Principle of Mathematical Induction on $\mathbb{N}$)이다.

Goal 1은 아주 쉽게 해결된다.

Goal 2에서는 IHn'을 가설로 사용할 수 있다. IHn'은 'n''에 대한 Induction Hypothesis'를 뜻한다. 라인 5의 simpl.까지 실행하면 고울이 다음과 같이 바뀐다.

```
S (n' + 0) = S n'
```

그 다음에 rewrite -> IHn'.를 실행하면 고울이 다음과 같이 바뀐다.

```
S n' = S n'
```

이제 reflexivity.를 실행하면 증명이 끝난다. ✓

다음의 정리는 기억해 둘 가치가 있다. 다른 정리들의 증명에 유용하게 쓰일 것이다.

```
Theorem plus_n_Sm : forall n m : nat,
  S (n + m) = n + S m.
```

참고로 원래 plus의 정의는 S n + m = S (n + m)이다. 이 정의식에서 좌변의 S n + m을 n + S m으로 바꾸고, 이어서 등식의 좌우변을 서로 바꾸면 바로 위의 정리 plus_n_Sm이 된다. 이 정리의 증명은 앞에서 증명한 add_0_r의 증명과 똑같다.

다음의 정리도 후에 쓸 일이 많다. 증명은 간단하다.

```
Theorem eqb_refl : forall n : nat,
  (n =? n) = true.
Proof. Admitted.
```

Proofs Within Proofs

수학의 증명 과정에서 증명 내에 증명이 등장하는 경우가 흔히 있다. 예를 들어

$$\forall n\, m : \mathbb{N},\ (n + 0 + 0) \times m = n \times m$$

을 증명하고자 할 때 일단 $n + 0 + 0 = n$을 증명해 놓는 것이 좋을 것이다. 이를 Coq에서는 다음과 같이 구현할 수 있다. 증명 중에 사용한 정리 plus_n_O는 $\forall n : \mathbb{N},\ n + 0 = n$를 뜻한다.

```
1  Theorem mult_0_plus' : forall n m : nat,
2    (n + 0 + 0) * m = n * m.
3  Proof.
4    intros n m. (* not use induction *)
5    assert (H: n + 0 + 0 = n).
6      { rewrite <- plus_n_O. rewrite <- plus_n_O. reflexivity. }
7    rewrite -> H. reflexivity.
8    Qed.
```

assert의 구문을 잘 익혀 두기 바란다. 증명하고자 하는 프랍, 즉 *assertion*은 나중에 가설로써 사용될 것이므로 적당한 이름, 예를 들어 H를 준다. Assertion을 괄호쌍으로 두르고 마침표를 찍는다. assert를 위한 또 하나의 구문이 있다. 라인 5를 다음과 같이 바꾸어도 된다.

```
assert (n + 0 + 0 = n) as H.
```

그 다음에 바로 이어서 assertion의 증명을 중괄호 쌍에 넣는다.

assert 책략을 쓰는 과정에서 꼭 알아두어야 할 것은 assert를 실해한 직후의 Coq Goals 화면이다. 위의 예에서는 다음과 같다.

```
2  Goal 1
3  n, m : nat
4  (1 / 2) -----------------------
5  n + 0 + 0 = n
6
7  Goal 2
8  n, m : nat
9  H : n + 0 + 0 = n
10 (2 / 2) -----------------------
11 (n + 0 + 0) * m = n * m
```

Assertion이 Goal 1으로 삽입되었고 원래의 Goal 1은 Goal 2로 밀려났다. 이제 통상적인 절차를 밟아 Goal 1을 증명하고 이어서 원래의 고울을 증명하면 된다.

assert 책략은 다음과 같은 경우에도 흔히 쓰인다. 고울을 척 보면 잘 알려진 정리를 사용하면 쉽게 해결될 것 같은데, 고울의 형태가 원래 정리에 딱 들어맞지 않아서 정리를 살짝 바꾸어 고울에 맞추고 싶을 때가 있다. 예를 들어

$$(n + m) + (p + q) = (m + n) + (p + q) \tag{3.1}$$

를 증명하고자 할 때 누구든지 add_comm 정리를 사용해야 한다는 것을 쉽게 알 것이다. 그러나 이를 (3.1)의 좌변에 직접 적용하면 우리는 원하는 결과가 아닌 $(p + q) + (m + n)$를 (3.1)의 좌변에 얻게 된다.[3] 이런 경우 $H : n+m = m+n$을 먼저 assert하고 이를 증명한 후에 add_comm 대신 H를 가설로 써 먹으면 된다.

```
1  Theorem plus_rearrange : forall n m p q : nat,
2    (n + m) + (p + q) = (m + n) + (p + q).
3  Proof.
4    intros n m p q.
5    assert (H: n + m = m + n).
6    { rewrite add_comm. reflexivity. }
7    rewrite H. reflexivity.  Qed.
```

지금쯤은, 위의 정리 정도는 Coq Goals 화면을 보지 않고도 이해할 수 있을 것이다.

라인 6에 정리 add_comm이 사용되었는데, 다음의 정리들은 앞으로 자주 사용할 것이니 눈여겨 보기 바란다. 이들은 Induction과 assert를 써서 쉽게 증명할 수 있으므로 반드시 모두 스스로 증명해 보기 바란다.

```
add_0_r: forall n: nat, n + 0 = n
mul_0_r: forall n: nat, n * 0 = 0
plus_n_Sm: forall n m: nat, S (n + m) = n + (S m)
add_comm: forall n m: nat, n + m = m + n
add_assoc: forall n m p: nat, n + (m + p) = n + m + p
```

[3] 이것은 Coq의 *outermost-leftmost* 원칙에 의한 것이다.

add_assoc의 우변에 괄호가 없는 것은 +의 왼쪽결합성left associativity 때문이다.

증명 속에서 증명을 사용하는 데 사용하는 또 하나의 책략이 있다. 바로 replace 책략이다. 이를 이용하여 (3.1)을 증명해 보자.

```
1 Theorem plus_rearrange' : forall n m p q : nat,
2   (n + m) + (p + q) = (m + n) + (p + q).
3 Proof.
4   intros n m p q.
5   replace (n + m) with (m+n).
6   { reflexivity. }
7   rewrite add_comm. reflexivity. Qed.
```

assert와 비슷하지만 일단 고울을 변경시켜 놓고 증명한 다음에 replace에 대한 수습 증명을 한다는 점에서 다르다. 가설을 도입하지도 않으며 그 가설을 써서 rewrite 하지도 않는다는 점에서 assert보다 약간 더 편리하다.

replace 책략은 assert의 가설이 등식인 경우에만 사용할 수 있다는 점에서 응용범위는 좁다고 말할 수 있다. 그러나 원하는 가설이 등식인 경우에는 앞서 말했듯이 replace가 조금 더 편리하다.

assert 책략을 사용할 때는 그것이 도입한 가설의 증명을 먼저 하고, 그 다음에 원래의 고울을 증명하도록 되어 있는데, 많은 경우 이 순서를 바꾸고 싶을 수 있다. assert가 도입하는 가설은 대부분의 경우 증명할 수 있음을 이미 알고 있으므로 그것을 먼저 증명한 다음에 혹시 이 가설이 원래의 증명에 도움이 되지 않는다는 것을 발견하게 된다면 이건 시간만 허비한 셈이 된다. assert 가설을 고울을 증명한 뒤에 증명하려면 다음과 같이 하면 된다.

> 원래는 assert (H: assertion_prop). { proof of H }의 구문을 사용하지만 이를 바꿔 assert (H: assertion_prop). admit.로 두고 고울의 증명을 계속한다. 고울의 증명을 마친 후에 admit.를 { proof of H }로 바꾼다.

———— ○ ————

plus_rearrange를 증명하는 또 하나의 방법이 있다. rewrite .. with 책략이 그것이다.

```
1 Theorem plus_rearrange'' : forall n m p q : nat,
2   (n + m) + (p + q) = (m + n) + (p + q).
3 Proof.
4   intros n m p q.
5   rewrite add_comm with (n:=n) (m:=m).
6   (* rewrite add_comm with (n:=n). *)
7   reflexivity. Qed.
```

Check add_comm을 실행해 보면 rewrite add_comm이 하는 일은 n과 m의 위치를 맞바꾸는 것이다. 그런데 인수 없이 rewrite add_comm을 실행하면 outermost-leftmost 원칙에 따라 n := n + m, m := p + q의 치환이 자동으로 적용된다. 자동 치환을 방지하고 n의 역할을 할 표현항term과 m의 역할을 할 표현항을 직접 지정하려면 rewrite .. with .. 구문을 쓰면 된다.

라인 5 대신 라인 6을 써도 된다. Coq의 자동 치환 알고리듬은 일부 인수를 지정했을 때 나머지 인수에 대해 적용된다.

라인 5에서 with (n:=n) (m:=m)을 with (n:=p) (m:=q)로 바꾸어 실행하면 어떤 결과가 나올지 예측하고, 실제 관찰한 바를 설명해 보라.

라인 5는 다음과 같이 써도 된다.

```
rewrite (add_comm n m).
```

일반적으로 책략의 인수에 인수를 줄 때

```
tactic my_thrm with (x1:=v1) .. (xn:=vn).
```

대신 더 간단한 구문인

```
tactic (my_thrm v1 .. vn).
```

를 쓸 수 있다.

Formal vs. Informal Proof

Coq를 공부하다 보면 수학적 명제의 증명을 위하여, 증명 과정 전체의 조망 없이, 덮어놓고 여러 책략을 시도해 보는 경우가 있다. 간단한 증명이라면 이런 방식으로도 충분히 오류 없는 증명을 얻을 수 있다. 하지만 조금만 내용이 있는 명제라면 이런 방식으로는 증명에 실패하는 것이 일반적이다.

Coq 증명을 얻기 위하여는, 아주 간단한 경우가 아니라면, 먼저 그 증명을 수학자가 평소에 하듯이 펜으로 종이에 써 보고 나서, 이 informal proof를 Coq의 formal proof로 번역하는 것이 좋다.

3.3 Exercises

- ```
  Theorem mul_comm : forall m n : nat,
    m * n = n * m.
  ```

  이 정리는 m에 대한 귀납으로 증명한다. 다음의 보조정리lemma를 사용하면 좋을 것이다.

- ```
  Lemma mult_n_Sm : forall n m : nat,
    n * m + n = n * S m.
  ```

  mult_n_Sm은 표준 라이브러리에 있으므로 그냥 사용해도 되지만 스스로 증명해 보는 것을 권한다. 이 보조정리는 n에 대한 귀납으로 증명한다. plus_n_Sm과 add_assoc을 사용하면 편하다.

- ```
  Theorem plus_leb_compat_l : forall n m p : nat,
    n <=? m = true -> (p + n) <=? (p + m) = true.
  ```

  이 정리에 대한 귀납은 p에 대해서 하는 것이 좋다.

- ```
  Theorem mult_plus_distr_r : forall n m p : nat,
    (n + m) * p = (n * p) + (m * p).
  ```

  평범한 n에 대한 귀납으로 해결된다. 귀납단계에서 add_assoc을 사용해야 할 것이다.

- ```
  Theorem mult_assoc : forall n m p : nat,
    n * (m * p) = (n * m) * p.
  ```

  평범한 n에 대한 귀납으로 해결된다. 귀납단계에서 mult_plus_distr_r을 사용해야 할 것이다.
  ```

Nat to Bin and Back to Nat

자연수를 2진법으로 표시하는 다음의 방법은 Basics.v에서 소개된 바 있다.

```
Inductive bin : Type :=
  | Z
  | B0 (n : bin)
  | B1 (n : bin).
```

이 정의가 의미를 가지기 위하여는 바로뒤 원소successor 함수가 있어야 한다. 이를 incr라는 이름을 주어 다음에 정의하였다. incr가 bin에서 하는 일은 S가 nat에서 하는 일과 같다.

```
Fixpoint incr (m:bin) : bin :=
  match m with
  | Z => B1 Z
  | B0 m' => B1 m'
  | B1 m' => B0 (incr m')
  end.
```

통상적인 2진법과 위에서 정의한 2진법을 비교하여 아래에 표로 보였다.

10진법	2진법	Inductive bin: Type
0	0	Z
1	1	B1 Z
2	10	B0 (B1 Z)
3	11	B1 (B1 Z)
4	100	B0 (B0 (B1 Z))
5	101	B1 (B0 (B1 Z))
6	110	B0 (B1 (B1 Z))
⋮	⋮	⋮

통상적인 2진법과 bin을 비교해 보면 B0은 0, B1은 1에 대응되는데, 2진법에서는 왼쪽에 있는 수가 높은 자릿수이지만 bin에서는 오른쪽에 있는 수가 높은 자릿수라는 점이 다르다. 이는 matching 계산의 효율성을 위한 것이다. 그리고 0를 나타내는 bin이 공문자열이 되는 것을 방지하기 위하여 Z를 일종의 end-marker로 사용하였다.

bin으로 표현된 수를 nat로 변환하는 함수는 다음과 같다.[4]

```
Fixpoint bin_to_nat (m:bin) : nat :=
  match m with
  | Z => 0
  | B0 m' => 2 * (bin_to_nat m')
  | B1 m' => 1 + 2 * (bin_to_nat m')
  end.
```

incr와 bin_to_nat의 unit test를 아래 보였다.

---

[4]이 함수가 하는 일을 정확하게 말하면 다음과 같다: 임의의 bin 표현이 주어졌을 때 그것이 나타내는 자연수의 nat 표현을 리턴한다.

```
Example test_bin_incr1 : (incr (B1 Z)) = B0 (B1 Z).
Proof. simpl. reflexivity. Qed.
Example test_bin_incr2 : (incr (B0 (B1 Z))) = B1 (B1 Z).
Example test_bin_incr3 : (incr (B1 (B1 Z))) = B0 (B0 (B1 Z)).
Example test_bin_to_nat1 : bin_to_nat (B0 (B1 Z)) = 2.
Example test_bin_to_nat2:
        bin_to_nat (incr (B1 Z)) = 1 + bin_to_nat (B1 Z).
Example test_bin_bin_to_nat3 :
        bin_to_nat (incr (incr (B1 Z))) = 2 + bin_to_nat (B1 Z).
```

위의 Example들에 대한 모든 증명은 Proof. simpl. reflexivity. Qed.이므로 첫 번째 예를 제외한 모든 예에서 증명을 생략하였다.

역으로 nat를 bin으로 변환하는 함수는 다음과 같다.

```
Fixpoint nat_to_bin (n:nat) : bin :=
  match n with
  | O => Z
  | S n' => incr (nat_to_bin n')
  end.
```

이 두 함수들의 정의를 보면 비교적 간단하므로 틀린 부분이 없는 것 같다. 하지만 Coq에서는 모든 직관을 배제하고 오로지 정해진 규칙에 따른 기계적 계산에 의하여 이 두 함수들이 서로 역함수라는 것을 증명해야 한다.

다음은 우리가 증명하고자 하는 정리의 한 방향이다.

```
Theorem nat_bin_nat : forall n: nat,
  bin_to_nat (nat_to_bin n) = n.
```

역방향은 더 어려우므로 조금 후에 다루기로 한다.

nat_to_bin은 n에 대한 귀납으로 증명해야 할 것은 당연한데, 귀납단계에서 해야 할 일은 귀납가설인 (3.2)를 가정하고 (3.3)을 증명하는 것이다.

bin_to_nat (nat_to_bin n') = n' (3.2)

bin_to_nat (nat_to_bin (S n')) = S n' (3.3)

nat_to_bin의 정의에 의하여 (3.3)의 좌변은

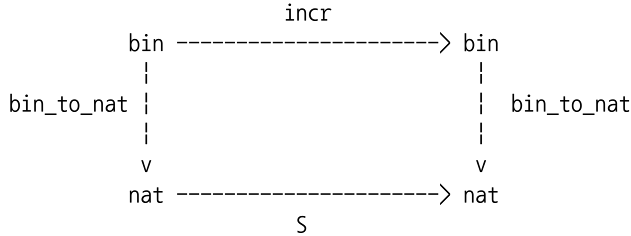
bin_to_nat (incr (nat_to_bin n')) (3.4)

과 같다. 만일 (3.4)에서 incr을 밖으로 끄집어 내고 S로 바꿔

S (bin_to_nat (nat_to_bin n')) (3.5)

로 변환할 수 있다면 이 표현 (3.5)에 귀납가설 (3.2)를 활용하여 우리가 원하는 결과인 S n'을 얻을 수 있게 된다.

(3.4)를 (3.5)로 변환하는 것은 다음과 같은 자연스러운 가환그림commutative diagram에서 영감을 얻은 것이다. 이 그림의 좌상귀에 있는 bin에 nat_to_bin n' 을 넣었을 때 그림 아래에 보인 정리의 프랍 bin_to_nat_pres_incr가 나온다.



이 가환그림을 다음의 정리로 나타내고 증명하였다.

```
1  Theorem bin_to_nat_pres_incr : forall b : bin,
2    bin_to_nat (incr b) = S (bin_to_nat b).
3  Proof.
4    intros b. induction b as [|b'|b''].
5    - simpl. reflexivity.
6    - simpl. reflexivity.
7    - simpl. rewrite IHb''. (* ... *)
8    (* ... *)
```

(해설). (정리 이름에 나타나는 pres는 preserve를 뜻하는 것으로 보인다.) 라인 4의 induction은 bin 타입 원소 b에 대한 것인데 bin 타입은 3개의 생성자(arity는 각각 0, 1, 1)를 사용하여 정의되었으므로 induction의 intro pattern을 [|b'|b'']으로 잡았다.

이 증명의 계산에서 2 * t 형태의 표현이 자주 나타나는데 이것을 simpl 하면 다음과 같이 된다는 것을 알고 있으면 좋을 것이다.

```
2 * t => S (S 0) * t => t + (S 0) * t => t + (t + 0)
```

라인 7에서 simpl 후에는 귀납가설을 적용할 수 있다. 이후에는 plus_n_Sm와 add_0_r을 몇 번 사용하게 될 것이다. 이 정도면 bin_to_nat_pres_incr의 증명을 완성하는 데에 필요한 힌트로 충분하다고 본다. ✓

이제 본 정리를 증명할 준비가 되었다.

```
1  Theorem nat_bin_nat : forall n: nat,
2    bin_to_nat (nat_to_bin n) = n.
3  Proof.
4    intros n. induction n as [|n' IHn'].
5    - simpl. reflexivity.
6    - simpl. rewrite bin_to_nat_pres_incr.
7    (* ... *)
```

보조정리 bin_to_nat_pres_incr을 사용하면 본 정리의 증명은 의외로 간단하다. 라인 6 후에는 귀납가설을 사용할 수 있게 되고 그 다음은 아주 쉽다. ✓

Bin to Nat and Back to Bin (Advanced)

이제 nat_bin_nat의 역 정리 bin_nat_bin를 증명할 차례다. 그런데 만일 이 정리를 다음과 같이 놓는다면 이것은 증명할 수 없다.

```
Theorem bin_nat_bin_fails : forall b: bin,
  nat_to_bin (bin_to_nat b) = b.
Proof. Abort.
```

왜냐하면 이 정리가 증명하고자 하는 프랍은 참이 아니기 때문이다!

```
Compute (nat_to_bin (bin_to_nat (B0 Z))). (* Z *)
Compute (nat_to_bin (bin_to_nat (B1 (B0 Z)))). (* B1 Z *)
Compute (nat_to_bin (bin_to_nat (B0 (B0 Z)))). (* Z *)
```

`B0 Z` 같은 표현은 `nat_to_bin` 함수의 값이 될 수 없다. (증명할 수 있겠는가?)

`bin`의 원소들 중에 `nat_to_bin` 함수의 값이 될 수 있는 것들을 정규형식normal form이라고 부르기로 한다. `n: nat`에 대해서 `bin_to_nat b = n`인 `b: bin`은 일반적으로 무한히 많이 존재한다.(증명할 수 있겠는가?)

'`b: bin`의 정규형식'은 정규형식인 `b': bin`으로서 `bin_to_nat b'= bin_to_nat b`를 만족하는 것을 뜻하는 것으로 정의하자. 그리고 `b: bin`를 입력 받아 이것의 정규형식을 계산하는 함수를 `normalize: bin -> bin`이라고 하면 이는 다음과 같이 Coq 코드로써 정의할 수 있다.

라인 4에서 사용한 `match .. with` 구문은 지금까지 못 보았던 형식이다. 지금까지는 `match` *variable* 형식이었지만 이번에는 `match` *term_expression* 형식을 사용하였다.

```
1  Fixpoint normalize (b: bin) : bin :=
2    match b with
3    | Z => Z
4    | B0 b0 => match normalize b0 with
5      | Z => Z
6      | _ => B0 (normalize b0)
7      end
8    | B1 b1 => B1 (normalize b1)
9    end.
10
11 Compute(normalize Z). (* Z *)
12 Compute(normalize (B0 Z)). (* Z *)
13 Compute(normalize (B0 (B0 Z))). (* Z *)
14 Compute(normalize (B1 (B0 Z))). (* B1 Z *)
15 Compute(normalize (B0 (B0 (B1 (B0 Z))))). (* B0 (B0 (B1 Z)) *)
16 Compute(normalize (B0 (B1 (B0 (B0 Z))))). (* B0 (B1 Z) *)
```

이제 우리가 증명하려는 메인 정리를 `normalize`를 이용하여 다음과 같이 쓸 수 있다.

```
Theorem bin_nat_bin : forall (b: bin),
  nat_to_bin (bin_to_nat b) = normalize b.
```

이 정리를 증명하기 위하여 이것의 역방향 정리 격인 `nat_bin_nat`의 증명을 흉내내어 보자. 그 증명에서 사용했던 핵심적인 보조정리는 가환그림을 나타낸 다음의 정리였다.

```
Theorem bin_to_nat_pres_incr : forall b : bin,
  bin_to_nat (incr b) = S (bin_to_nat b).
```

이 보조정리에 대응되는 정리로 다음을 생각할 수 있다.

```
Theorem nat_to_bin_pres_S : forall n : nat,
  nat_to_bin (S n) = incr (nat_to_bin n).
```

n에 여러 값을 넣어 보면 이 보조정리는 참인 것으로 확인된다. 증명도 아주 쉽게 찾을 수 있다. 그러나 이 보조정리는 bin_nat_bin의 증명에 도움이 되지 않는다! 왜냐하면 bin_nat_bin의 증명은 b: bin에 대한 귀납을 사용해야 할 것인데 b의 정의를 보면 생성자 B0 혹은 B1을 적용했을 때 $+1$씩 증가하는 것이 아니라 2배, 혹은 2배 $+ 1$씩 증가하기 때문이다.

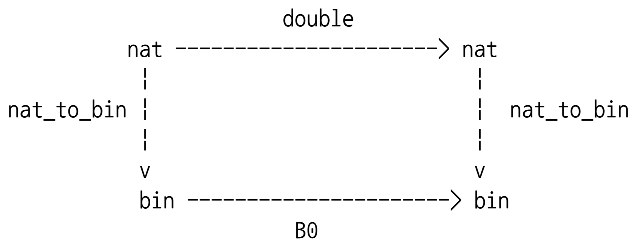
시행착오를 거친 끝에 다음의 보조정리 nat_to_bin_double_n이 유용함을 알게 되었다. 이 보조정리의 증명에 사용하는 간단한 하위 보조정리 incr_twice_B0를 먼저 보였다.

```
Lemma incr_twice_B0 : forall b: bin,
  incr (incr (B0 b)) = B0 (incr b).
Proof. intros n. simpl. reflexivity. Qed.
```

다음의 보조정리는 n > 0인 경우에는 n을 두 배 하는 것이 B0를 앞에 덧붙이는 것에 대응한다는 것을 말하고 있다. n > 0 조건을 위하여 n 대신 S n을 사용하였다. 그리고 이제부터 $n \mapsto 2n$ 함수를 double이라는 이름으로 다음과 같이 정의하여 사용한다.

```
1  Fixpoint double (n:nat) :=
2    match n with
3    | O => O
4    | S n' => S (S (double n'))
5    end.
6
7  Lemma nat_to_bin_double_n : forall (n : nat),
8    nat_to_bin (double (S n)) = B0 (nat_to_bin (S n)).
9  Proof. intros n.
10   induction n as [| n' IHn'].
11   - simpl. reflexivity.
12   - replace (double (S (S n'))) with (S (S (double (S n')))).
13     + replace (nat_to_bin (S (S (double (S n'))))) with
14         (incr (incr (nat_to_bin (double (S n'))))).
15       { rewrite IHn'. rewrite incr_twice_B0.
16         replace (nat_to_bin (S (S n'))) with
17           (incr (nat_to_bin (S n'))).
18         - reflexivity.
19         - simpl. reflexivity. }
20       { simpl. reflexivity. }
21     + simpl. reflexivity.
22   Qed.
```

위의 보조정리를 가환그림으로 나타내면 다음과 같다. 좌상귀의 nat 자리에 S n을 놓으면 된다.



증명에서 사용할 간단한 보조정리 하나를 추가한다.

```
Lemma dbl_lem0 : forall n: nat,
  n + (n + 0) = double n.
Proof. intros n. rewrite <- plus_n_O.
  induction n.
  - simpl. reflexivity.
  - simpl. rewrite <- plus_n_Sm. rewrite IHn.
    reflexivity. Qed.
```

다음은 메인 정리의 증명이다. 세 부분으로 나누어 설명하겠다.

```
1  (* Part 1 of 3 *)
2  Theorem bin_nat_bin : forall (b: bin),
3    nat_to_bin (bin_to_nat b) = normalize b.
4  Proof.
5    induction b as [| b0 IHb0 | b1 IHb1].
6    - (* b = Z *) simpl. reflexivity.
7    - (* b = B0 b0 *) simpl. rewrite <- IHb0.
```

라인 6에서 simpl 후에 고울 프랍에 match가 나타난다. 이런 예는 이 책에서 처음이다. 그 다음의 rewrite <- IHb0를 실행한 결과 그렇잖아도 복잡했던 고울이 더욱 더 길어졌다. 하지만 이렇게 함으로써 고울 프랍에서 normalize가 제거되었다는 이점을 얻었다. 새로운 고울 프랍은 다음과 같다.

```
nat_to_bin (bin_to_nat b0 + (bin_to_nat b0 + 0)) =
  match nat_to_bin (bin_to_nat b0) with
  | Z => Z
  | _ => B0 (nat_to_bin (bin_to_nat b0))
  end
```

이 고울 프랍에는 bin_to_nat b0라는 비교적 긴 표현이 4번이나 나타난다. 이는 아래에 보인 라인 9의 destruct (bin_to_nat b0)에 의하여 제거된다.

라인 9에서 사용한 destruct .. 구문은 지금까지 못 보았던 형식이다. 지금까지는 destruct *variable* 형식이었지만 이번에는 destruct *term_expression* 형식을 사용하였다.

라인 9와 라인 13에서 사용한 annotation eqn:E_nat와 eqn:E_bin은 증명에 직접 사용되지는 않지만 Coq Goals 화면을 읽으면서 증명을 이해하는 데 도움이 된다.

```
8  (* Part 2 of 3 *)
9      destruct (bin_to_nat b0) eqn:E_nat.
10     + (* 0 *) simpl. reflexivity.
11     + (* S n *) rewrite dbl_lem0.
12       rewrite nat_to_bin_double_n. simpl.
13       destruct (nat_to_bin n) eqn:E_bin.
14       { (* Z *) simpl. reflexivity. }
15       { (* B0 n0 *) simpl. reflexivity. }
16       { (* B1 n0 *) simpl. reflexivity. }
```

라인 12의 rewrite nat_to_bin_double_n이 이 증명의 핵심이며 나머지는 사실 아주 단순한 계산이다.

다음의 (Part 3 of 3)는 b = B1 b1 경우에 대한 것인데 b = B0 b0 경우와 거의 똑같은 방식으로 증명된다.

```
17 (* Part 3 of 3 *)
18   - (* b = B1 b1 *) simpl. rewrite <- IHb1.
19     destruct (bin_to_nat b1) eqn:E_nat.
20     + (* 0 *) simpl. reflexivity.
21     + (* S n *) rewrite dbl_lem0.
22       rewrite nat_to_bin_double_n. simpl.
23       (* ... *)
24   Qed.
```

생략된 부분을 채워 넣는 것은 어렵지 않을 것이다. ✓

이건 전에 한 번 말했던 것인데, 이 정도의 수학적 내용이 있는 정리라면 증명을 찾을 때, 먼저 informal proof를 작성한 후에 이것을 formal한 Coq proof로 번역하는 방식을 사용하는 것이 좋을 것이다. 무작정 이리 저리 계산하면서 요행으로 reflexivity를 사용할 수 있게 되기를 바래서는 잘 안 될 것이다.

위에서 보인 메인 정리 bin_nat_bin의 증명은 따라가면서 이해하기는 그리 어렵지 않으나 증명을 찾기는 상당히 까다로울 수 있다. 실은 이 책에서 아직 배우지 않은 증명 책략을 이용하면 조금 더 쉽게 직관적이고 자연스럽게 증명을 찾을 수 있다. 예를 들어 다음의 보조정리를 사용하는 증명을 찾아 보는 것은 좋은 연습문제가 될 수 있다.

```
1 Lemma Sn2bin_not_Z: forall n: nat,
2   nat_to_bin (S n) = Z -> False.
3 Proof. intros n. simpl. destruct (nat_to_bin n).
4   - simpl. intros. discriminate H.
5   - simpl. intros. discriminate H.
6   - simpl. intros. discriminate H.
7 Qed.
```

4 Lists

4.1 Pairs of Numbers

자연수의 순서쌍 타입 natprod를 다음과 같이 정의한다.

```
Inductive natprod : Type :=
  | pair (n1 n2 : nat).
```

nat의 정의에 pred, plus 등의 nat에 대한 연산 혹은 함수의 정의들을 보충하지 않으면 nat라는 데이터 타입이 별 의미가 없듯이, natprod도 이를 활용하기 위한 함수가 없이는 쓸모가 없다.

주어진 natprod 원소의 각 성분을 추출하는 함수들 fst와 snd를 다음과 같이 정의한다.

```
Definition fst (p : natprod) : nat :=
  match p with
  | pair x y => x
  end.

Definition snd (p : natprod) : nat :=
  match p with
  | pair x y => y
  end.
```

순서쌍의 표준 표기법을 사용하도록 한다.

```
Notation "( x , y )" := (pair x y).
```

이제 순서쌍의 성분들을 맞바꾸는 함수 swap_pair를 다음과 같이 정의할 수 있다.

```
Definition swap_pair (p : natprod) : natprod :=
  match p with
  | (x,y) => (y,x)
  end.
```

두 개 이상의 인수를 한꺼번에 match 하는 *multiple pattern*과 순서쌍 표현을 혼동하지 않아야 한다.

```
Definition bad_fst (p: natprod): nat :=
  match p with
```

```
  | x, y => x (* x, y is a multiple pattern *)
  end.

Definition good_fst (p: natprod): nat :=
  match p with
  | (x, y) => x (* (x, y) is a pair expression *)
  end.
  end.

Fixpoint bad_minus (n m: nat): nat :=
  match n, m with
  | (O, _) => O (* pair doesn't work in multiple matching *)
  | (S _, O) => n
  | (S n', S m') => bad_minus n' m'
  end.

Fixpoint good_minus (n m: nat): nat :=
  match n, m with
  | O, _ => O
  | S _, O => n
  | S n', S m' => good_minus n' m'
  end.
```

간단한 정리 하나를 증명해 보자.

```
Theorem surjective_pairing : forall (p : natprod),
  p = (fst p, snd p).
Proof.
  intros p. destruct p as [n m].
  simpl. reflexivity. Qed.
```

`destruct`는 생성자가 하나만 있는 경우에도 이렇게 활용될 수 있다. 이런 경우에는 intro pattern에서 세로선 |를 사용하지 않는다. 생성자의 인수가 2개 이므로 두 변수를 intro pattern에 `as [n m]`과 같이 넣으면 된다.

생성자가 개수가 n이라면 separator로 쓰이는 세로선은 $n-1$개를 사용하게 된다. 예를 들어 생성자 3개를 사용하는 귀납적 타입에서 이들의 애리티arity가 각각 0, 1, 2 라면 다음과 같은 형태의 구문을 사용하게 된다.

```
destruct x as [| a' | b' c'].
```

`surjective_pairing`라는 이름은 `pairing` 함수가 전사, 즉 `pairing` 함수의 공역의 모든 원소는 `pairing` 함수의 어떤 인수에서의 값이 된다는 것을 의미한다. 그러므로 이 정리가 증명하는 프랍은 원래 다음과 같이 되어야 할 것이다.

```
forall (p : natprod), exists (a b: nat), p = (a, b)        (4.1)
```

우리가 `surjective_paring`에서 증명한

```
forall (p : natprod), p = (fst p, snd p).
```

로부터 (4.1)을 쉽게 증명할 수 있다.[1] 어쩌면 `surjective_paring`은 사실 `surjective_paring_lemma`가 더 나은 이름이었는지도 모르겠다.

4.2 Lists of Numbers

리스트(*list*)는 어떤 언어에서든지 대단히 중요한 데이터 타입이다. `Coq`에서는 리스트를 다음과 같이 정의한다. 여기서는 일단 자연수의 리스트만 정의하기로 한다.

```
Inductive natlist : Type :=
  | nil
  | cons (n : nat) (l : natlist).

Notation "x :: l" := (cons x l)
                     (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).
```

상수 생성자 `nil` 하나와 2항 생성자 `cons` 하나를 사용한다. 후자의 인수 2개는 타입이 서로 다름에 유의하라.

연산자 `::`는 오른쪽 결합이다. 이것은 `1 :: (2 :: (3 :: nil))`은 `1 :: 2 :: 3 :: nil`로 써도 된다는 뜻이다. 실은 `(1 :: 2) :: 3 :: nil` 같은 표현은 문법에도 어긋난다. 마지막에 보인 `Notation` 정의는 예를 들어 [1; 2; 3] 같은 표현으로 `1 :: (2 :: (3 :: nil))`을 나타낼 수 있다는 의미이다.

`::`의 우선순위는 자연수 연산들보다 약하므로 예를 들어 `1 + 2 :: [3]`은 `(1 + 2) :: [3]`을 뜻하게 된다.

리스트에 대한 몇 가지 함수들을 정의해 보자.

```
1  Fixpoint repeat (n count: nat) : natlist :=
2    match count with
3    | 0 => nil
4    | S count' => n :: (repeat n count')
5    end.
6
7  Fixpoint length (l: natlist) : nat :=
8    match l with
9    | nil => 0
10   | h :: t => S (length t)
11   end.
```

두 리스트를 이어붙이는 함수 `app`을 다음과 같이 정의한다. `app`은 APPend를 뜻한다.

```
12 Fixpoint app (l1 l2: natlist) : natlist :=
13   match l1 with
14   | nil => l2
15   | h :: t => h :: (app t l2)
16   end.
17
```

[1] `exists`가 포함된 프랍을 증명하는 방법은 나중에 배우게 된다.

```
18 Notation "x ++ y" := (app x y)
19                      (right associativity, at level 60).
20
21 (** head element of a list *)
22 Definition hd (default: nat) (l: natlist) : nat :=
23   match l with
24   | nil => default
25   | h :: t => h
26   end.
27
28 Compute hd 99 [7; 0; 4]. (* = 7 *)
29 Compute hd 99 []. (* = 99 *)
30
31 (** tail list of a list *)
32 Definition tl (l: natlist) : natlist :=
33   match l with
34   | nil => nil
35   | h :: t => t
36   end.
```

인수 `default : nat`가 필요한 이유는 `nil`이 입력되었을 때의 리턴값의 타입을 `nat`로 맞추어 주어야 하기 때문이다.

연습문제 4.1 다음과 같은 성질을 가지는 함수 `alternate (l1 l2: natlist): natlist`를 만들어 보자.

```
Example test_alternate1:
  alternate [1;2;3] [4;5;6] = [1;4;2;5;3;6].
Proof. simpl. reflexivity. Qed.

Example test_alternate2:
  alternate [1] [4;5;6] = [1;4;5;6].
Proof. simpl. reflexivity. Qed.

Example test_alternate3:
  alternate [1;2;3] [4] = [1;4;2;3].
Proof. simpl. reflexivity. Qed.

Example test_alternate4:
  alternate [] [20;30] = [20;30].
Proof. simpl. reflexivity. Qed.
```

힌트: 다음과 같이 시작하는 것이 당연하다.

```
Fixpoint alternate (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil => l2
  | h1 :: t1 => match l2 with
    | nil => (* ... *)
    | h2 :: t2 => (* ... *)
    end
  end.
```

bag은 다중집합multiset을 뜻하는 용어이다. 자연수의 bag은 단순히 natlist로 정의하면 된다. 특정한 원소의 개수를 bag에서 세는 함수를 만들어 보자.

```
1  Definition bag := natlist.
2
3  Fixpoint count (v: nat) (s: bag) : nat :=
4    match s with
5    | nil => 0
6    | h :: t => match (eqb v h) with
7      | true => 1 + (count v t)
8      | false => count v t
9      end
10   end.
11
12 Compute count 1 [1;2;3;1;4;1]. (* = 3: nat *)
13 Compute count 6 [1;2;3;1;4;1]. (* = 0: nat *)
```

Bag에서 값이 일치하는 첫 원소 하나를 삭제하는 함수 remove_one, 값이 일치하는 모든 원소를 삭제하는 함수 remove_all, 그리고 두 bag 간의 포함관계를 계산하는 부울값 함수 included를 만들어 보자. 이들은 연습문제이므로 완성된 답이 아닌 힌트만 보였다.

```
Fixpoint remove_one (v:nat) (s:natlist) : natlist :=
  match s with
  | nil => nil
  | h :: t => match (eqb v h) with
    | true => (* ... *)
    | false => (* ... *)
    end
  end.

Fixpoint remove_all (v:nat) (s:natlist) : natlist :=
  match s with
  | nil => nil
  | h :: t => match (eqb v h) with
    | true => (* ... *)
    | false => (* ... *)
    end
  end.

Fixpoint included (s1 s2:natlist) : bool :=
  match s1 with
  | nil => true
  | h :: t => match (count h s2) with
    | 0 => (* ... *)
    | S n => (* ... *)
    end
  end.
```

bag에 원소 하나를 더하면 그 원소의 count가 하나 증가한다는 (당연한) 사실을 정리로 나타내고 증명해 보자. (이 문제는 난이도가 별 2개로 되어 있어 그리 높지는 않지만 Coq에 익숙하지 않은 사람에게는 많이 어려울 수 있다고 본다. 왜냐하면 지금까지 사용한 적이 별로 없는 익숙하지 않은 기법이 필요하기 때문이다.)

```
1 Theorem add_inc_count : forall (v: nat) (s: bag),
2   count v (v :: s) = 1 + (count v s).
3 Proof.
4   intros v s.
5   simpl.
6   (* ... *)
```

라인 4까지만 실행했을 때 다음과 같은 고울이 나타나게 된다.

$$\texttt{count n (add n s) = S (count n s)} \tag{4.2}$$

count와 add의 정의를 보면 이것의 증명을 위하여 구태여 induction이나 destruct를 사용할 필요가 없다는 것을 알게 된다.

라인 5를 실행하면, simpl은 함수의 정의에 따라 결과를 보여줄 뿐이므로 고울 프랍은 다음과 같이 match를 포함한 표현이 된다. 익숙하지 않은 기법이 필요하다는 것은 이를 두고 말한 것이다. 이렇게 match가 증명 스크립트가 아닌 고울 프랍에 나타나는 경우는 이 책에서는 두 번째이다. 이 책 맨 뒤의 찾아보기index에서 'match appearing in goals'를 찾아 보라.

또한 여기서 match의 인수는 변수가 아니라 2항 연산자 =?를 포함하고 있는 표현항term인데 이것도 앞서 한두 번 나온 적은 있지만 아직까지는 그리 익숙하지 않은 기법이다.

```
match n =? n with
  | true => S (count n s)
  | false => count n s
  end = S (count n s)
```

n =? n은 true이므로 위 식의 좌변은 S (count n s)이 된다. 이것은 우변과 동일하므로 여기서 reflexivity를 쓸 수 있으면 좋겠지만, Coq의 reflexivity는 n =? n의 값이 true라는 것은 알지 못한다. 왜냐하면 n =? n의 값이 true라는 것은 정의가 아니라 정리이기 때문이다.[2]

(4.2)의 좌변을 S (count n s)로 대체하려면 전에 증명해 두었던 다음의 정리를 써야 한다.

```
Theorem eqb_refl : forall n : nat,
  (n =? n) = true.
```

rewrite 책략을 쓰면 되겠다. 이하 생략. ✓

Reasoning About Lists

이제부터 리스트 변수에 대한 destruct와 induction을 사용하여 리스트에 대한 증명을 본격적으로 연습하도록 하자.

간단한 정리로부터 시작한다. 증명을 진행하면서 Coq Goals 화면의 변화를 관찰하면 쉽게 이해가 될 것이다.

```
Theorem tl_length_pred : forall l : natlist,
  pred (length l) = length (tl l).
Proof.
```

[2] simpl은 inductive definition의 정의만 사용하여 계산한다. reflexivity는 inductive definition뿐만 아니라 inductively 정의된 함수의 합성composition도 계산에 이용할 수 있다는 점에서 simpl보다는 강력하다. 하지만 reflexivity도 정리는 이용하지 못한다.

```
intros l. destruct l as [| n l'].
- (* l = nil *) simpl. reflexivity.
- (* l = n :: l' *) simpl. reflexivity. Qed.
```

destruct l를 사용하는 이유는, 이 정리가 증명하려는 프랍에 나타나는 두 함수 length l과 tl l은 리스트를 인수로 하며, 이 함수들은 모두 ① 인수 l이 nil인 경우와, ② l이 n :: l'인 두 경우로 나누어 정의되었기 때문이다. 그러므로 각 경우에 대하여 증명을 따로 하는 것이 당연하다.

destruct 책략으로 증명하지 못하는 정리의 예로 app의 결합법칙을 증명해 보자.

```
1 Theorem app_assoc : forall l1 l2 l3 : natlist,
2   (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).
3 Proof.
4   intros l1 l2 l3. induction l1 as [| n l1' IHl1'].
5   - (* l1 = nil *) simpl. reflexivity.
6   - (* l1 = n :: l1' *) simpl.
7     rewrite -> IHl1'.
8     reflexivity. Qed.
```

리스트에 대한 귀납은 이번이 처음이므로 라인 4의 induction intro pattern as [| n l1' IHl1']에 대해서 설명해 보겠다. natlist의 정의에서 첫번째 생성자 nil은 상수이므로 | 왼쪽은 비워 놓았다. 두번째 생성자 cons는 2항 생성자인데 (n: nat) (l: natlist) 형태의 인수를 가지므로 | 오른쪽에는 두 변수 n l1'을 넣어 두었다. l1에 대한 귀납이므로, 즉 귀납단계에서 l1 = n :: l1'으로 놓고 증명하는 것이므로, l1에는 '를 붙여서 intro pattern에 l1'을 사용한다.

그 다음에는 귀납가설 [IHl1']을 인트로 패턴에 넣었다. 귀납가설이라 함은 l1의 전 단계, 즉 l1'에 대해서는 고울 프랍이 성립한다는 것을 가정하는 것이므로 다음과 같은 것이 컨텍스트에 들어간다.

```
IHl1' : (l1' ++ l2) ++ l3 = l1' ++ l2 ++ l3
```

고울 프랍에서는 l1을 n :: l1'로 치환해 주어야 한다.

```
1 ((n :: l1') ++ l2) ++ l3 = (n :: l1') ++ l2 ++ l3
```

이것이 라인 6의 simpl 직전의 고울 프랍이다. 증명이 진행됨에 따른 고울 프랍의 변화 과정을 세밀하게 들여다 보면 다음과 같다.

```
2 n :: (l1' ++ l2) ++ l3 = n :: l1' ++ l2 ++ l3 (* after simpl in line 6 *)
3 n :: l1' ++ l2 ++ l3 = n :: l1' ++ l2 ++ l3 (* after rewrite IHl1' in line 7 *)
```

노파심에서 말하겠는데 이제부터의 설명은, 증명 스크립트가 짧은 것을 고려하면, 생각보다 길고 상세할 것이다. 무릇 어떤 학문 분야에서든 배움과 깨달음이 있으려면 최소 이 정도의 끈기는 있어야 한다.

simpl에 의하여 라인 1에 보인 고울 프랍이 라인 2로 바뀌는 이유는 ++의 정의가 다음과 같았기 때문이다.

```
++defn: (h :: t) ++ l2 => h :: (t ++ l2)
```

h :: (t ++ l2)는 괄호를 제거하여 h :: t ++ l2로 써도 혼동의 여지가 없으므로 앞으로는 항상 이렇게 쓸 것이다.

이 (h :: t) ++ l2 패턴이 라인 1의 좌변에도 나타나고, 또한 우변에도 나타나고 있다. 다음은 라인 1의 일부에 색깔을 넣어 이 패턴을 보여준 것이다.

```
4 ((n :: l1') ++ l2) ++ l3 = (n :: l1') ++ (l2 ++ l3) (* same as line 1 *)
```

라인 4의 우변의 부분 표현subexpression (l2 ++ l3)에, 라인 1에서는 없던 괄호를 두른 이유는 패턴을 쉽게 볼 수 있게 하기 위함이다. ++의 오른쪽 결합에 의하여 이렇게 해도 된다.

라인 4에 simpl을 적용하면, 좌변에는 ++defn을 두 번 적용하고 우변에는 ++defn을 한 번 적용하여 라인 2를 얻을 수 있다.[3]

이제 IHl1'을 인수로 하여 rewrite를 실행하면 라인 3을 얻게 된다.

그 다음은 아주 쉬우므로 생략한다. ✓

리스트를 뒤집는reverse 함수를 다음과 같이 정의할 수 있다.

```
Fixpoint rev (l:natlist) : natlist :=
  match l with
  | nil    => nil
  | h :: t => rev t ++ [h]
  end.
```

Unit test를 위한 예제를 2개만 들어 보았다.

```
Example test_rev1 : rev [1;2;3] = [3;2;1].
Proof. reflexivity.  Qed.
Example test_rev2 : rev nil = nil.
Proof. reflexivity.  Qed.
```

뒤집은 리스트의 길이는 원래 리스트의 길이와 같다는 것을 증명해 보자.

```
1  Theorem rev_length_firsttry : forall l : natlist,
2    length (rev l) = length l.
3  Proof.
4    intros l. induction l as [| n l' IHl'].
5    - (* l = nil *)
6      reflexivity.
7    - (* l = n :: l' *)
8      simpl.
9      rewrite <- IHl'.
10 Abort.
```

잘 안 된다. 라인 8의 simpl을 실행한 후의 Coq Goals 화면은 다음과 같다.

```
n : nat
l' : natlist
IHl' : length (rev l') = length l'
(1 / 1) ------------------------
length (rev l' ++ [n]) = S (length l')
```

[3] simpl은 이렇게 inductive definition에 의한 환원을 최대한 여러 번 실행한다는 점에서 rewrite와 구별된다. rewrite는 (가설이나 정리를 인수로 하여 고울 프랍에) 한 번 적용할 때마다 환원이 딱 1번 일어난다. 따라서 만약 rewrite에 의한 환원이 n 번 필요하다면 rewrite 책략을 n번 실행해야 한다. 이때 *outermost-leftmost* 순서로 환원이 일어난다.

고울 프랍의 좌변에는 귀납가설이 아무런 영향을 주지 못하고 있다. 고울 프랍 좌변을 보면

```
length(rev l' ++ [n]) = length (rev l') + 1
```

을 증명할 수 있다면 좋을 것이 확실하다. 그러니 이걸 먼저 증명하기로 하자. 그런데 하는 김에 조금 더 일반적인 다음의 정리를 증명하는 것이 좋아 보인다.

```
Theorem app_length : forall l1 l2 : natlist,
  length (l1 ++ l2) = (length l1) + (length l2).
Proof.
  intros l1 l2. induction l1 as [| n l1' IHl1']
  - (* l1 = nil *)
    reflexivity.
  - (* l1 = n :: l1' *)
    simpl. rewrite -> IHl1'. reflexivity.  Qed.
```

위의 증명에서 simpl이 여러 개의 환원을 한꺼번에 하고 있는데, 정확히 어떤 환원들이 실행되었는지 설명해 보라. 또한 reflexivity는 어떤 환원(들)을 하는지도 알아보라.

이제 원래 정리를 증명해 보자.

```
Theorem rev_length : forall l : natlist,
  length (rev l) = length l.
Proof.
  intros l. induction l as [| n l' IHl'].
  - (* l = nil *)
    reflexivity.
  - (* l = n :: l' *)
    simpl. rewrite -> app_length.
    simpl. rewrite -> IHl'. rewrite add_comm.
    reflexivity.
Qed.
```

이 증명 과정을 즐겼기를 바란다.

Search

이전에 나왔던 정리를 사용하고자 하는데 이름이 기억나지 않을 때 Search 명령을 사용하면 좋다. 예를 들어 rev가 나타나는 정리들을 찾고자 한다면 다음과 같이 하면 된다.

```
Search rev.
```

Search app의 결과는 프랍에 app뿐만 아니라 ++가 사용된 모든 정리를 찾아준다.
"+"가 나타나는 정리는 대단히 많으므로 특정한 파일 내에서 찾으려면 다음과 같이 한다.

```
Search "+" inside Basics.
```

덧셈의 교환법칙을 나타내는 정리를 찾고자 한다면 다음과 같이 하면 된다.

```
Search (_ + _ = _ + _).
(* or better yet, *)
Search (?x + ?y = ?y + ?x).
```

Search와 비슷한 것으로 Locate가 있다. 전자는 정리를 찾는 데 사용되고 후자는 표기법notation을 찾는 데 사용된다. 기호를 괄호로 둘러서 사용한다.

```
Locate "<=". (* some results *)
Locate "_ <= _". (* just one result *)
```

Exercises

① 공리스트empty list는 ++ 연산에 대한 좌항등원left identity이다. 즉, 모든 리스트 l: natlist에 대해서 [ ] ++ l = l이며, 이는 ++의 정의에 의해서 성립한다. [ ]가 ++에 대한 우항등원right identity이기도 함을 증명해 보자.

```
1 Theorem app_nil_r : forall l : natlist,
2   l ++ [] = l.
3 Proof.
4   intros l. induction l as [| n l' IHl'].
5   - (* l = [ ] *) (* .. *)
6   - (* l = n :: l' *) (* use IHl' *) Qed.
```

② rev는 덧셈에 대하여 분배법칙이 성립한다.

```
1 Theorem rev_app_distr: forall l1 l2 : natlist,
2   rev (l1 ++ l2) = rev l2 ++ rev l1.
3 Proof.
4   intros. induction l1 as [| n l1' IHl1'].
5   - (* l1 = [ ] *) (* use app_nil_r *)
6   - (* l1 = n :: l1' *) (* use IHl1' and app_assoc *) Qed.
```

③ rev는 self-inverse property를 가진다. 즉 *involutive*하다.

```
1 Theorem rev_involutive : forall l : natlist,
2   rev (rev l) = l.
3 Proof.
4   intros. induction l as [| n l' IHl'].
5   - (* l = [ ] *) (* ... *)
6   - (* l = n :: l' *) (* use rev_app_distr and IHl' *) Qed.
```

④ 주어진 리스트의 nonzero 원소들만 추출하여 이루어진 리스트를 만드는 함수를 nonzeros라고 했을 때, 두 리스트의 합의 nonzeros는 각각의 nonzeros를 구한 다음에 그 결과들을 합한 것과 같음을 증명해 보자.

```
1  Fixpoint nonzeros (l:natlist) : natlist :=
2    match l with
3    | nil => []
4    | h :: t => match h with
5      | O => nonzeros t
6      | S h' => h :: nonzeros t
7      end
8    end.
9
10 Lemma nonzeros_app : forall l1 l2 : natlist,
11   nonzeros (l1 ++ l2) = (nonzeros l1) ++ (nonzeros l2).
```

```
12 Proof.
13   intros. induction l1 as [| n l1' IHl1'].
14   - (* l1 = [ ] *) simpl. reflexivity.
15   - (* l1 = n :: l1' *) destruct n as [| n'] eqn:Eq.
16     + (* n = 0 *) (* use rewrite IHl1' *)
17     + (* n = S n' *) (* use rewrite IHl1' *)
```

라인 15의 destruct 앞에 simpl을 넣어도 된다. 하지만 오히려 더 복잡해지므로 증명에 도움이 되지 않는다.

⑤ 두 리스트가 동일하면(길이가 같도 대응하는 위치의 원소들끼리 모두 같으면) true, 아니면 false를 리턴하는 함수 eqblist를 만들어라.

```
1   Fixpoint eqblist (l1 l2 : natlist) : bool :=
2   match l1, l2 with
3   | nil, nil => true
4   | nil, _ => false
5   | _, nil => false
6   | h1 :: t1, h2 :: t2 => match (* ... *) with
7     | true => (* ... *)
8     | false => (* ... *)
9     end
10  end.
```

⑥ eqblist는 반사적reflexive임을, 즉 모든 리스트는 자기 자신과 동일함을 증명하여라.

```
1 Theorem eqblist_refl : forall l:natlist,
2   true = eqblist l l.
3 Proof.
4   intros. induction l as [| n l' IHl'].
5   - (* l = [ ] *) (* ... *)
6   - (* l = n :: l' *) (* use eqb_refl *)
7 Qed.
```

⑦ 다음은 $\forall n \in \mathbb{N},\ n \leqslant n + 1$의 증명이다. 정리의 프랍을 보면 내용적으로는 부등식이지만 형식적으로는 등식이다. 증명은 귀납을 사용하면 쉽게 할 수 있다.

```
1 Theorem leb_n_Sn : forall n,
2   n <=? (S n) = true.
3 Proof.
4   (* ... *) Qed.
```

⑧ 다음의 증명에서는 방금 보인 leb_n_Sn을 보조정리로 사용한다.

```
1 Theorem remove_does_not_increase_count : forall (s : bag),
2   (count 0 (remove_one 0 s)) <=? (count 0 s) = true.
3 Proof.
4   intros. induction s as [|h t IHt].
5   - (* s = [ ] *) simpl. reflexivity.
6   - (* s = h :: t *) destruct h as [| h'] eqn:Eq.
7     + (* use leb_n_Sn *)
8     + (* use IHt *)
```

⑨ 한 문제만 더 풀어보자.

```
1 Theorem bag_count_sum: forall (s1 s2: bag),
2   count 0 (sum s1 s2) = (count 0 s1) + (count 0 s2).
```

(힌트). s1에 대한 귀납을 사용한다. 귀납단계에서 s1 = n :: s1'으로 두었을 때 destruct n을 사용한다. ✓

The apply tactic on universal propositions

다음 문제는 함수가 involutive이면 injective임을 증명하는 것이다. 라인 13에는 처음 보는 책략인 symmetry가 사용되었다. 그리고 이 라인에서 apply도 사용되었다. 이 책략은 전에 사용한 적은 있지만 설명을 거의 하지 않았으며 이 문제에서 제대로 설명하고자 한다.

```
1 Theorem involution_injective : forall (f : nat -> nat),
2     (forall n : nat, n = f (f n)) ->
3       (forall n1 n2 : nat, f n1 = f n2 -> n1 = n2).
4 Proof.
5   intros f H. intros n1 n2.
6   intros Heq.
7   replace (n1) with (f (f n1)).
8   replace (n2) with (f (f n2)).
9   { rewrite Heq. reflexivity. }
10  { replace (f (f n2)) with n2.
11    - reflexivity.
12    - rewrite <- H. reflexivity. }
13  { symmetry. apply H with (n:= n1). }
14 Qed.
```

라인 6을 실행한 후의 Coq Goals 화면은 다음과 같다.

```
15 f : nat -> nat
16 H : forall n : nat, n = f (f n)
17 n1, n2 : nat
18 Heq : f n1 = f n2
19 (1 / 1) ------------------------
20 n1 = n2
```

이제부터 할 증명의 아이디어는 다음과 같다. 고울 프랍 n1 = n2를 증명하기 위하여 가설 H를 이용하면 f (f n1) = f (f n2)를 보이면 충분할 것이다. 왜냐하면 가설 Heq: f n1 = f n2가 있기 때문이다. 그래서 라인 7과 라인 8을 실행하면 다음과 같은 3 개의 고울들이 나타난다. Goal 1은 원래의 고울, Goal 2는 라인 8을 위한 것, Goal 3은 라인 7을 위한 것이다.

```
21  Goal 1
22  f : nat -> nat
23  H : forall n : nat, n = f (f n)
24  n1, n2 : nat
25  Heq : f n1 = f n2
26  (1 / 3) ------------------------
27  f (f n1) = f (f n2)
28
29  Goal 2
```

```
30    f : nat -> nat
31    H : forall n : nat, n = f (f n)
32    n1, n2 : nat
33    Heq : f n1 = f n2
34    (2 / 3) ------------------------
35    f (f n2) = n2
36
37    Goal 3
38    f : nat -> nat
39    H : forall n : nat, n = f (f n)
40    n1, n2 : nat
41    Heq : f n1 = f n2
42    (3 / 3) ------------------------
43    f (f n1) = n1
```

Goal 1은 라인 9로써 아주 쉽게 증명된다. ✓

Goal 2의 증명은 10–12, Goal 3의 증명은 라인 13이다. 거의 같은 형태의 고울이지만 증명을 사못 다르게 써 보았다.

Goal 2는 라인 10에 의하여 다시 2개의 subgoal로 나뉜다. 첫 번째 subgoal은 n2 = n2이므로 간단하게 라인 11의 reflexivity로 증명된다. 두 번째 subgoal은 n2 = f (f n2)이므로 라인 11의 rewrite <- H와 n := n2의 자동 치환에 의하여 n2 = n2로 바뀐 후, reflexivity에 의하여 증명된다. ✓

Goal 3은 Goal 2와 거의 동일한 내용인데, 이번에는 라인 13의 symmetry와 apply를 사용하여 이전보다 훨씬 더 간단하게 처리하였다.

라인 13의 symmetry는 고울 프랍(Goal 3 에서의)의 좌변과 우변을 맞바꾼다. 그러면 이제 고울 프랍은 가설 H의 바디body n = f (f n)과 똑같은 형태가 된다. 다만 H의 묶인 변수 n은 고울에 나타난 변수 n1으로 치환substitute되어야 한다.

라인 13의 apply H with (n := n1).은 다음의 셋 중 어느 것으로도 대신할 수 있다.

```
44  apply (H n1).
45  apply H.
46  exact H (n n1).
```

exact H는 컨텍스트의 가설 H가 고울 프랍과 정확히 일치할 때 사용하는 책략이다. 그런데 H가 전칭문이고 고울 프랍이 이 전칭문의 instance이면, 그리고 이때 필요한 치환이 n := n1이라면 라인 46과 같은 구문을 써 주면 된다. 전칭문인 H를 apply의 인수로 사용하였기에 이 하위절의 제목을 apply tactic on universal propositions라고 한 것이다.

apply는 exact보다 더 일반적인 책략이다. 우선 라인 3과 동일한 구문인 라인 44를 써도 되고, 아니면 라인 13에서처럼 좀 더 읽기 편한 apply H with (n := n1) 구문을 써도 된다. 그리고 필요한 치환을 알아서 자동으로 찾아주는 구문인 라인 45를 써도 된다.

apply는 가설이 함의문(또는 함의문의 전칭한정문)이고 고울 프랍이 이 함의문의 후건일 때도 사용할 수 있는데 이러한 예는 나중에 보게 될 것이다. ✓

단평 4.2 apply 책략은, rewrite 책략과 마찬가지로 가설뿐만 아니라 정리를 인수로 삼아서 사용할 수 있다. 앞으로 이런 예들은 많이 보게 될 것이다. ⊣

단평 4.3 지금까지 다룬 모든 고울 프랍은 등식, 혹은 등식의 전칭한정문이었다. 따라서 모든 증명은 reflexivity 책략으로 마무리 할 수 있으며 apply나 exact를 써서 더 간편하게 처리할 수 있었다.

등식이나 등식의 전칭한정문이 아닌 고울 프랍을 증명할 때는 reflexivity는 사용할 수 없으며 apply나 exact 등 다른 책략을 사용해야 한다. ⊣

다음은 rev가 injective 임을 보이는 문제이다. 우리는 이전에 rev가 involutive임을 보였다. 이것을 이용하면 쉽게 증명할 수 있다.

```
Theorem rev_injective : forall (l1 l2 : natlist),
  rev l1 = rev l2 -> l1 = l2.
Proof.
  intros l1 l2 H.
  (* use rev_involutive *)
Qed.
```

Options

전에 언급은 했지만 설명이 별로 없었던 함수 hd: nat -> natlist -> nat를 다시 들여다 보자.

```
1  Definition hd (default: nat) (l: natlist) : nat :=
2    match l with
3    | nil => default
4    | h :: t => h
5    end.
6
7  Compute hd 99 [7; 0; 4]. (* = 7 *)
8  Compute hd 99 []. (* = 99 *)
```

인수 defalut: nat는 오로지 인수 l이 nil일 때 오류가 나는 것을 방지하기 위한 것이다. 이런 식으로 예외exception를 처리하는 것은 좋지 않다 왜냐하면 예를 들어 l = [99, 17, ...]인 경우에도 리턴값은 99가 되어서 l이 nil일 때와 구별이 가지 않기 때문이다.

이 문제는 함수의 리턴 타입을 다음과 같이 바꿔줌으로써 해결할 수 있다.

```
1  Inductive natoption : Type :=
2    | Some (n : nat)
3    | None.
4
5  Fixpoint hd' (l: natlist) : natoption :=
6    match l with
7    | nil => None
8    | h :: t => Some h
9    end.
10
11 Definition option_elim (default: nat) (o: natoption): nat :=
12   match o with
13   | Some n' => n'
14   | None => default
15   end.
```

hd'의 결과가 None이라면 예외가 발생한 것을 알고 적당히 처리하면 되고, 그렇지 않으면 정상적으로 실행된 것이므로 hd'의 리턴값을 option_elim에 주어 원래의 원하던 값을 얻을 수 있다.

Option을 이용하면 주어진 리스트의 n번째 원소를 찾아 리턴하는 함수를, nil이 입력되었을 때 오류 없이 잘 처리하도록 만들 수 있다. hd'은 이러한 함수를, $n = 0$일 때의 특별한 경우에 대해서 구현한 것으로 볼 수 있다.

4.3 Partial Maps

*Partial map*은 다른 언어에서 흔히 *dictionary* 혹은 *associative array*라고 부르는 데이터 타입이다.[4]

Partial map은 (id, value)쌍들의 집합을 다루기 위한 도구라고 보면 된다.[5]

Partial map은 기본적으로 *update*와 *find* 함수를 가지고 있어야 한다. 전자는 기존의 partial map에 새로운 (id, value) 쌍을 추가하는 함수이다. 기존에 (id, value')이 들어 있었다면 이것은 (id, value)로 대체되어야 할 것인데, 우리는 (일단 여기서는) 효율성을 무시하고 이를 아주 간단하게 구현할 예정이다. 후자는 partial map에서 id에 대응되는 value를 찾아주는 함수이다.

Partial map에 id가 주어졌을 때 그것에 대응하는 value가 없을 수 있다. 이런 경우에 대한 원만한 처리를 위하여 option을 사용해야 한다. 즉, partial map의 value의 타입은, 정확히 말하면 find 함수의 리턴값의 타입은 natoption이어야 한다.

value의 타입은 일단 nat인 것으로 하자. 나중에 이를 확장하는 방법을 배울 것이다. 그리고 id는 다음과 같이 정의하여 사용하기로 한다.

```
Inductive id : Type :=
  | Id (n : nat).

Definition eqb_id (x1 x2: id) : bool :=
  match x1, x2 with
  | Id n1, Id n2 => n1 =? n2
  end.
```

간단한 문제 하나를 풀어 보자.

```
1 Theorem eqb_id_refl : forall x,
2   eqb_id x x = true.
3 Proof.
4   intros x.
5   destruct x as [m].
6   simpl. rewrite -> eqb_refl. reflexivity. Qed.
```

라인 5의 destruct x를 보자. destruct는 원래 case analysis에 사용하는 것이라고 알고 있었지만, 생성자가 1개뿐인 경우에도 이렇게 응용된다. destruct x as [m] 없이는 그 다음의 simpl은 아무런 계산도 하지 못한다.

이제 partial map과 update, find를 차례로 정의한다.

① 먼저 partial map의 정의이다.

[4]어떤 사람들은 *map*, *symbol table* 등의 용어를 쓰기도 한다.

[5]통상 id 대신 key라는 용어를 많이 사용한다.

```
Inductive partial_map : Type :=
  | empty
  | record (i: id) (v: nat) (m: partial_map).
```

연습문제 4.4 Partial map의 예를 몇 개 들어 보라. (힌트). 아주 쉽다. ⊣

② Partial map을 업데이트 하는 함수는 다음과 같이 정의한다.

```
Definition update (d: partial_map)
                  (x: id) (value: nat)
                  : partial_map :=
  record x value d.
```

update는 기존의 partial map에 (id, value)가 존재하고, (id', value')을 입력했을 때, id = id'인 경우 value를 value'으로 업뎃하는 것이 아니라, (id, value)는 놓아둔 채로 (id, value')을 그냥 push한다. 어차피 find할 때는 최근 것을 리턴하도록 구현할 것이므로 큰 문제는 없다.

연습문제 4.5 update를 이용하여 partial map을 몇 개 만들어 보라. ⊣

③ 마지막으로 find를 정의한다.

```
Fixpoint find (x: id) (d: partial_map) : natoption :=
  match d with
  | empty => None
  | record y v d' => if eqb_id x y
                     then Some v
                     else find x d'
  end.
```

간단한 두 문제를 풀어 보자.

```
1  Theorem update_eq :
2    forall (d : partial_map) (x : id) (v: nat),
3      find x (update d x v) = Some v.
4  Proof. intros d x v.
5    simpl.
6    rewrite -> eqb_id_refl.
7    reflexivity. Qed.
8
9  Theorem update_neq :
10   forall (d : partial_map) (x y : id) (o: nat),
11     eqb_id x y = false -> find x (update d y o) = find x d.
12 Proof. intros d x y o. intros H.
13   simpl.
14   rewrite -> H. reflexivity. Qed.
```

라인 5에 있는 simpl의 동작을 세밀하게 분석해 보자. 현재의 고울은 다음과 같다.

```
find x (update d x v) = Some v
```

update의 정의에 의하여

```
find x (record x v d) = Some v
```

그 다음은 `find`의 정의에 의하여

```
(if eqb_id x x then Some v else find x d) = Some v
```

가 된다. 그 다음 라인 6, 7은 설명할 필요가 없을 것이다.

라인 13의 `simpl`도 비슷하게 분석할 수 있다. 라인 14는 쉽다. ✓

5 Poly

5.1 Polymorhpism

Polymorphic Types

지금까지 우리가 다루었던 리스트는 모두 정수들의 리스트였다. 다른 타입의 원소들로 이루어진 리스트, 예를 들어 부울리언 리스트를 다루려면 natlist에서 했던 작업을 반복하면 된다.

그런데 문제는 boollist: Type만 정의한다고 해서 끝나는 일이 아니고, natlist에 대해 정의하고 증명했던 온갖 함수와 정리들에 대해서도 똑같은 작업을 되풀이 해야 한다는 것이다. 이렇게 중복 작업을 하는 것은 매우 비효율적이다. 우리는 이 문제를 해결하기 위하여 다형성(*polymorphism*)이라는 개념을 도입한다.

```
Inductive list (X: Type) : Type :=
  | nil
  | cons (x : X) (l : list X).
```

다형 타입(*polymorphic type*)인 list는 함수로 볼 수 있다.[1] X라는 <u>타입을 인수로</u> 받아 list X라는 <u>타입을 리턴</u>하는 함수 말이다. 통상적인 함수들은 특정 <u>타입의 원소를 인수로</u> 받아 특정 <u>타입의 원소를 리턴</u>하므로 list와 구별된다.[2]

```
Check list. (* list: Type -> Type *)
Check list nat. (* list nat: Set *)
```

list의 생성자 nil은 nullary인 것으로 보이지만 실제로는 1개의 타입 인수를 요구한다. 생성자 cons는 binary인 것으로 보이지만 실제로는 1개의 타입 인수와 2개의 값 인수를 요구한다. 이들은 다형 생성자(*polymorphic constructor*)이다.

```
Check nil nat. (* nil nat: list nat *)
Check nil bool. (* nil bool: list bool *)
```

여기서 정말로 흥미있는 것은 Check nil.의 결과이다. 이것은 뒤에 어떤 타입 X를 붙여서 list X 형태가 되어야 의미가 있다. 그렇다고 Check nil X., 혹은 Check nil X: list X.를 실행하면 에

---

[1] 다형 타입은 Type -> Type, Type -> Type -> Type, Type -> Type -> Type -> Type 등의 원소를 뜻한다.

[2] 특정 타입의 원소를 인수로 받아 타입을 리턴하는 경우도 있다. 이를 *dependent type*이라고 한다.

러가 난다. 왜냐하면 X가 정의되어 있지 않은 상태에서 이를 표현에 사용할 수는 없기 때문이다. 뜸 들이지 않고 답을 알아 보자.

```
Check nil. (* nil: forall X : Type, list X *)
```

이런 것은 일찌기 본 적이 없다. 수학에서 다루어 본 적이 없는 물건이며 집합이 아닌 타입으로서 최초의 예이다.

X는 타입변수이다. 실제의 타입을 정의하기 전에는 그냥 막연히 타입 X의 리스트는 단 한 개도 만들 수 없다. 다른 생성자 cons도 이 점에서 비슷하다.

```
Check cons. (* cons: forall X : Type, X -> list X -> list X *)
Check cons nat. (* cons nat: nat -> list nat -> list nat *)
Check cons nat 3. (* cons nat 3: list nat -> list nat *)
Check cons nat 3 (nil nat). (* cons nat 3 (nil nat): list nat *)
Check (cons nat 2 (cons nat 1 (nil nat))). (* ..: list nat *)
```

지금까지 보아온 타입에는 다음과 같은 유형이 있다.

```
A : Set
P : Prop
f : T -> S (* T, S : Set *)
forall X : Type, pol_type X (* pol_type is a polymorphic type *)
```

나중에 다음과 같은 유형의 타입도 보게 될 것이다.

```
forall x : X, P x : Prop (* P : X -> Prop *)
```

다형 버전 repeat 함수를 만들어 보자.

```
Fixpoint repeat (X: Type) (x: X) (count: nat) : list X :=
  match count with
  | 0 => nil X
  | S count' => cons X x (repeat X x count')
  end.

Example test_repeat1 :
  repeat nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).
Proof. simpl. reflexivity. Qed.

Example test_repeat2 :
  repeat bool false 1 = cons bool false (nil bool).
Proof. simpl. reflexivity. Qed.
```

다음은 아무 쓸모는 없지만 다형성의 개념을 익히기에 좋은 예이다.

```
Inductive mumble : Type :=
  | a
  | b (x: mumble) (y: nat)
  | c.

Inductive grumble (X: Type) : Type :=
  | d (m: mumble)
  | e (x: X).
  | f (* added by me *)
```

mumble은 특별할 것 없는 평범한 타입이다. a와 c는 mumble 타입에 속하는 원소, 혹은 상수 생성자이다. b는 2개의 인수(하나는 mumble 타입, 다른 하나는 nat 타입)를 받아서 mumble 타입 원소를 리턴하는 2항 생성자이다.

```
Check mumble. (* mumble: Set *)
Check a. (* a: mumble *)
Check b. (* b: mumble -> nat -> mumble *)
Check b a 5. (* b a 5: mumble *)
```

다형 타입인 grumble도 처음엔 쉽다.

```
Check grumble. (* grumble: Type -> Type *)
Check grumble nat. (* grumble nat: Set *)
```

grumble nat의 원소를 만들려면 어떻게 해야 하는가? 당연히 grumble의 생성자를 사용해야 한다. 그런데 예를 들어 d a는 안 된다. 정의에 보면 d (m: mumble)이라고 되어 있으므로 m 자리에 a를 넣으면 될 것 같지만 안 된다. 이건 마치 다형 타입 list의 nil 생성자가 상수$_{\text{nullary}}$이지만 nil nat 등으로 어떤 타입을 붙여 주어야 하며, cons 생성자는 2항$_{\text{binary}}$이지만 cons nat 3 (nil nat)과 같이 써 주어야 하는 것과 같다.

즉 다형 타입의 생성자는 무조건 바로 뒤에 타입, 즉 Type의 원소를 사용해야 하며 타입의 원소를 쓰면 안 된다.

이것만 확실하게 알면 이제 술술 풀린다. 아까의 문제로 돌아와서 grumble nat의 원소를 만들려면 grumble의 생성자를 사용하면 된다. 3 종류의 생성자 모두 쓸 수 있다.

```
Check d nat a. (* d nat a: grumble nat *)
Check e nat 3. (* e nat 3: grumble nat *)
Check f nat. (* f nat: grumble nat *)
```

SF에 나와 있는 예들을 다음에 보였다.

```
1 Fail Check d (b a 5).
2 Check d mumble (b a 5). (* grumble mumble *)
3 Check d bool (b a 5). (* grumble bool *)
4 Check e bool true. (* grumble bool *)
5 Check e mumble (b c 0). (* grumble mumble *)
6 Fail Check e bool (b c 0).
7 Check c. (* mumble *)
```

라인 1과 6의 Fail은 에러가 나도 그냥 넘어가라는 뜻이다. 라인 1은 d의 첫번째 인수가 타입이어야 하는데 원소가 있어서 문제가 된 것이다. 라인 6은 bool 뒤에 부울리언 타입 원소가 와야 하는데 다른 타입이 있어서 문제가 되었다.

Type Annotation Inference

다형 repeat의 정의는 다음과 같이 시작한다.

```
Fixpoint repeat (X: Type) (x: X) (count: nat) : list X :=
```

그런데 이것을 다음과 바꿔도 된다. 똑같이 동작한다.

```
Fixpoint repeat X x count : list X :=
```

왜냐하면 Coq이 repeat의 인수 X, x 및 count의 타입을 모두 알아서 추론해 주기 때문이다. 이것을 타입 추론(*type inference*)이라고 한다. 그렇다고 항상 타입 지정을 생략하라는 것은 아니다. 인수들의 타입을 지정함으로써 코드의 가독성이 좋아지고 실수의 위험을 줄일 수 있으므로 그러하다.

Type Argument Synthesis

다형 함수의 인수에서 타입 지정을 생략할 수 있었는데, 다형 타입의 생성자의 인수에서는 타입 인수를 간략하게 밑줄underscore로 대체할 수 있다. 다음은 repeat 함수의 몸체body에서 사용한 생성자 nil과 cons에 대한 예이다. 생성자 뿐만 아니라 함수 몸체 내의 함수의 인수에서도 마찬가지로 할 수 있다.

```
Fixpoint repeat'' X x count : list X :=
  match count with
  | 0        => nil _
  | S count' => cons _ x (repeat'' _ x count')
  end.
```

다음과 같은 경우에는, 그다지 크게 도움되는 것 같지는 않지만, nat 대신 _를 쓰면 되므로 조금은 편리한 것으로 보인다.

```
Definition list123' :=
  cons _ 1 (cons _ 2 (cons _ 3 (nil _))).
```

Implicit Arguments

다형성을 편리하게 사용하기 위한 도구 중에서 이것이 가장 좋아 보인다 — 이건 밑줄 _ 마저도 생략할 수 있게 해 준다. 다음과 같이 Arguments 명령을 사용하면 된다.

```
Arguments nil {X}.
Arguments cons {X}.
Arguments repeat {X}.

Definition list123'' :=
  cons 1 (cons 2 (cons 3 nil)).
```

또는 위와 같은 별도의 Arguments 명령 대신 함수를 정의할 때 다음과 같이 타입 인수에 소괄호쌍 대신 중괄호쌍을 사용하면 된다. 이렇게 하면 함수 몸체에서는 아예 타입 인수를 생략할 수 있다.

```
Fixpoint repeat''' {X : Type} (x : X) (count : nat) : list X :=
  match count with
  | 0        => nil
  | S count' => cons x (repeat''' x count')
  end.
```

앞으로는 다형 함수의 정의는 항상 이 방식을 사용할 것이다. 예를 들어 app, rev, length 등의 함수도 이 방식으로 다시 정의해서 사용한다.

Arguments nil {X}와 같이 중괄호쌍으로 타입 변수를 둘러 사용하는 경우, nil의 인수를 암묵적implicit 인수라고 한다. 암묵적 타입 인수는 참 편리하지만 하나의 약점이 있다. 다음의 예를 보라.

```
Fail Definition mynil := nil.
Fail Check mynil nat.
Fail Check nil nat.
```

mynil의 정의에서 우변은 타입인수를 받아들이지 않으므로(암묵적으로 정의되었으므로) 이런 정의는 있을 수 없다. nil 에 공급하는 타입 인수는 타입 추론이 가능한 환경에서만 의미를 가진다.

이 문제는 다음과 같이 @를 앞붙여 사용하면 해결된다.

```
Definition mynil := @nil.
Check mynil nat. (* mynil nat: list nat *)
Check @nil nat. (* nil nat: list nat *)
```

또는 다음과 같이 해도 된다.

```
Definition mynil_nat := @nil nat.
Check mynil_nat. (* mynil_nat: list nat *)
```

암묵적 인수를 사용하는 생성자의 타입에 나타나는 타입 인수의 타입에는 물음표 ?를 앞붙여 그 인수가 암묵적 인수임을 나타낸다. 앞서 말했듯이 이런 생성자에 @를 앞붙이면 타입인수가 암묵적에서 명시적으로 바뀐다.

```
Check nil. (* list ?X *)
Check @nil. (* forall X, Type, list X *)
```

Polymorphic Pairs

다형 순서쌍polymorphic pair을 다음과 같이 정의한다.

```
Inductive prod (X Y : Type) : Type :=
| pair (x : X) (y : Y).

Arguments pair {X} {Y}.

Notation "( x , y )" := (pair x y).

Notation "X * Y" := (prod X Y) : type_scope.
(* ↑ Note that [type_scope] prevents confusing products with multiplications. *)

Definition fst {X Y : Type} (p : X * Y) : X :=
  match p with
  | (x, y) => x
  end.

Definition snd {X Y : Type} (p : X * Y) : Y :=
```

```
  match p with
  ¦ (x, y) => y
  end.
```

타입이 다른 두 타입의 곱도 정의할 수 있다.

```
Check pair 3 true. (* ..: prod nat bool *)
Check (3, true). (* ..: nat * bool *)
```

두 리스트를 하나의 리스트로 합쳐 순서쌍들의 리스트를 만드는 함수 `combine`을 다음과 같이 정의할 수 있다. 통상 이 함수를 `combine` 대신 `zip`이라고 부른다.

```
Fixpoint combine X Y: Type (lx: list X) (ly: list Y): list (X * Y) :=
  match lx, ly with
  ¦ [], _ => []
  ¦ _, [] => []
  ¦ x :: tx, y :: ty => (x, y) :: (combine tx ty)
  end.

Check combine [1;2;3;4] [true;false].
Compute combine [1;2;3;4] [true;false].
(*↑ [(1, true), (2, false)]: list(nat * bool) *)
```

Polymorphic Options

```
Inductive option (X:Type) : Type :=
  ¦ Some (x : X)
  ¦ None.

Arguments Some {X}.
Arguments None {X}.

Fixpoint nth_error X: Type (l: list X) (n: nat): option X :=
  match l with
  ¦ nil => None
  ¦ a :: l' => match n with
    ¦ O => Some a
    ¦ S n' => nth_error l' n'
    end
  end.

Compute (nth_error [4;5;6;7] 0). (* = Some 4 : option nat *)
Compute (nth_error [[1];[2]] 1). (* = Some [2] : option (list nat) *)
Compute (nth_error [true] 2). (* = None : option bool *)
```

5.2 Functions as Data

Higher-Order Functions

고차 함수(*Higher-order function*)란 인수, 또는 리턴값이 함수인 함수를 말한다. 예를 들면

```
Definition doit3times {X: Type} (f: X -> X) (n: X) : X :=
  f (f (f n)).

Check @doit3times. (* forall X : Type, (X -> X) -> X -> X *)

Definition my_square (n: nat) := n * n.
Compute doit3times my_square 2. (* = 256 : nat *)
```

Filter

리스트 l: list X에서 술어predicate test: X -> bool를 만족하는 원소들만 추려서filter 만든 리스트를 리턴하는 함수 filter를 다음과 같이 정의할 수 있다.

```
Fixpoint filter {X: Type} (test: X -> bool) (l: list X) : list X :=
  match l with
  | [] => []
  | h :: t => if f h then h :: (filter test t)
              else filter test t
  end.
```

리스트의 리스트가 주어졌을 때 길이가 1인 원소만 추려내기 위한 테스트 함수를 다음과 같이 정의한다.

```
1 Definition test_len1 {X: Type} (l: list X) : bool :=
2   (length l) =? 1.
3
4 Compute (filter test_len1 [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]).
5 (* ↑  = [ [3]; [4]; [8] ] : list (list nat) *)
```

Anonymous Functions

filter 함수를 사용할 때 filter의 인수로 사용할 술어, 예를 들어 test_len1 같은 함수는 딱 한 번만 쓰고 버리게 되는 경우가 많다.

이런 함수는 이름을 주지 않고 정의하여 사용할 수 있다. 위의 라인 1–3은 다음과 같이 바꿀 수 있다.

```
Compute filter (fun l => (length l) =? 1) [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ].
```

이렇게 (fun .. => ..) 구문으로 정의한 함수를 익명 함수(*anonymous function*)라고 한다.

익명 함수의 예를 하나 더 들어 보자. 자연수의 리스트가 주어졌을 때, 이 리스트의 원소들 중에 7보다 큰 짝수들만 추출하여 리턴하는 함수 filter_even_gt7을 다음과 같이 정의할 수 있다.

```
Definition ltb (n m: nat): bool :=
  (S n) <=? m.

Notation "x <? y" := (ltb x y) (at level 70) : nat_scope.

Definition filter_even_gt7 (l : list nat) : list nat :=
  filter (fun n => even n && (7 <? n)) l.
```

```
Example test_filter_even_gt7 :
  filter_even_gt7 [1;2;6;9;10;3;12;8] = [10;12;8].
Proof. reflexivity. Qed.
```

익명함수 대신 지역 함수(*local function*)를 사용하는 방법도 있다.

```
Definition filter_even_gt7' (l : list nat) : list nat :=
  let f n := even n && (7 <? n) in filter f l.

Compute filter_even_gt7' [1;2;6;9;10;3;12;8]. (* = [10; 12; 8]: list nat *)
```

주어진 리스트를 술어를 만족하는 원소들로 이루어진 리스트와 그렇지 않은 원소들로 이루어진 리스트로 분할하는 함수 partition을 다음과 같이 쉽게 정의할 수 있다.

```
Definition partition {X : Type} (test : X -> bool) (l : list X)
  : list X * list X :=
(filter test l, filter (fun x => negb (test x)) l).
```

Map

리스트 l: list X의 각 원소에 함수 f: X -> Y를 적용하여 만들어진 리스트를 리턴하는 함수 map을 다음과 같이 정의할 수 있다.

```
Fixpoint map {X Y: Type} (f: X -> Y) (l: list X) : list Y :=
  match l with
  | [] => []
  | h :: t => (f h) :: (map f t)
  end.
```

다음은 map 관련 연습문제들이다. 첫 번째는 정리를 증명하는 것이다.

```
Theorem map_rev : forall (X Y : Type) (f : X -> Y) (l : list X),
  map f (rev l) = rev (map f l).
```

귀납단계에 가 보면 다음과 같은 보조정리가 필요하다는 것을 알게 된다.

```
Lemma map_dist_r: forall (X Y : Type) (f : X -> Y) (l1 l2 : list X),
  map f (l1 ++ l2) = (map f l1) ++ (map f l2).
Proof.
  intros. induction l1 as [| h l' IHl'].
  - simpl. reflexivity.
  - simpl. rewrite -> IHl'. reflexivity. Qed.
```

그 다음은 쉬우므로 생략한다. ✓

이번에는 다음과 같은 기능을 하는 flat_map을 만들어 보자. 이 함수가 하는 일은 X의 원소들로 이루어진 리스트의 각 원소에 함수 f: X -> list Y를 적용하여 만들어진 리스트들을 모두 합쳐서 만든 리스트를 리턴하는 것이다.

```
flat_map (fun n => [n;n+1;n+2]) [1;5;10] = [1; 2; 3; 5; 6; 7; 10; 11; 12].
```

답은 평이한 방법으로 얻을 수 있다.

```
Fixpoint flat_map X Y: Type (f: X -> list Y) (l: list X)
                   : (list Y) :=
  match l with
  | [] => []
  | h :: t => (f h) ++ (flat_map f t)
  end.
```

Fold

다음의 함수 fold가 무슨 일을 하는지 생각해 보라.

```
Fixpoint fold {X Y: Type} (f: X -> Y -> Y) (l: list X) (b: Y): Y :=
  match l with
  | nil => b
  | h :: t => f h (fold f t b)
  end.
```

이 함수는 다른 프로그래밍 언어에서 흔히 reduce라고 부른다. 간단한 경우를 예로 들어 X = Y = nat로 두었을 때 f는 자연수의 2항연산이 되어야 한다. 예를 들어 fold plus l 0를 실행하면 리턴값은 리스트의 모든 원소들의 합이 된다. fold를 *accumulator*라고 한다면 b는 초깃값이다. 덧셈에서는 0, 곱셈에서는 1을 초깃값으로 주면 된다.

여러 리스트들을 합치는 것도 fold app ...를 사용하면 쉽게 할 수 있다.

Functions That Construct Functions

함수를 만들어 리턴하는 함수는 쉽게 만들 수 있다. 리턴되는 함수의 속성(파라메터, 형태 등)을 인수로 받아 그에 맞는 함수를 몸체에서 만들어 리턴하면 된다.

가장 쉬운 상수함수부터 보자.

```
Definition constfun {X: Type} (x: X) : nat -> X :=
  fun (k: nat) => x.

Definition ftrue := constfun true.
Compute ftrue 0. (* = true: bool *)
```

$x \mapsto ax + b : \mathbb{N} \to \mathbb{N}$를 리턴하는 함수를 만들어 보자.

```
Definition affine_fn (a b: nat) : nat -> nat :=
  fun (x: nat) => a * x + b.

Definition fn_times2_plus1 := affine_fn 2 1.

Compute fn_times2_plus1 7. (* = 15: nat *)
```

Exercises

① fold를 이용하여 리스트의 길이를 구하는 함수 fold_length를 다음과 같이 만들 수 있다.

```
Definition fold_length X: Type (l: list X) : nat :=
  fold (fun _ n => S n) l 0.
```

② fold를 이용하여 map과 동일한 기능을 하는 함수 fold_map을 만들어 보자.

```
Definition fold_map {X Y: Type} (f: X -> Y) (l: list X) : list Y :=
  fold (fun x ly => (f x) :: ly) l [].
```

③ fold_map이 정말로 map과 동일한 기능을 한다는 것을 증명해 보자.

```
1 Theorem fold_map_correct : forall X Y (f : X -> Y) (l : list X),
2   fold_map f l = map f l.
3 Proof.
4   intros. induction l as [| h l' IHl'].
5   - unfold fold_map. simpl. reflexivity.
6   - simpl. rewrite <- IHl'.
7     unfold fold_map. simpl. reflexivity. Qed.
```

라인 5와 라인 7에서 사용한 unfold fold_map. simpl.은 생략해도 된다. reflexivity는 이 두 명령을 자동으로 포함하여 실행해 주기 때문이다.

여기서 unfold, simpl 및 reflexivity 책략에 대하여 조금 더 알아보기로 하겠다.

unfold는 p13에서 고울 프랍에 나타난 프랍의 이름을 인수로 하여 사용되었다.

unfold *name_of_proposition*

이 명령을 실행하면 고울 프랍은 그것의 정의 표현으로 바뀐다. 그런데 실은 증명 스크립트에서 unfold가 이런 식으로 사용되는 경우는 별로 없다. unfold는 대부분의 경우 고울 프랍에 나타난 함수의 이름 문자열을 인수로 사용되며, 이때 그 문자열은 그 함수의 정의로 바뀌게 된다.

unfold *name_of_function*

unfold의 인수는 고울 프랍, 혹은 컨텍스트의 가설에 암묵적으로 나타난 경우에도 사용할 수 있다. 이러한 예는 p108에 나와 있다.

simpl은 unfold를 포함하지 않는다. simpl이 간략화 계산을 못하는 경우에는 그 앞에 unfold를 써 주면 되는 경우가 많다. 때로는 그 앞에 destruct를 써 주어야 한다. 이러한 예는 p46, p59 등에 나와 있다.

reflexivity는 unfold와 simpl을 포함한다. reflexivity가 실해하는 간략화 계산을 보고 싶다면 그 앞에 simpl., 혹은 unfold ⟨my_function⟩. simpl.을 써 주면 된다. reflexivity는 destruct는 포함하지 않는다.

Currying

f: X -> Y -> Z는 함수 (f x) : Y -> Z를 리턴하는 1변수 함수이지만, 한편 f x y : Z이므로 (x, y) : X * Y를 인수로 받는 2변수 함수로도 생각할 수 있다. X * Y -> Z의 원소를 X -> Y -> Z의 원소로 변환하는 함수를 커링(*currying*)이라고 한다. 아래에 커링 함수를 prod_curry라는 이름으로 정의하였다.

```
Definition prod_curry {X Y Z: Type} (f: X * Y -> Z) (x: X) (y: Y) : Z :=
  f (x, y).
```

이건 다음과 같이 정의해도 된다.

```
Definition prod_curry' {X Y Z: Type} (f: X * Y -> Z) : X -> Y -> Z :=
  fun (x: X) (y: Y) => f (x, y).
```

또는 다음과 같이 정의해도 된다.

```
Definition prod_curry'' {X Y Z : Type} (f : X * Y -> Z) : X -> Y -> Z :=
  fun (x: X) => fun (y: Y) => f (x, y).

Definition prod_curry''' {X Y Z : Type} (f : X * Y -> Z) (x: X) : Y -> Z :=
  fun (y : Y) => f (x, y).
```

타입 확인.

```
Check @prod_curry.
(* @prod_curry : forall X Y Z : Type, (X * Y -> Z) -> X -> Y -> Z *)
Check @prod_curry'.
(* @prod_curry' : forall X Y Z : Type, (X * Y -> Z) -> X -> Y -> Z *)
Check @prod_curry''.
(* @prod_curry'' : forall X Y Z : Type, (X * Y -> Z) -> X -> Y -> Z *)
Check @prod_curry'''.
(* @prod_curry''' : forall X Y Z : Type, (X * Y -> Z) -> X -> Y -> Z *)
```

이 정의들은 모두 동등하다. 하나만 보이겠다.

```
Theorem prod_curry_eq : forall (X Y Z: Type) (f: X * Y -> Z) x y,
  prod_curry f x y = prod_curry' f x y.
Proof. reflexivity. Qed.
```

이제 prod_curry의 역함수를 만들어 보자.

```
Definition prod_uncurry {X Y Z : Type}
  (f : X -> Y -> Z) (p : X * Y) : Z := f (fst p) (snd p).

Check @prod_uncurry.
(* @prod_uncurry: forall X Y Z : Type, (X -> Y -> Z) -> X * Y -> Z *)
```

이 두 함수가 서로 역함수라는 것을 증명하자.

```
Theorem uncurry_curry : forall (X Y Z : Type),
  forall (f : X -> Y -> Z) x y,
  prod_curry (prod_uncurry f) x y = f x y.
```

이건 증명이 너무 쉬워서 생략하고, 이것의 역에 대해서는 증명이 아주 조금 다른 부분이 있다.

```
Theorem curry_uncurry : forall (X Y Z : Type)
                        (f : (X * Y) -> Z) (p : X * Y),
  prod_uncurry (prod_curry f) p = f p.
Proof.
  intros. destruct p as [x y]. reflexivity. Qed.
```

다음은 리스트의 길이와 nth_error의 관계에 대한 팩트이다. 먼저 nth_error의 정의는 다음과 같음을 상기하자.

```
Fixpoint nth_error {X: Type} (l: list X) (n: nat): option X :=
  match l with
  | nil => None
  | a :: l' => match n with
    | 0 => Some a
    | S n' => nth_error l' n'
    end
  end.
```

nth_error의 정의를 보면 다음 팩트가 성립함을 알 수 있다. 이것을 증명하는 데는 아직 공부하지 않은 책략들이 필요하므로 일단 informal proof를 얻도록 한다.

> 길이가 n인 리스트 l에 대해서 nth_error l n = None이다.

먼저 틀린 증명을 보여주겠다. 여기서 어디가 잘못되었는지 알아내어 올바른 증명을 찾도록 할 것이다.

Wrong Proof. 다음의 프랍을 증명해야 한다.

```
forall (X: Type) (l: list X) (n: nat),
  length l = n -> nth_error l n = None.
```

X가 타입이라고 하고, 리스트 l에 대한 귀납을 사용한다.

- 기저단계: l = nil로 둔다. nth_error nil n은 정의에 의해서 None이다.
- 귀납단계: l = n :: l'로 둔다. nth_error (n :: l') n의 값은 정의에 의해서 n = 0인 경우와 n = S n'인 경우로 나누어 생각해야 한다.

  ① n = 0 경우에는 length l = 0로부터 l = nil을 얻는다. 이 경우에 nth_error nil 0의 값은 정의에 의해서 None이다.

  ② n = S n' 경우, 귀납가설 IHl' : length l' = S n' -> nth_error l' (S n') = None을 가정할 수 있다. 그런데 이 가정은 증명에 도움되지 않는다! 우리가 귀납가설로 원하는 것은 다음의 프랍이다.

  IHl'2 : length l' = n' -> nth_error l' n' = None

  length l = n을 가정하면, length l = length n :: l' = S n'이고 이것으로부터 length l' = n'을 얻을 수 있고, 이때 IHl'2를 이용하여 nth_error l' n' = None을 얻게 된다. 이제 nth_error l n = nth_error (n :: l') (S n')의 값은 nth_error의 정의에 의하여 nth_error l' n'의 값과 같다.

  이것으로부터 우리가 원하던 nth_error l n = None을 얻는다. ✓

관건은 귀납가설에서 S n' 대신 n'가 나타나야 한다는 것이다. 이를 위하여 다음과 같이 증명을 고쳐 보자.

Proof. 다음의 프랍을 증명해야 한다.

```
forall (X: Type) (l: list X),
  forall (n: nat), length l = n -> nth_error l n = None.
```

이렇게 두고 앞서와 같이 증명을 진행하면 귀납가설을 다음과 같은 형태로 얻게 된다.

```
IH1' : forall n : nat, length l' = n -> nth_error l' n = None
```

여기서 n := n'로 instantiate하면 우리가 원했던 IH1'2를 얻게 된다. 그 다음은 생략한다. □

5.3 Church Numerals

우리는 Coq에서 자연수를 다루기 위한 타입 nat를 가지고 있다. 자연수의 타입은 이것 말고 다른 방법으로도 정의할 수 있는데 그중에 가장 유명한 것이 처지 뉴머럴(*Church numeral*) cnat이다.

```
Definition cnat := forall X: Type, (X -> X) -> X -> X.
```

이것은 다형 타입이다. nat는 Inductive 타입이었으므로 생성자를 사용했지만 cnat은 생성자도 없고, 아무튼 nat와는 사뭇 다르다.

타입은 정의했는데 아직 타입의 원소는 하나도 없다. 생성자가 없으니 inhabit 하기가 막막한 것 같지만, 실은 전에 소개했던 doit3times, 정확히 말하자면 @doit3times가 cnat의 원소 중 하나이다.

```
Check @doit3times.
(* @doit3times: forall X: Type, (X -> X) -> X -> X *)

Print doit3times.
(** doit3times = fun (X : Type) (f : X -> X) (x : X) =>
                 f (f (f x)) : forall X : Type, (X -> X) -> X -> X *)

Compute @doit3times nat (fun n => n * n) 2. (* = 256 : nat *)
Compute doit3times (fun n => n * n) 2. (* = 256 : nat *)
```

자연수 3은 다음과 같이 정의한다.

```
Definition three : cnat := @doit3times.
```

zero, one, two를 다음과 같이 정의할 수 있다.

```
Definition zero : cnat := fun (X: Type) (f: X -> X) (x: X) => x.
Definition one : cnat := fun (X: Type) (f: X -> X) (x: X) => f x.
Definition two : cnat := fun (X: Type) (f: X -> X) (x: X) => f (f x).

Compute two nat S 0. (* = 2 : nat *)
```

이런 식으로 cnat의 모든 원소를 정의하는 것은 불가능하다. nat에서 보았듯이 cnat에서도 바로 뒤successor 함수를 만들어 사용하면 좋을 것이다.

```
Definition scc (n: cnat) : cnat :=
  fun (X: Type) (f: X -> X) (x: X) => f (n X f x).
Check scc. (* scc: cnat -> cnat *)
```

scc의 정의에서 이것이 cnat 타입 원소를 받아서 cnat 타입 원소를 리턴하는 함수라는 것이 보이는가? cnat 타입 원소는 (X: Type) (f: X -> X) (x: X)를 받아서 X 타입 원소를 리턴해야 한

다. scc의 정의를 보면 리턴 값이 f (n X f x)인데 n : cnat이므로 n X f x는 X 타입 원소이다. 그런데 f : X -> X이므로 f (n X f x)는 X의 원소가 맞다.

한편 cnat 타입 원소는 (X: Type) (f: X -> X)를 받아서 X -> X 타입 원소를 리턴한다고도 말할 수 있으므로, 바로뒤 함수를 다음과 같이 정의해도 된다.

```
Definition scc' (n: cnat) : cnat :=
  fun (X: Type) (f: X -> X) => fun (x: X) => f (n X f x).
Check scc'. (* scc': cnat -> cnat *)

Fact scc_equiv : forall (n: cnat),
  scc n = scc' n.
Proof. intros. reflexivity. Qed.
```

scc와 관련하여 마땅히 성립해야 할 기본적인 사실들을 확인해 보자.

```
1 Example scc_1 : scc zero = one.
2 Proof. unfold scc. unfold zero. unfold one. reflexivity. Qed.
3
4 Example scc_2 : scc one = two.
5 Proof. reflexivity. Qed.
6
7 Example scc_3 : scc two = three.
8 Proof. reflexivity. Qed.
```

라인 2의 reflexivity 앞에 있는 3개의 명령은 생략해도 되지만, 계산이 진행되는 과정의 이해를 돕기 위해서 넣어 두었다. (simpl은 아무런 효과가 없다.) 라인 5와 라인 8도 이렇게 바꾸어 보라.

주의할 것 하나가 있는데 Compute scc two.의 결과가 three로 나타나지 않는다는 것이다. 물론 scc_3에서 증명되었듯이 scc two가 표상하는 대상은 three가 표상하는 대상과 같다. 즉, 두 표현의 정규형식normal form이 동일하다는 것이고 이 사실은 아래의 라인 1과 라인 2에서 확인할 수 있다. 다만 Coq은 f 대신 x, x 대신 x0를 사용했다.

```
1 Compute scc two. (* = fun (X : Type) (x : X -> X) (x0 : X) => x (x (x x0)) : cnat *)
2 Compute three. (* = fun (X : Type) (x : X -> X) (x0 : X) => x (x (x x0)) : cnat *)
```

더 구체적으로 scc를 파악하려면 다음과 같이 하면 된다.

```
Compute three nat S 0. (* = 3: nat *)
Compute scc one nat S 0. (* = 2 : nat *)
```

쳐치 뉴머럴에 대한 바로뒤 함수를 정의했으니 이제 더하기, 곱하기 등의 연산을 정의해 보자. 우리는 귀납 대신 함수의 합성function composition을 사용할 것이며 이것이 쳐치 뉴머럴의 특징 중 하나이다. 다음의 정의에서 f를 succ로, x를 zero로 대체하면 이해하기가 더 쉬울 수 있다. (SF에서는 이렇게 했는데 이게 어쩌면 더 혼동스러울 수도 있겠다는 생각이다.)

```
Definition cplus (n m: cnat) : cnat :=
  fun (X: Type) (f: X -> X) (x: X) =>
    n X f (m X f x).

Definition cmult (n m: cnat) : cnat :=
```

```
    fun (X: Type) (f: X -> X) (x: X) =>
      n X (m X f) x.

Definition cmult' (n m: cnat) : cnat :=
  fun (X: Type) (f: X -> X) (x: X) =>
    n X (fun y => m X f y) x.

Definition cexp (n m: cnat) : cnat :=
  fun (X: Type) (f: X -> X) (x: X) =>
    (m (X -> X) (n X) f) x.
```

확인해 보자.

```
Compute plus two one.
(* ↑  = fun (X : Type) (x : X -> X) (x0 : X) => x (x (x x0)) : cnat*)
Compute cplus two three nat S 0. (* = 5 : nat *)

Example cplus_2 : cplus two three = cplus three two.
Proof. reflexivity. Qed.

Fact mult_equiv: forall (n m: cnat),
  mult n m = cmult n m.
Proof. intros. reflexivity. Qed.

Compute cmult two three nat S 0. (* = 6 : nat *)

Example cmult_2 : cmult zero (cplus three three) = zero.
Proof. reflexivity. Qed.

Compute cexp three two nat S 0. (* = 9 : nat *)
Compute cexp two (plus two (mult three one)) nat S 0. (* = 32 : nat *)

Example cexp_1 : cexp two two = cplus two two.
Proof. reflexivity. Qed.

Example cexp_2 : cexp three zero = one.
Proof. reflexivity. Qed.

Example cexp_3 : cexp three two = cplus (cmult two (cmult two two)) one.
Proof. reflexivity. Qed.
```

cplus와 cmult가 각각 덧셈과 곱셈임은 각자 확인해 보기 바란다. cexp는 이해하기가 그리 쉽지 않으므로 어떻게 작동하는지를 예를 들어 설명하겠다. 정의를 다시 한 번 보면 다음과 같다.

```
Definition cexp (n m: cnat) : cnat :=
  fun (X: Type) (f: X -> X) (x: X) =>
    (m (X -> X) (n X) f) x.
```

m: cnat의 첫 인수가 X -> X이므로 두 번째 인수인 n X의 타입은 (X -> X) -> (X -> X)가 되어야 하고 세 번째 인수인 f의 타입은 X -> X이어야 한다. f에 대해서는 타입이 정의에 f: X -> X로 주어져 있으므로 문제가 없고, n X: (X -> X) -> X -> X를 확인해 보아야 하겠다.

n: cnat이므로 n X f x: X이다. 즉 n X f: X -> X이다. 이것으로부터 n X: (X -> X) -> X -> X임이 확인된다.

수학의 표현을 사용하면 $n\ X\ f\ x = f^n(x)$이므로 $(n\ X)\ f = f^n$이다.
다시 말하면 $n\ X = ($fun $f\ \Rightarrow\ f^n)$이다.

(1) one $(X \to X)\ (n\ X)\ f$는 $(n\ X)\ f = f^n$이다.

(2) two $(X \to X)\ (n\ X)\ f$는 $(n\ X)\big((n\ X)\ f\big) = (n\ X)(f^n) = (f^n)^n = f^{(n^2)}$이다. 좀 더 자세히 들여다 보면 $(n\ X)(f^n)(x) = f^n(f^n(\cdots f^n(x)\cdots)) = f^{(n^2)}(x)$이다.

(3) three $(X \to X)\ (n\ X)\ f$는 $(n\ X)\big((n\ X)^2\ f\big) = (n\ X)(f^{(n^2)}) = (f^{(n^2)})^n = f^{(n^3)}$이다.

이런 식으로 유추하면 $m\ (X \to X)\ (n\ X)\ f$는 $f^{(n^m)}$이다. ✓

이상은 물론 엄격한 증명은 아니다. 하지만 이 정도면 이 정의의 기본 아이디어를 이해하기에는 충분하다고 본다. 이것이 정말 지수함수라는 것을 엄격하게 증명하려면 Coq에서 지수함수 cexp'를 Inductive를 써서 정의하고 이 두 함수가 항상 같은 값을 가짐을 증명하면 된다.

6 Tactics

6.1 Part 1

The apply Tactic

apply 책략은 이전에 p57에서 한 번 나왔었지만 그때는 apply의 인수가 전칭문인 경우에 한정하여 설명하였고, 이번에는 더 일반적인 경우에 대하여 알아보고자 한다.

① 복습하는 의미에서 아주 간단한 예로 시작하자.

```
1 Theorem apply1 : forall (n m : nat),
2   n = m -> n = m.
3 Proof.
4   intros n m. intros eq. (* This line can be replaced with intros n m eq. *)
5   apply eq. Qed.
```

다음은 라인 4를 실행한 후의 Coq Goals 화면이다.

```
6 n, m : nat
7 eq : n = m
8 (1 / 1) ________________________
9 n = m
```

이 상태에서 rewrite -> eq를 실행하면 고울 프랍이 n = n으로 바뀌어 이어서 reflexivity로 증명을 마무리 할 수 있다. 혹은 rewrite <- eq를 실행하면 고울 프랍이 m = m으로 바뀌므로 역시 reflexivity로 증명이 완료된다.

그런데 이렇게 고울 프랍과 동일한 프랍이 가설 eq: n = m으로 존재할 때는 라인 5와 같이 apply eq를 쓰면 더 간단하다.[1] ✓

② 이번에는 가설 H가 전칭문이고 고울 프랍은 H의 instance일 때 apply H의 활용 예이다.

```
1 Theorem apply2 : forall (f: nat -> nat) (m: nat),
2   (forall n: nat, f n = n) ->
3   f (m + 2) = m + 2.
4 Proof.
```

[1] 이럴 때는 exact eq를 써도 된다.

```
5   intros f m H.
6   apply H.  Qed.
```

라인 5를 실행한 후의 Coq Goals 화면은 다음과 같다.

```
f : nat -> nat
m : nat
H : forall n : nat, f n = n
(1 / 1) ------------------------
f (m + 2) = m + 2
```

H에 치환 (n := m + 2)를 적용하면 고울 프랍과 정확히 일치한다. 이럴 때 라인 6의 apply H를 실행하면 Coq는 이 치환을 알아서 찾아서 적용해 주어 증명이 끝난다. ✓

③ 그 다음 예는 가설이 조건문일 때 apply의 활용법을 보여준다. 일반적으로 고울 프랍이 Q일 때 조건문 가설 H: P -> Q가 존재한다면 apply H를 실행했을 때 고울 프랍이 P로 바뀐다. 이 책략이 타당함은 다음과 같이 설명된다.

> 지금 Q를 증명해야 한다. 그런데 H: P -> Q가 참임을 가정하고 있으므로 P만 증명하면 충분하다. 따라서 고울 프랍을 P로 교체한다.

이를 조금 더 일반화 하여 조건문 가설이 H: P -> Q -> R이고 고울 프랍이 R일 때 apply H를 실행하는 경우를 생각해 보자. 이럴 때는 subgoal이 2개 생긴다—각각의 고울 프랍은 P와 Q이다.[2]

```
1 Theorem apply3 : forall (n m : nat),
2   n = m ->
3   (n = m -> n <= m) ->
4   n <= m.
5 Proof.
6   intros n m eq1 eq2.
7   apply eq2.
8   exact eq1. Qed.
```

다음은 라인 6을 실행한 후의 Coq Goals 화면이다.

```
n, m : nat
eq1 : n = m
eq2 : n = m -> n <= m
(1 / 1) ------------------------
n <= m
```

고울 n <= m이 가설 eq2 : n = m -> n <= m의 후건과 일치함을 알 수 있다. 이럴 때 라인 7의 apply eq2를 실행하면 고울이 eq2의 전건 n = m으로 바뀐다. 그런데 때마침 이 전건은 가설 eq1과 동일하다. 그러면 이제 고울 프랍이 eq1과 일치하므로 라인 8의 exact eq1을 실행하여 증명을 완료할 수 있다. ✓

④ 좀 더 흥미로운 경우는 가설 H가 조건문을 몸체로 하는 전칭한정문이고, 이 전칭문의 적당한 인스턴스가 고울 프랍과 일치할 때의 apply H이다. 다음의 예를 보자.

[2] 이를 이용한 증명의 예는 p90에서 볼 수 있다.

```
1  Theorem apply4 : forall (n m: nat),
2    (n,n) = (m,m) ->
3    (forall (q r: nat), (q,q) = (r,r) -> [q] = [r]) ->
4    [n] = [m].
5  Proof.
6    intros n m H1 H2.
7    apply H2. exact H1.  Qed.
```

다음은 라인 6을 실행한 후의 Coq Goals 화면이다.

```
8   n, m : nat
9   H1 : (n, n) = (m, m)
10  H2 : forall q r : nat, (q, q) = (r, r) -> [q] = [r]
11  (1 / 1) ------------------------
12  [n] = [m]
```

라인 10의 H2에 치환 (q := n) (r := m)을 적용하여 전칭한정사를 소거하면 고울 프랍이 H의 인스턴스의 후건과 일치하게 된다. 이 치환은 Coq가 알아서 찾아준다. 그리고 고울 프랍은 H의 인스턴스의 전건인 (n, n) = (m, m)으로 바뀐다.

이 고울은 H1과 동일하므로 이제 exact H1을 써서 증명을 마칠 수 있다. ✓

지금까지의 모든 예에서 apply의 인수로는 컨텍스트에 들어 있는 가설을 사용했는데, 가설 대신 정리를 apply의 인수로 사용해도 된다. 참고로 정리는 프랍이므로 대부분 전칭문이다. 이런 예는 다음 하위절에서 다룰 것이다.

The apply with Tactic

Theorem apply2에서 라인 6의 apply H를 실행하면 필요한 치환 (n := m+2)를 Coq가 알아서 찾아준다고 하였다. 이 치환은, 원한다면 사용자가 다음과 같이 지정해 줄 수 있다. 라인 1과 라인 2 중 어느 것을 사용해도 된다.

```
1  apply (H (m+2)).
2  apply H with (n := m+2).
```

마찬가지로 Theorem apply4에서 라인 7의 apply H2를 실행하면 필요한 치환 (q := n) (r := m)을 Coq가 알아서 찾아준다. 이 치환을 사용자가 다음과 같이 지정해 줄 수 있다. 라인 1과 라인 2 중 어느 것을 사용해도 된다.

```
1  apply (H2 n m).
2  apply H2 with (q := n) (r := m).
```

이렇게 치환을 명시적으로 지정해 주는 것은 사용자의 취향에 따라 할 수도, 하지 않을 수도 있다. 그런데 때로는 필요한 치환을 Coq이 찾지 못하기 때문에 반드시 지정해 주어야 하는 경우가 있다. 다음의 예를 보라.

```
Example trans_eq_example : forall (a b c d e f : nat),
  [a;b] = [c;d] ->
  [c;d] = [e;f] ->
  [a;b] = [e;f].
```

증명은 평이하게 되므로 생략하였다. 그런데 다음의 보조정리를 이용하면 위의 trans_eq_example를 좀 더 간단하게 증명할 수 있다. trans_eq의 증명은 역시 생략한다.

```
Lemma trans_eq : forall (X:Type) (n m o : X),
  n = m -> m = o -> n = o.
```

이 보조정리를 이용하면 trans_eq_example를 다음과 같이 증명할 수 있다. 라인 5의 apply에서 치환을 명시적으로 지정하였다. 이 부분을 빼먹으면 증명이 되지 않는다.

라인 5에 apply trans_eq with [c;d]를 써도 된다. 하지만 apply (trans_eq [c;d])는 안된다.

```
1 Example trans_eq_example : forall (a b c d e f : nat),
2   [a;b] = [c;d] -> [c;d] = [e;f] -> [a;b] = [e;f].
3 Proof.
4   intros a b c d e f H1 H2.
5   apply trans_eq with (m:= [c;d]).
6   - exact H1.
7   - exact H2.   Qed.
```

(해설). 라인 4를 실행한 후의 Coq Goals 화면이다.

```
a, b, c, d, e, f : nat
H1 : [a; b] = [c; d]
H2 : [c; d] = [e; f]
(1 / 1) ------------------------
[a; b] = [e; f]
```

여기서 apply trans_eq를 실행하면 치환 (n := [a;b]) (m := [c;d]) (o := [e;f])가 필요할 것인데 이 3개의 치환들 중에 (m := [c;d])는 Coq이 제대로 찾아주지 못하므로 사용자가 apply ... with를 써서 지정해 주어야 하는 것이다.

라인 5를 실행한 후에 Coq Goals 화면이 다음과 같이 바뀐다.

```
Goal 1
a, b, c, d, e, f : nat
H1 : [a; b] = [c; d]
H2 : [c; d] = [e; f]
(1 / 2) ------------------------
[a; b] = [c; d]

Goal 2
a, b, c, d, e, f : nat
H1 : [a; b] = [c; d]
H2 : [c; d] = [e; f]
(2 / 2) ------------------------
[c; d] = [e; f]
```

이제는 라인 6, 7의 exact를 써서 각각의 고울을 증명할 수 있다. ✓

실은 Coq는 추이성의 증명을 위한 책략인 transitivity를 제공한다. apply with 대신 간단히 transitivity를 쓰면 된다. 이 책략의 사용법은 다음의 (6.1) 대신 (6.2)를 쓰는 것이다.

```
apply trans_eq with (m:= [c;d])        (6.1)
transitivity [c;d].                    (6.2)
```

The injection and discriminate Tactics

nat의 정의를 다시 보자.

```
Print nat. (* Inductive nat : Set :=
               O : nat
             | S : nat -> nat. *)
```

다음의 두 사실을 알 수 있다.

① 생성자 S는 단사injective이다. 즉 S n = S m이면 n = m이다: 단사성(*injectivity*)

② O는 어떤 n: nat에 대해서도 S n과 다르다: 배타성(*disjointness*)

이 원칙은 다른 모든 귀납적으로 정의된 타입에서도 성립한다. 생성자 S의 단사성은 Coq에서 다음과 같이 증명할 수 있다.

```
Theorem S_injective : forall (n m : nat),
  S n = S m -> n = m.
Proof.
  intros n m H1.
  replace (n) with (pred (S n)).
  - rewrite -> H1. simpl. reflexivity.
  - simpl. reflexivity. Qed.
```

이런 방식으로 모든 생성자의 단사성을 증명할 수 있다. 그러나 이런 방식은 매우 번거롭다. Coq는 이런 단사성을 증명하지 않고 그냥 사용할 수 있도록 하는 injection 책략을 제공한다. 예를 들어 위의 S_injective를 다음과 같이 간단하게 증명할 수 있다.

```
1 Theorem S_injective' : forall (n m : nat),
2   S n = S m -> n = m.
3 Proof.
4   intros n m H.
5   injection H as H'.
6   exact H'. Qed.
```

라인 4 실행 후의 Coq Goals 화면은 다음과 같다.

```
n, m : nat
H : S n = S m
(1 / 1) ------------------------
n = m
```

injection 책략은, 어떤 생성자 Cons에 대하여 가설 H : Cons t1 = Cons t2이 존재할 때

```
induction H as H'
```

구문을 써서 적용한다.[3] 그러면 가설 H는 새로운 가설 H' : t1 = t2로 바뀐다.

위의 예에서는 고울 프랍이 H'와 일치하였기에 바로 exact H'으로써 증명을 마칠 수 있었다. 고울이 꼭 t1 = t2가 아닌 경우에도 injection 책략을 사용할 수 있다. 다음의 예를 보자.

[3] injection을 사용할 때 as H' 없이 injection H 구문을 사용할 수도 있다. 이 경우 H에는 변화가 없고 고울이 조건문으로 바뀐다. 즉, 고울을 Q라고 할 때 새로운 고울 프랍은 t1 = t2 -> Q가 된다.

```
1 Theorem S_injective'' : forall (n m k: nat),
2   S n = S m -> m = k -> n = k.
3 Proof.
4   intros n m k H1 H2.
5   injection H1 as H1'.
6   transitivity m.
7   exact H1'. exact H2. Qed.
```

라인 4를 실행한 후의 Coq Goals 화면은 다음과 같다.

```
n, m, k : nat
H1 : S n = S m
H2 : m = k
(1 / 1) ------------------------
n = k
```

그 다음의 라인 5를 실행하면 H1은 H1' : n = m으로 바뀐다. 그러면 이제 transitivity를 사용하여 남은 증명을 쉽게 완성할 수 있게 된다.

때로는 injection에 의하여 등식을 2개 이상 얻게 되는 경우가 있다. 예를 들어 가설 H : (n, m) = (o, p)에 injection을 적용하면 n = o와 m = p라는 2개의 새로운 가설을 얻게 된다. 일반적으로 Cons가 2항 생성자인 경우 등식 Cons t1 s1 = Cons t2 s2에 injection을 적용하면 t1 = t2, s1 = s2라는 2개의 등식 가설을 얻게 된다. 이런 경우에는 injection 책략을 다음과 같은 구문으로 사용해야 한다.

```
injection H as H1 H2.
```

이상으로 생성자의 단사성에 대한 이야기를 마치고 이제 배타성에 대한 이야기를 시작하자. 배타성을 이용하는 책략은 discriminate라고 하는 것이다.

생성자 O와 S의 배타성을 증명하는 것으로 출발해야 할 것인데 이를 증명하기에 앞서 표현하는 것조차 현재로서는 쉽지 않다. 지금까지의 증명한 모든 명제는 등식 명제(의 전칭문), 혹은 이들로 이루어진 조건문(의 전칭문)뿐이었다.[4]

그런데 배타성은 등식 명제가 아니라 부정 명제이며, 부정 명제는 부정 결합자 not를 이용하여 not P형태인 것으로 정의한다. 이를 위하여, 항상 거짓인 프랍 False를 도입하고

```
not P := P -> False
```

로 정의한다. 이러한 내용은 다음 장에서 다룰 예정이고, 현재로서는 O와 S의 배타성은 다음과 같이 등식명제로 표현하는 수밖에 없다.

```
Theorem O_not_S' : forall (n : nat),
  0 =? S n = false.
```

그런데 실은 이 정리는 이전에 zero_neqb_S라는 이름의 정리로 증명한 적이 있다. 그때 증명에서 (0 =? S n) = false라는 고울을 simpl 하면 그냥 고울이 false = false로 바뀌었었는데, 이

[4] SF는 아직까지도 우리가 증명해야 할 대상인 명제, 즉 Prop에 대해서 구체적으로 언급한 적이 없다. 명제의 부정, 논리곱, 논리합, 존재한정문 등은 지금보다 훨씬 이전에 다루었어야 했다고 본다.

증명이 잘못된 부분은 전혀 없지만, 이 등식은 '=?' 정의에서 바로 나온 것이므로 증명을 통하여 새로 알아낸 것은 없다고 말할 수 있다.

'=?'와 '='의 차이를 이해하는 것은 Coq을 공부할 때 극복해야 할 장애물 중 하나이며 앞으로 이에 대한 논의를 많이 하게 될 것이다.

우리는 0와 S의 배타성을 증명에 이용할 때 전적으로 discriminate 책략에 맞길 것이다. 다시 말하면, 단사성은 injection 책략을 사용하지 않고도 증명에서 이용할 수 있었고, injection 책략은 단사성을 편리하게 이용하는 하나의 방편이었지만, 배타성은 discriminate 책략을 사용하지 않고는 이용할 수 없다는 것이다.

discriminate 책략의 사용 예를 보자.

```
1 Theorem discriminate_ex2 : forall (n : nat),
2   S n = 0 -> 2 + 2 = 5.
3 Proof.
4   intros n contra.
5   discriminate contra. Qed.
```

라인 4를 실행한 후의 Coq Goals 화면은 다음과 같다.

```
n : nat
contra : S n = 0
(1 / 1) ------------------------
2 + 2 = 5
```

가설의 이름을 contra로 지어 사용하는 이유는 이것이 모순명제contradiction임을 알고 있기 때문이다. 가설 중에 생성자들의 배타성에 의한 모순명제가 있으면 고울은 볼것도 없이 바로 discriminate 책략을 적용하면 된다.

destruct를 discriminate와 함께 사용하는 예를 보자. (참고로 이 정리의 역방향은 rewrite를 써서 아주 쉽게 증명된다.)

```
1 Theorem eqb_0_l : forall n,
2    0 =? n = true -> n = 0.
3 Proof.
4   intros. destruct n.
5   - reflexivity.
6   - discriminate H. Qed.
```

라인 4를 실행한 후의 Coq Goals 화면은 다음과 같다.

```
7  Goal 1
8  H : (0 =? 0) = true
9  (1 / 2) ------------------------
10 0 = 0
11
12 Goal 2
13 n : nat
14 H : (0 =? S n) = true
15 (2 / 2) ------------------------
16 S n = 0
```

라인 6의 discriminate는 라인 14의 H : (0 =? S n) = true에 적용하는 것인데 이것은 실은 생성자들의 배타성을 simpl 계산을 거쳐 사용한 것이다. 무슨 얘기냐 하면 라인 6을 다음과 같이 바꾸어 보면,

```
  - simpl in H.
    discriminate H. Qed.
```

simpl in H 뒤의 Coq Goals 화면은 다음과 같이 되는데,

```
n : nat
H : false = true
(1 / 1) ------------------------
S n = 0
```

이때 discriminate H로써 부울리언 생성자들의 배타성을 사용한다는 뜻이다.

여기서 처음 보는 구문 simpl in을 사용했다. 단순 simpl이 고울에 대하여 작동하는 것임에 반하여 이러한 simple in는 그 오른쪽에 나오는 가설 H에 대하여 작동한다. 이에 대해서는 다음 하위섹션에서 다루도록 하겠다.

이 하위섹션을 마치기에 앞서 또 하나의 책략 f_equal을 소개하고자 한다. 이 책략은 대략 injection의 '고울 버전'이라고 보면 된다. 컨텍스트에 있는 가설 S n = S m을 n = m으로 바꾸어 주는 것이 injection이라면, f_equal은 고울 프랍 S n = S m을 n = m으로, 그리고 일반적으로 임의의 함수 f와 그것의 인수 x에 대하여 고울 프랍 f x = f y를 x = y로 바꾸어 주는 것이다. 이 책략은 가설 중에 x = y가 있을 경우 흔히 쓰이며, 이럴 경우 rewrite를 써서 증명을 진행할 수도 있지만, 편리성을 위하여 제공된다.

f_equal에 대해서 주의할 것은 고울 프랍 f x = f y에 적용할 때 함수 f가 단사1-to-1 함수일 필요는 없다는 것이다. 왜냐하면 x = y는 f x = f y의 충분조건이지만 필요조건은 아니기 때문이다. 다만 f가 단사인 경우에는 x = y가 f x = f y의 필요충분조건이 된다.

```
1 Fact eq_implies_succ_equal : forall (n m : nat),
2   n = m -> S n = S m.
3 Proof.
4   intros n m H. (* H : n = m *)
5   apply f_equal. (* Goal : n = m *)
6   exact H. Qed.
```

Using Tactics on Hypotheses

책략은 그 디폴트 작동 대상이 고울이다. 그런데 고울을 작동 대상으로 할 때 컨텍스트 내의 어떤 가설을 사용하는 경우에는 그 가설을 책략 사용 명령에서 지정해야 한다. 그래서 책략을 적용한다고 말할 때 그 대상은 고울을 뜻하기도 하고 가설을 뜻하기도 한다. 이런 혼동을 막기 위하여 나는 책략의 타겟(*target*)과 인수(*argument*)라는 용어를 사용하고자 한다. 예를 들어

```
apply H.
```

라는 명령은 가설 H를 인수로, 고울을 타겟으로 한다. (타겟을 따로 지정하지 않으면 디폴트로 고울 프랍이 타겟이 된다.)

두 가지를 말하고자 한다. ① 인수는 또 다른 인수를 사용할 때가 있다. 예를 들어

```
injection H as H1 H2.
```

라는 명령은 가설 H를 인수로 하는데 여기에는 as 뒤의 H1 H2가 또 인수로 사용되고 있다.

② 지금까지는 모든 경우에 타겟은 고울이었지만 그렇지 않은 경우도 있다는 것이다. 이제 그런 경우를 살펴보자.

```
1  Theorem S_inj : forall (n m : nat) (b : bool),
2    ((S n) =? (S m)) = b  ->
3    (n =? m) = b.
4  Proof.
5    intros n m b H.
6    simpl in H.
7    exact H.  Qed.
```

라인 5를 실행한 후의 Coq Goals 화면은 다음과 같다.

```
n, m : nat
b : bool
H : (S n =? S m) = b
(1 / 1) ------------------------
(n =? m) = b
```

라인 6의 simpl in H는 타겟을 고울이 아닌 가설 H로 삼는다. (여기서 인수는, simpl 책략의 대부분의 경우에서 그러하듯이, 없다.) 그래서 그 결과로 가설이 (이름은 바뀌지 않고), =?의 정의에 의하여 H : (n =? m) = b로 바뀐다.

예를 하나 더 들어보자. 여기서는 symmetry와 apply의 타겟을 가설로 삼는다.

```
1  Theorem silly4 : forall (n m p q : nat),
2    (n = m -> p = q) ->
3      m = n -> q = p.
4  Proof.
5    intros n m p q EQ H.
6    symmetry in H.
7    apply EQ in H.
8    symmetry in H. exact H. Qed.
```

라인 6까지 실행한 후의 Coq Goals 화면은 다음과 같다.

```
n, m, p, q : nat
EQ : n = m -> p = q
H : n = m
(1 / 1) ------------------------
q = p
```

이어서 라인 7을 실행하면 타겟 H가 H : p = q로 바뀐다.

apply 책략을 조건문 가설 P -> Q를 인수로 하여 타겟에 적용할 때 두 가지 경우가 있다. (다음의 설명 ①과 ②는 모두, 가설이 조건문 P -> Q을 몸체로 하는 전칭한정문일 때도 적용된다. 단, 이때는 P와 Q는 각각 이들의 적당한 치환 인스턴스로 바꾸어 생각해야 한다.)

① 타겟이 고울이다. 이때는 타겟이 Q이어야 하고, 책략 실행 후에는 타겟이 P로 바뀐다. 왜냐하면 새로운 타겟 P를 증명한다면 이것으로 원래의 타겟 Q는 증명된 셈이기 때문이다.

② 타겟이 가설 H이다. 이때는 타겟이 H: P이어야 하고, 책략 실행 후에는 타겟이 H: Q로 바뀐다. 왜냐하면 가설 P -> Q와 P로부터 Q가 논리적 귀결로 얻어져 고울의 증명에 사용될 수, 즉 가설로 도입될 수 있기 때문이다.

이 경우에 컨텍스트의 변화를 보면 가설집합에 {P -> Q, P}가 포함되어 있던 것이 {P -> Q, Q}로 바뀌며 P가 사라졌는데, 혹시라도 나중에 P를 필요로 하게 되면 낭패다.

다음의 예를 보라. 라인 4의 specialize (H2 H1)은 apply H2 in H1를 대신한 것으로 이를 적용한 후의 결과가 어떻게 다른지는 증명 스크립트 아래에서 설명하겠다.

```
1 Example specialize_hyp : forall P Q : Prop, P -> (P -> Q) -> P /\ Q.
2 Proof.
3   intros P Q H1 H2. (* H1: P, H2: P -> Q *)
4   specialize (H2 H1). (* H1: P, H2: Q *)
5   split. (* Split the goal A /\ B into 2 subgoals A and B. *)
6   - exact H1.
7   - exact H2.
8 Qed.
```

H2: P -> Q이고 H1: P일 때

① apply H2 in H1을 적용하면 H2는 변화가 없고 H1은 Q로 변한다.
② specialize (H2 H1)을 적용하면 H2는 Q로 변하고 H1에는 변화가 없다.

specialize는 원래 전칭문의 묶인변수를 특정할 때 쓰이는 책략인데(바로 다음 하위절에서 배울 것이다.) 이런 상황에서도 사용된다. 구문이 좀 혼동스럽다는 생각이다.

Specializing Hypotheses

가설이 전칭문 H: forall (x: T), P(x)인데 우리가 원하는 것은 어떤 표현항 t에 대해서 P(t)라면, 즉 H: (x := t) P, 혹은 H: P(t)라면 다음과 같이 specialize 책략을 사용하면 된다.

```
1 Theorem specialize_example: forall n,
2   (forall m, m * n = 0) -> n = 0.
3 Proof.
4   intros n H.
5   specialize H with (m := 1).
6   simpl in H. rewrite add_comm in H. simpl in H.
7   apply H. Qed.
```

라인 4를 실행한 후의 Coq Goals 화면은 다음과 같다.

```
n : nat
H : forall m : nat, m * n = 0
(1 / 1) ------------------------
n = 0
```

이어서 라인 5를 실행하면 H는 H : 1 * n = 0으로 바뀐다. 그 다음은 설명할 필요가 없을 것이다.

specialize를 사용하는 증명의 예를 하나 더 보자.

```
1 Fact specialize_example2 : forall (m: nat) (f: nat -> nat),
2   (forall n: nat, f n = 2 * n) ->
3   f m = 2 * m.
4 Proof.
5   intros m f H.
6   specialize H with (n := m).
7   exact H. Qed.
```

이번 증명에서는 specialize 대신 apply를 써도 된다. 하지만 이전의 specialize_example의 증명에서는 apply를 쓸 수 없음에 유의하라.

이 책략은 전칭묶인변수의 값을 특정instantiate해 주는 것이므로 그 이름을 specialize 대신 instantiate라고 지었으면 어땠을까 한다.

Coq는 같은 이름의 책략이 여러 서로 다른 기능을 수행하는 경우가 많으며, 반대로 다른 이름의 책략이 같은 기능을 공유하여 수행하는 경우도 많으니 주의해야 한다. 변수의 특정과 관련된 책략으로는 specialize 외에도 apply, rewrite, intro 등이 있다. 이상은 전칭문의 경우이고 존재문의 경우에 대해서는 exists, intro가 더 있다.[5]

specialize 책략은, apply에서도 그러했듯이, 가설뿐만 아니라 정리를 인수로 사용할 수도 있다.

```
1 Example trans_eq_example''' : forall (a b c d e f : nat),
2   [a;b] = [c;d] ->
3   [c;d] = [e;f] -> [a;b] = [e;f].
4 Proof.
5 intros a b c d e f eq1 eq2.
6 specialize trans_eq with (m:=[c;d]) as H. (* trans_eq is a theorem *)
7 apply H.
8 exact eq1. exact eq2. Qed.
```

라인 5를 실행한 후의 Coq Goals 화면은 다음과 같다.

```
a, b, c, d, e, f : nat
eq1 : [a; b] = [c; d]
eq2 : [c; d] = [e; f]
(1 / 1) ------------------------
[a; b] = [e; f]
```

이 증명에 등호의 추이성transitivity을 사용할 것이므로 라인 5의 specialize의 인수로 trans_eq를 인수로 사용한다. 그리고 이때 도입되는 가설에 이름 H를 준다. 그러면 Coq Goals 화면에 다음의 가설이 추가된다. 다른 변화는 없다.

```
H : forall n o : list nat,
    n = [c; d] -> [c; d] = o -> n = o
```

이제 라인 7을 실행하면 Coq는 H의 후건의 후건 n = o를 고울 [a; b] = [e; f]에 매치시켜 다음의 치환을 얻어낸다.

```
(n := [a;b]) (o := [e;f])
```

[5] 존재문의 경우에는 묶인변수의 특정은 신규기호fresh symbol의 도입(흔히 Rule-C라고 부르는)이므로 조금 다른 개념이다.

그러므로 라인 7의 apply H는 다음의 프랍을 H로 보고 apply H하는 것과 같다.

```
H: [a; b] = [c; d] -> [c; d] = [e; f] -> [a; b] = [e; f]
```

그러면 Coq은 [a; b] = [c; d]와 [c; d] = [e; f]를 각각 고울 프랍으로 하는 두 개의 서브고울을 생성하여 Coq Goals 화면을 다음과 같이 바꿔 놓는다.[6] 이 경우 P는 전건, Q는 후건의 전건이다—이 둘을 각각 고울로 하여 증명하면 된다.

```
Goal 1
a, b, c, d, e, f : nat
eq1 : [a; b] = [c; d]
eq2 : [c; d] = [e; f]
H : forall n o : list nat,
    n = [c; d] -> [c; d] = o -> n = o
(1 / 2) ------------------------
[a; b] = [c; d]

Goal 2
(* same hypothesis as in Goal 1 *)
(2 / 2) ------------------------
[c; d] = [e; f]
```

그 다음은 설명할 필요가 없을 것이다. ✓

6.2 Part 2

Varying the Induction Hypothesis

induction 책략이 잘 안 먹히는 경우가 있다. 예를 들어 다음의 정리를 별 생각없이 증명하려 들면 어떻게 되는지 보자.

```
1  Theorem double_injective_FAILED : forall n m,
2    double n = double m ->
3    n = m.
4  Proof.
5    intros n m. induction n as [| n' IHn'].
6    - (* n = 0 *) simpl. intros eq. destruct m as [| m'] eqn:E.
7      + (* m = 0 *) reflexivity.
8      + (* m = S m' *) simpl in eq.
9        discriminate eq.
10   - (* n = S n' *) intros eq. destruct m as [| m'] eqn:E.
11     + (* m = 0 *) simpl in eq. discriminate eq.
12     + (* m = S m' *) f_equal.
13 Abort.
```

증명은 평이하게 잘 진행되다가 라인 12까지 마쳤을 때 Coq Goals 화면이 다음과 같이 바뀐다.

```
n', m, m' : nat
E : m = S m'
```

[6] 이렇게 하는 이유는 p80에 설명되어 있다.

```
IHn' : double n' = double (S m') -> n' = S m'
eq : double (S n') = double (S m')
(1 / 1) ------------------------
n' = m'
```

귀납가설 IHn'은 그것의 후건 n' = S m'이 고울 n' = m'과 매치되지 않으므로 사용할 수 없다. 그리고 eq는 double n' = double m'까지 환원시켜 놓는다고 해도(실제로 이것까지는 가능하다.) 아무 소용이 없다. 왜냐하면 double n' = double m' -> n' = m'는 바로 지금 우리가 증명하려는 정리이기 때문이다.

문제는 귀납가설 IHn'이 너무 융통성이 없다inflexible하다는 것이다. 이는 원래의 고울 프랍의 묶인 변수 m이 이미 S m'으로 특정되어 있기 때문이다. 우리는 묶인 변수 m이 '살아있기를' 원한다. 즉, 우리가 원하는 귀납가설은

```
IHn' : double n' = double (S m') -> n' = S m'
```

이 아니라

```
IHn' : forall m, double n' = double m -> n' = m
```

이다. 이렇게 된다면 (m := m')의 치환을 통하여 증명을 잘 진행할 수 있게 된다.

이를 위하여 라인 5의 intros n m을 intros n으로 바꿔 보자.

```
1  Theorem double_injective : forall n m,
2    double n = double m ->
3    n = m.
4  Proof.
5    intros n. induction n as [| n' IHn'].
6    - (* n = 0 *) simpl. intros m eq.
7      destruct m as [| m'] eqn:E.
8      + (* m = 0 *) reflexivity.
9      + (* m = S m' *) simpl in eq. discriminate eq.
10   - (* n = S n' *)
11     intros m eq.
12     destruct m as [| m'] eqn:E.
13     + (* m = 0 *) discriminate eq.
14     + (* m = S m' *) f_equal.
15       apply (IHn' m').
16       simpl in eq. injection eq as eq'.
17       exact eq'.
18 Qed.
```

라인 13을 실행한 후의 Coq Goals 화면은 다음과 같다.

```
n' : nat
IHn' : forall m : nat, double n' = double m -> n' = m
m, m' : nat
E : m = S m'
eq : double (S n') = double (S m')
(1 / 1) ------------------------
n' = m'
```

이번에는 IHn'의 m이 전칭묶인변수이므로, 라인 15에 의하여 m := m'의 치환이 일어난다.[7] 이하의 설명은 생략한다. ✓

요령 6.1 여러 개의 묶인변수를 가지는 전칭문의 증명에 induction을 사용하기 전에 필요이상으로 많은 변수를 intros 하지 말 것. ⊣

다음은 위의 요령을 적용한 예이다. SF의 연습문제이므로 답의 일부는 생략하였다.

```
1  Theorem eqb_true : forall n m,
2    n =? m = true -> n = m.
3  Proof.
4    intros n. induction n as [| n' IHn'].
5    - (* n = 0 *) intros m. destruct m.
6      + (* ... *)
7      + (* ... *)
8    - (* n = S n' *) intros m. destruct m as [| m'] eqn:Eq.
9      + (* m = 0 *) (* ... *)
10     + (* m = S m' *) intros.
11       assert (Ha: (n' =? m) = true).
12       { simpl in H. exact H. }
13       f_equal.
14       (* ... *)
15   Qed.
```

하나 더.

```
1  Theorem plus_n_n_injective : forall n m,
2    n + n = m + m ->
3    n = m.
4  Proof.
5    intros n. induction n as [| n' IHn'].
6    - intros. destruct m.
7      + (* ... *)
8      + (* ... *)
9    - intros. destruct m.
10     + (* ... *)
11     + simple in H.
12       rewrite <- plus_n_Sm in H. rewrite <- plus_n_Sm in H.
13       injection H as Hi. apply IHn' in Hi.
14       f_equal. exact Hi. Qed.
```

[요령 6.1]로도 해결되지 않는 경우가 있다. 귀납에 사용할 변수가 전칭문의 묶인변수들 중에 맨 앞에 위치하고 있지 않은 경우가 그것이다.

예를 들어 증명해야 할 명제가 forall (n: nat) (l: list nat), ... 형태인데 우리가 원하는 것이 induction n이 아니라 induction l인 경우이라면 어떻게 해야 할까?[8]

답은 generalize dependent라는 책략을 사용하는 것이다. 다음의 예를 통하여 이를 알아보도록 하자.

---

[7] 라인 15 대신 specialize IHn' with (m := m')를 사용해도 된다.

[8] 전칭한정사의 교환법칙을 이용하여 원래의 명제를 그것과 논리적으로 동등한 명제인 forall (l: list nat) (n: nat) , ... 로 바꾸는 것이 떠오르는데, 이것은 Coq에서 직접 지원하지 않는다.

```
1  Theorem nth_error_after_last : forall (n : nat) (X : Type) (l : list X),
2    length l = n -> nth_error l n = None.
3  Proof.
4    intros.
5    generalize dependent n.
6    induction l as [| h t IHt].
7    - intros. simpl. reflexivity.
8    - intros. destruct n as [| n'] eqn:Eq.
9      + simpl in H. discriminate H.
10     + simpl in H. injection H as Hi.
11       (* ... *)
12 Qed.
```

일단 라인 4까지 진행한다. 그러면 Coq Goals 화면은 다음과 같다.

```
n : nat
X : Type
l : list X
H : length l = n
(1 / 1) ------------------------
nth_error l n = None
```

여기서 우리는 n이 전칭묶인변수가 되기를 원한다. 이 작업을 해 주는 것이 라인 5이다. 그러면 Coq Goals 화면은 다음과 같이 바뀐다.

```
X : Type
l : list X
(1 / 1)
forall n : nat, length l = n -> nth_error l n = None
```

라인 10까지 마치면 귀납가설을 사용할 수 있게 된다. 그 다음은 생략한다. ✓

Unfolding Definitions

unfold 책략은 이전에 한 번 다룬적이 있으나 여기서 좀 더 상세히 알아보기로 한다.

우리가 증명에서 다루는 표현항은 생성자를 사용하는 귀납적 정의inductive definition로 정의된 경우에는 그 정의를 펼쳐서unfold 하여 사용하는 것이 더 편리한 경우가 많다. 예를 들어 square: nat -> nat이라는 함수를 다음과 같이 정의하였다고 하자.

```
Definition square (n : nat) : nat := n * n.
```

그리고 다음의 간단한 정리를 증명하고자 한다.

```
Theorem square_mult : forall n m,
  square (n * m) = square n * square m.
```

이 정리를 증명하기 위하여 intros n m를 실행하면 다음의 고울 화면을 얻는다.

```
n, m : nat
(1 / 1) ------------------------
square (n * m) = square n * square m
```

그 다음은 딱히 할 것이 없다. 이럴 때 square의 <u>정의를 펼치면</u> 다음과 같게 된다.

```
4 Proof.
5   intros n m. unfold square.
```

```
1 n, m : nat
2 (1 / 1) ------------------------
3 n * m * (n * m) = n * n * (m * m)
```

고울이 한결 다루기 쉬워졌다. 이제 mult_assoc과 mul_com을 사용하여 평이한 방법으로 증명을 마칠 수 있게 되었다. ✓

unfold 책략은 simpl이 작동하지 않을 때 해결책으로 흔히 사용된다. 예를 들어 bar: nat -> nat를 다음과 같이 (어리석게) 정의했다고 하자.

```
Definition bar (x: nat) : nat :=
  match x with
  | O => 5
  | S _ => 5
  end.
```

그리고 다음의 정리를 증명하고자 한다.

```
1 Theorem bar_idempotent : forall n,
2   bar (bar n) = bar n.
3 Proof.
4   intros n.
5   simpl. (* does nothing! *)
```

라인 4를 실행하면 고울명제는 다음과 같다.

```
bar (bar n) = bar n
```

이 상태로는 더 이상 할 것이 없다. 이때 unfold를 사용하면

```
6 unfold bar.
```

다음과 같이 된다. 조금 복잡해 보이지만 차분히 들여다 보면 어렵지 않다.

```
1 n : nat
2 (1 / 1) ------------------------
3 match match n with
4 | 0 | _ => 5
5 end with
6 | 0 | _ => 5
7 end = match n with
8 | 0 | _ => 5
9 end
```

라인 4, 6, 8에 보이는 | 0 | _ => 5는 다음의 표현을 줄여 쓴 것이다.

```
| O => 5
| S _ => 5
```

위에 보인 라인 3 – 라인 9의 고울은 n에 대한 정보가 없으므로 simpl이 듣지 않는다. 이럴 때 destruct를 사용하면 n = 0 또는 n = S _라는 정보를 가지게 되므로 simpl이 작동하게 된다.

destruct를 실행하면 이어서 자동으로 simpl을 불러 실행해 주므로 편하긴 한데, 이렇게 알아서 다 해주는 것은 증명과정을 이해하는 데는 오히려 방해가 되는 느낌이다. 완결된 증명을 처음부터 보이면 다음과 같다.

```
Theorem bar_idempotent : forall n,
  bar (bar n) = bar n.
Proof.
  intros n.
  unfold bar.
  destruct n as [| n'] eqn:Eq.
  - (* n = 0 *) reflexivity.
  - (* n = S n' *) reflexivity. Qed.
```

위의 증명에서 unfold bar를 사용하는 이유는, bar의 정의에 의하면 bar n의 값은 인수 n이 0인 경우와 S _인 경우 중 어느 것인지를 알아야만 계산을 진행할 수 있기 때문이다. 그러므로 destruct에 의한 case analysis를 활용하면 unfold bar 없이도 증명이 가능하다.

```
Theorem bar_idempotent' : forall n,
  bar (bar n) = bar n.
Proof.
  intros n.
  destruct n as [| n'] eqn:Eq.
  - simpl. reflexivity.
  - simpl. reflexivity.
Qed.
```

여기서 알아 두어야 할 것은 bar_idempotent의 증명은 unfold를 써도 되고 쓰지 않아도 가능하지만, square_mult의 증명은 unfold를 반드시 써야 하며 destruct 책략은 도움이 되지 않는다는 사실이다. 왜냐하면 square n의 값은 n = 0일 때와 n = S _로 나누어서 정의되어 있지 않기 때문이다.

Using destruct on Compound Expressions

destruct 책략은 destruct term ...의 형태로 사용되며 이때 term은 변수기호인 경우가 많다. 그러나 이때 term은 꼭 변수기호일 필요는 없고 *compound expression*, 즉 함수 표현항이나 생성자 표현항일 수도 있다. 예를 들어 destruct (n =? 3), 혹은 destruct (bin_to_nat b0) 등을 사용할 수 있다. 후자는 p 42에서 사용했었다.

*expr*가 compound expression인 경우 destruct는 expr의 값의 타입 T가 생성자를 사용하여 귀납적으로 정의된 경우 destruct *expr* 책략을 사용할 수 있다. 이때 Coq Goals에서 다음과 같이 거동한다.

> T의 각 생성자 c에 대하여 subgoal이 생성된다. 그리고 각 서브고울의 컨텍스트와 고울 프랍에서 나타나는 모든 *expr*는 c로 치환된다. c는 그것의 애리티에 따라 c v, c v1 v2 등의 형태를 가지게 된다.

destruct 책략은 destruct *expr* 구문으로 사용할 수도 있으나 통상 다음과 같이 intro pattern을 덧붙인 구문을 사용한다.

destruct *expr* as [(1) | (2) | .. | (k)] eqn:Eq

여기서 (i)는 i번째 생성자에 대한 변수들의 목록이다. 생성자가 상수인 경우에는 (i)는 공문자열로 두면 된다. 생성자의 애리티가 m_i인 경우 (i)는 $v_1\ v_2\ \ldots\ v_{m_i}$로 둔다. 여기서 v_j들은 사용자가 원하는 대로 이름지은 변수기호이다.

eqn:Eq에 의하여 각 컨텍스트에는 Eq : *expr* = c라는 가설이 도입된다. 이 등식의 우변에는 생성자의 인수를 덧붙여 주어야 한다. 예를 들어 생성자 c가 binary인 경우에는 이 가설은 Eq : *expr* = c v1 v2 형태가 된다.

as [..]를 생략하면 Coq이 임의로 적당한 이름을 지어주므로 증명에 큰 지장은 없으나, eqn:Eq를 생략하면 때로는 증명에 필요한 정보의 손실로 인하여 증명이 실패할 수도 있다. eqn:Eq가 증명에 사용되지 않는 경우에도 이를 써 주는 것이 증명의 완성도를 높여 주므로, 아주 간단한 증명이 아니라면, 될 수 있으면 이를 사용하도록 한다.

Exercises

① =?가 symmetric임을 증명한다. 이를 위하여 =?, 즉 eqb 관련하여 전에 증명해 두었던 두 정리를 사용한다.

```
Check eqb_true. (* forall n m, (n =? m) = true -> n = m *)
Check eqb_refl. (* forall n, (n =? n) = true *)

Theorem eqb_sym : forall (n m : nat),
  (n =? m) = (m =? n).
Proof.
  intros. destruct (n =? m) eqn:H.
  (* ... *)
```

n =? m이 true일 때와 false일 때로 나누어 생각하는 것이 당연하다. 그러므로 destruct (n =? m)을 사용했으며, 그러면 Coq Goals 화면은 다음과 같이 된다.

```
1  Goal 1
2  n, m : nat
3  H : (n =? m) = true
4  (1 / 2) ------------------------
5  true = (m =? n)
6
7  Goal 2
8  n, m : nat
9  H : (n =? m) = false
10 (2 / 2) ------------------------
11 false = (m =? n)
```

먼저 Goal 1을 보자. 라인 3이 H: (n =? m) = true이 아니라 H: n = m이었다면 rewrite H를 써서 뭔가 될 것 같다—eqb_true를 사용하면 Goal 1은 다음의 한 줄로 증명된다.

```
- apply eqb_true in H. rewrite H. symmetry. apply eqb_refl.
```

Goal 2는 이렇게 간단하게 증명되지 않는다. 왜냐하면 (n =? m) = false -> not (n = m) 같은 정리는 (아직) 없기 때문이다. 해결책은 다시 한 번 destruct를 사용하는 것이다.

destruct (m =? n) eqn:H1을 실행하면 Coq Goals 화면은 다음과 같이 바뀐다. 여기서 m =? n을 n =? m으로 잘못 쓰지 않아야 한다. m =? n을 사용해야 하는 이유는 이 표현이 고울 프랍에 등장하기 때문이다.

```
Goal 1
n, m : nat
H : (n =? m) = false
H1 : (m =? n) = true
(1 / 2) ------------------------
false = true

Goal 2
n, m : nat
H : (n =? m) = false
H1 : (m =? n) = false
(2 / 2) ------------------------
false = false
```

여기서의 Goal 2는 reflexivity로 간단하게 처리된다.

Goal 1에서는 고울 명제가 모순이므로 컨텍스트 내에서 모순을 찾아야만 한다. 라인 4의 H1과 eqb_true로부터 m = n이라는 가설을 얻어내고, 이를 이용하여 라인 3의 H의 좌변을 n =? n으로 대체한 다음, eqb_refl을 써서 모순된 가설 true = false를 컨텍스트에 도입할 수 있다. 이렇게 완성된 증명 전체를 아래에 보였다.

```
Proof.
  intros. destruct (n =? m) eqn:H.
  - apply eqb_true in H. rewrite H. symmetry. apply eqb_refl.
  - destruct (m =? n) eqn:H1.
    + apply eqb_true in H1. rewrite H1 in H.
      replace (n =? n) with (true) in H.
      { discriminate H. }
      { symmetry. apply eqb_refl. }
    + reflexivity.
Qed.
```

② 두 번째 문제를 보자. Poly에서 사용했던 두 함수의 정의를 아래에 다시 보였다.

```
Fixpoint combine {X Y : Type} (lx : list X) (ly : list Y)
           : list (X*Y) :=
  match lx, ly with
  | [], _ => []
  | _, [] => []
  | x :: tx, y :: ty => (x, y) :: (combine tx ty)
  end.

Fixpoint split {X Y: Type} (l: list (X*Y)) : (list X) * (list Y) :=
  match l with
  | [] => ([], [])
  | (x, y) :: t => (x :: fst (split t), y :: snd (split t))
  end.
```

이 두 함수는 대략 서로의 역함수라고 볼 수 있다.

다음의 정리는 (어떤 의미에서) combine이 split의 좌역함수left inverse임을 보여준다.[9]

```
Theorem combine_split : forall X Y (l : list (X * Y)) l1 l2,
  split l = (l1, l2) ->
  combine l1 l2 = l.
Proof.
  intros X Y. induction l as [| h t IHt].
  - intros l1 l2 H.
    simpl in H. injection H as H1 H2.
    rewrite <- H1. rewrite <- H2. simpl. reflexivity.
  - intros l1 l2 H.
    destruct h as [h1 h2] eqn:Eh.
    simpl in H.
    (* ... *)
    injection H as H1 H2.
    rewrite <- H1. rewrite <- H2. simpl. (* ... *)
    apply IHt. reflexivity.
Qed.
```

③ 다음은 위 정리의 역방향 정리인 split_combine이다. 즉 split이 combine의 좌역함수임을 증명한다. 다만 이 정리가 성립하기 위하여는 length l1 = length l2라는 조건이 추가되어야 한다. 증명을 위하여 먼저 보조정리 combine_nil을 증명한다.

```
Definition split_combine_statement : Prop :=
  forall X Y (l : list (X * Y)) l1 l2,
    length l1 = length l2 ->
    combine l1 l2 = l ->
    split l = (l1, l2).

Lemma combine_nil: forall X Y (l1: list X) (l2: list Y),
  length l1 = length l2 ->
  combine l1 l2 = [] ->
  (l1, l2) = ([], []).
Proof.
intros X Y l1 l2 H1 H2. induction l1.
- simpl in H1. destruct l2 as [| h t] eqn:El2.
  + reflexivity.
  + simpl in H1. discriminate H1.
- destruct l2 as [| h t] eqn:El2.
  + (* ... *) discriminate H1.
  + simpl in H2. (* ... *)
Qed.

Theorem split_combine : split_combine_statement.
Proof.
unfold split_combine_statement.
intros X Y. induction l as [| h t IHt].
- intros l1 l2 H. simpl. intros H1. symmetry.
  apply combine_nil. { exact H. } { exact H1. }
```

[9] 역함수란 좌역함수인 동시에 우역함수left inverse인 것이다. combine이 split의 좌역함수임을 말하는 등식은 combine (split l) = l이다. 그런데 이 등식은 타입 조건을 만족하지 못한다. 그래서 combine_split에서 사용한 조금 어색한 표현으로 combine이 split의 좌역함수임을 나타내었다.

```
27 - intros l1 l2 H. intros H1.
28   destruct h as [h1 h2].
29   destruct l1 as [| h1' l1'] eqn:Eql1.
30   + (* l1 = [ ] *)
31     simpl in H1. discriminate H1.
32   + (* l1 = h1' :: l1' *)
33     destruct l2 as [| h2' l2'] eqn:Eql2.
34     * (* ... *) discriminate H.
35     * (* ... *) injection H as Hi.
36       (* ... *) injection H1 as H1i1 H1i2 H1i3.
37       assert(Ha: split t = (l1', l2')).
38       { apply IHt.
39         - (* ... *)
40         - (* ... *)  }
41       simpl. rewrite Ha.
42       (* ... *)
43 Qed.
```

라인 36에서 H1이 3개의 가설로 분해되는 것에 유의하여라.

④ 다음의 문제는 비교적 간단하므로 설명은 생략한다.

```
Theorem filter_exercise : forall (X : Type) (test : X -> bool)
                                 (x : X) (l lf : list X),
    filter test l = x :: lf ->
    test x = true.
Proof.
  intros X test x l lf H.
  induction l as [| h t IHt].
  - (* l = [ ] *)
  - (* l = h :: t *)
    simpl in H. destruct (test h) eqn:Eq.
    + (* test h = true *)
      injection H as H1 H2. (* ... *)
    + (* test h = false *)
      (* ... *)
Qed.
```

⑤ 이번 문제는 내 생각으로는 가장 중요하다. 먼저 다음과 같은 성질을 가지는 함수

```
forallb: (X -> bool) -> list X -> bool
existsb: (X -> bool) -> list X -> bool
```

를 정의한다.

```
forallb odd [1;3;5;7;9] = true
forallb even [0;2;4;5] = false
forallb (whatever) [] = true

existsb (andb true) [true;false] = true
existsb (whatever) [] = false
```

이러한 성질을 가지는 forallb, existsb는 암기해 둘 가치가 있다.

```
Fixpoint forallb X : Type (test : X -> bool) (l : list X) : bool :=
  match l with
  | [] => true
  | h :: t => andb (test h) (forallb test t)
  end.

Fixpoint existsb X : Type (test : X -> bool) (l : list X) : bool :=
  match l with
  | [] => false
  | h :: t => orb (test h) (existsb test t)
  end.
```

그 다음은 existsb의 nonrecursive 버전 existsb'을 forallb와 negb를 이용하여 정의하고 이 둘이 동일한 함수임을 증명하여라. 이 문제는 $\exists x,\ P(x) \Leftrightarrow \neg\forall x,\ \neg P(x)$에 기반을 두고 있다.

```
Definition existsb' X : Type (test : X -> bool) (l : list X) : bool :=
  negb (forallb (fun x => negb (test x)) l).

Theorem existsb_existsb' :
  forall (X : Type) (test : X -> bool) (l : list X),
  existsb test l = existsb' test l.
Proof.
  intros. induction l as [| h l' IHl'].
  - unfold existsb'. simpl. reflexivity.
  - simpl. destruct (test h) eqn:H.
    + simpl. unfold existsb'. simpl. rewrite H.
      (* ... *)
    + simpl. rewrite IHl'. unfold existsb'.
      (* ... *)
Qed.
```

7 Logic

7.1 Propositions and Predicates

Proposition, 즉 프랍의 의미는 pp10–14에서 대략 설명하였다. 내용 중에 하나만 강조하자면, Coq의 프랍은 컨텍스트에 따라 동일한 표현이 프랍일 수도 있고 ill-typed expression일 수도 있다는 것이며, 이것이 Coq의 프랍과 수리논리학의 논리식을 구별짓는 가장 큰 특징이다.

프랍은 논리식과 마찬가지로 아톰 프랍atomic proposition에 결합자logical connective와 한정기호quantifier를 써서 얼마든지 복잡하게 만들 수 있다. 현재까지 우리가 사용한 프랍에서는 결합자로는 ->, 한정기호로는 forall만 등장했었다. 이 장에서는 나머지 결합자 and, or, not, iff, False, True와 한정기호 exists를 도입하여 다룬다.[1]

또한 아톰 프랍에 사용되는 술어predicate로는 현재까지는 등호 = 하나만 사용했었는데, 이 장에서는 다양한 술어를 정의하여 사용할 것이다.

등호의 타입을 알아보기 위하여 Check =이나 Check "="을 실행하면 에러가 난다. Print "="를 실행하면 다음과 같은 결과를 얻는다.

```
1 Print "=".
2 (* Inductive eq (A : Type) (x : A) : A -> Prop :=
3               eq_refl : x = x. *)
```

eq는 Coq의 표준 라이브러리에 정의되어 있는 2항 술어binary predicate이다. 그리고 =는 역시 표준 라이브러리에 Notation을 사용하여 정의된, eq의 설탕구문(*syntactic sugar*)이다.

eq는 암묵적 타입인수를 사용하는 다형 술어이므로 이것의 타입을 알아보려면 @를 eq에 앞붙여서prefix 다음과 같이 한다.

```
Check @eq. (* @eq : forall A : Type, A -> A -> Prop *)
```

라인 2–3과 같이 Inductive를 사용하여 술어를 정의하는 것은 Coq에서 가장 중요한 개념 중의 하나이며 이에 대해서 제8장에서 심도있게 공부하게 될 것이다.[2]

---

[1] False와 Truth는 그냥 프랍이라고 정의해도 되겠지만, 수리논리학에서는 False를 기호 ⊥로 나타내고 nullary connective로 취급하며, True는 ⊥ → ⊥의 줄여쓰기abbreviation으로 정의하는 것이 더 일반적이다.

[2] 지금까지 Inductive는 타입을 정의할 때 사용되었다. Enumerated type, type with parameter, recursive type, polymorphic type 등.

이제 술어에 대해서 알아보자. 술어는 인수를 받아서 프랍을 리턴하는 함수이다. 이것은 *parameterized proposition*이라고 말할 수 있다. 인수의 개수는 술어의 애리티arity이다. 이 장에서 다양한 술어를 정의하여 사용할 것이라고 하였는데, 이 술어들의 부분 표현에서 사용되는 기본 술어primitive predicate는 모두 등식이다. 등식이 아닌 기본 술어는 제8장에서부터 등장하게 된다.

간단한 술어의 예로 주어진 인수가 3과 같음을 말하는 1항 술어를 정의해 보자. 이 술어는 nat -> Prop 타입을 가진다. nat -> bool이 아님에 주의하라.

```
Definition is_three (n: nat): Prop :=
  n = 3.
Check is_three. (* is_three: nat -> Prop *)
```

이 새로운 술어 is_three를 사용하여 만든 정리를 보자.

```
1  Theorem one_plus_two_is_three :
2    is_three (1 + 2).
3  Proof.
4    simpl.
5    unfold is_three.
6    reflexivity. Qed.
```

라인 4와 라인 5는 둘 중 어느 하나, 혹은 둘 다를 생략해도 된다.

이번에는 다형 술어polymorphic predicate의 예를 보자.

```
Definition injective {A B: Type} (f: A -> B) :=
  forall x y: A, f x = f y -> x = y.
```

자연수의 바로뒤successor 함수 S가 1-1 함수임을 증명해 보자.

```
Check S. (* S: nat -> nat *)

Lemma succ_inj: injective S.
Proof.
  intros n m H.
  injection H as H1. exact H1.
Qed.
```

타입인수를 명시적으로 사용하고 싶다면 (실제로는 이럴 필요가 없겠지만) 다음과 같이 하면 된다.

```
Lemma succ_inj' :
  @injective nat nat S.
Proof.
  intros n m H. injection H as H1. exact H1.
Qed.
```

7.2 Logical Connectives

Conjunction

논리곱conjunction을 위한 결합자는 and이다. 그리고 이것의 설탕구문은 /\이다. 즉 P와 Q의 논리곱은 and P Q, 혹은 P /\ Q로 쓸 수 있다.

```
Check and. (* and : Prop -> Prop -> Prop *)
```

and는, 그것의 타입을 보면 2개의 프랍을 인수로 받아서 하나의 프랍을 리턴하는 함수인 것으로 보인다. 이것은 맞는 말이다. 그런데 and의 정의를 보면 다음과 같다.

```
1 Print and.
2 (* Inductive and (A B : Prop) : Prop :=
3      conj : A -> B -> A /\ B. *)
```

지금까지 함수는 Definition, 또는 Fixpoint를 사용하여 정의했었다. 그리고 Inductive는 타입을 정의하는 데 사용되는 것이었다. 그런데 여기서는 Inductive를 써서 함수 and를 정의했다고?

Inductive는 생성자를 사용하여 타입을 정의할 때 사용한다. 생성자는 타입의 원소inhabitant를 생성한다. 다음과 같은 구문을 사용한다.[3]

> Inductive *type_name parameters* : Type :=
>   *constr1 : type1;*
>   *constr2 : type2;*
>   ...

위의 Inductive 활용 구문에서 Type 대신 Set를 사용할 수도 있다. 이에 대해서는 p9에 예를 들어 설명하였으니 읽어 보기 바란다. 거기 보면 Type 대신 사용할 수 있는 것이 Set 외에 또 하나 있다는 것을 알 수 있다. 그것은 바로 Prop이다. 위에 보인 and의 Inductive 정의에서 Type 대신 Prop을 사용하였으며, 인수로 2개의 프랍을 받고 있으므로 and는 두 개의 프랍을 받아 하나의 프랍을 만들어내는 함수, 즉 결합자인 것이다.

and는 conj라고 하는 생성자 1개를 사용한다. 이것은 A 타입의 원소와 B 타입의 원소를 받아 and A B 타입의 원소를 리턴하는 함수이다. Prop 타입의 원소, 즉 프랍은 그것의 증명항proof term, or evidence인데 이 부분은 여기서 충분히 설명하기 힘들며 다음에 나올 8장과 9장에서 상세하게 설명할 것이다.

Inductive를 사용하여 인수(타입 인수, 혹은 원소 인수)를 받아서 프랍을 리턴하는 함수는 만들 수 있지만, 통상적인 Set 타입의 원소를 리턴하는 함수를 만들 수 없다. 그 이유는 Inductive는 타입을 정의하기 때문이다. (타입은 원소에 의하여 규정되며 원소는 생성자에 의해서 만들어진다.)

conj의 타입을 알아보면 다음과 같다.

```
Check conj. (* conj: forall A B : Prop, A -> B -> A /\ B *)
```

SF에는 Check conj 대신 Check @conj를 사용하고 있지만 이럴 필요가 없다. 이 두 명령은 동일한 결과를 보여준다. @를 앞붙이는 것은 그것이 암묵적 인수를 받을 때 이를 드러내기 위함인데 여기에는 암묵적 인수가 존재하지 않으므로 @를 사용하는 것은 (문법적 오류는 아니지만) 아무런 효과가 없다.[4]

[3] Inductive 구문은 실은 이것보다 훨씬 더 복잡하다. 예를 들어 *parameters*에 (T : Type)을 넣거나 Type을 nat -> Type으로 바꿀 수도 있고, *constr*의 콜론 왼쪽, 혹은 오른쪽에도 파라메터를 넣을 수도 있다. 이 부분에 대해서는 제8장에서 더 설명할 것이다.

[4] SF의 저자가 @를 사용함으로써 암묵적 인수가 존재하지 않음을 명확히 하려고 한 것인가 생각해 보았는데 이것도 옳지 않다. @를 사용하면 ① 암묵적 인수가 있다면 그들을 보여주는 효과가 있고, ② 암묵적 인수가 없다면 아무런 효과가 없다. 즉 암묵적 인수의 존재여부는 @를 사용함으로써 알 수 없다.

① 고울 프랍이 P /\ Q 형태일 때 이를 증명하려면 split 책략, 혹은 이와 동등한 apply conj 책략을 사용한다. 그러면 각각의 곱인자conjunct를 고울로 하는 2개의 하위 고울이 생성된다. 먼저 전자의 예를 보자.

```
1  Example and_exercise :
2    forall n m : nat, n + m = 0 -> n = 0 /\ m = 0.
3  Proof.
4    intros n m. split.
5    - (* 1st conjunct *) (* .. *)
6      + (* n = 0 *) reflexivity.
7      + (* n = S n' *) simpl in H. discriminate H.
8    - (* 2nd conjunct *) (* .. *)
9      + (* m = 0 *) reflexivity.
10     + (* m = S m' *) rewrite <- plus_n_Sm in H. (* .. *)
11   Qed.
```

라인 4의 split는 고울 프랍이 P /\ Q 형태인 경우, 이 고울을 곱인자인 P와 Q를 각각 고울로 하는 2개의 subgoal로 *split* 한다.

```
Goal 1
n, m : nat
H : n + m = 0
(1 / 2) -----------------------
n = 0

Goal 2
n, m : nat
H : n + m = 0
(2 / 2) -----------------------
m = 0
```

② split은 apply conj와 동등하다. 왜 그런지 알아보기 위하여 다음과 같은 정리가 있다고 가정하자.

```
Check AB_Lemma.
(* AB_Lemma: forall (A B : Prop) (f : Prop -> Prop -> Prop),  A -> B -> f A B *)
```

만일 고울이 B -> f A B라면, 이 경우에 apply AB_Lemma을 실행하면 고울이 A로 바뀔 것이다. 이는 A를 증명하면 고울의 조건을 충족하기 때문이다.

만일 고울이 f A B라면, 이 경우에 apply AB_Lemma을 실행하면, 고울의 조건을 충족하기 위하여 두 개의 프랍 A와 B가 필요하다. 따라서 이들을 각각 고울로 하는 2개의 subgoal을 생성하는 것이 타당하며 Coq은 정확히 이 작업을 해 준다.

AB_Lemma에서 f에 and를 대입하면 정확히 conj의 타입이 된다.

```
Check conj. (* conj: forall A B : Prop, A -> B -> A /\ B *)
```

따라서 고울이 A /\ B일 때 apply conj를 실행하면 고울의 조건을 충족하기 위하여 두 개의 프랍 A와 B가 필요하다. 이것이 split과 apply conj이 동등한 이유이다.

Inductive로 정의된 프랍의 생성자는 정리와 같은 방법으로 증명에 활용할 수 있다는 것은 반드시 알고 있어야 할 중요한 사실이다.

③ 고울 프랍이 논리곱일 때 어떻게 증명을 하는지를 알았으니, 이번에는 역으로 컨텍스트에 논리곱 가설conjunctive hypothesis `H : P /\ Q`가 있을 때 이를 증명에 활용하는 방법을 알아보자. 이 경우 우리는 `destruct H as ..`를 사용하여 각각의 논리곱인자conjunct를 가설로 도입할 수 있다.

```
1 Lemma proj1 : forall P Q : Prop,
2   P /\ Q -> P.
3 Proof.
4   intros P Q H.
5   destruct H as [HP HQ].
6   exact HP.  Qed.
```

라인 4를 실행하면 `Coq Goals` 화면은 다음과 같이 된다.

```
P, Q : Prop
H : P /\ Q
(1 / 1) ------------------------
P
```

이어서 라인 5를 실행하면 `Coq Goals` 화면은 다음과 같이 바뀐다.

```
P, Q : Prop
HP : P
HQ : Q
(1 / 1) ------------------------
P
```

그 다음은 설명이 필요 없을 것이다.

그런데 바로 위의 화면에서 `HQ : Q`는 증명에 사용되지 않으므로 거기 있을 필요가 없다. 그러므로 라인 4를 다음과 같이 하는 것이 더 나아 보인다.

```
destruct H as [HP _].
```

이렇게 하면 `HQ : Q`는 화면에 나타나지 않게 된다. 사실 가장 간편한 증명은 라인 4와 라인 5를 합쳐서 다음과 같이 하는 것이다.

```
intros P Q [HP _].
```

다음은 가설에 논리곱이 있고 또한 고울 프랍도 논리곱인 예로서 /\의 결합법칙을 보자. /\는 오른쪽 결합을 따른다는right associative 것에 유의하라.

```
1 Theorem and_assoc : forall P Q R : Prop,
2   P /\ (Q /\ R) -> (P /\ Q) /\ R.
3 Proof.
4   intros P Q R H.
5   destruct H as [HP HQR]. destruct HQR as [HQ HR].
6   (* .. snip snip .. *)
```

라인 5는 다음과 같이 한 줄로 합쳐서 쓸 수 있다.

```
destruct H as [HP [HQ HR]].
```

더 간편하게, 라인 4와 라인 5를 합쳐 다음과 같이 써도 된다.

```
intros P Q R [HP [HQ HR]].
```

Disjunction

논리합disjuntion을 위한 결합자는 or이다. 그리고 이것의 설탕구문은 \/이다. 즉 P와 Q의 논리합은 or P Q, 혹은 P \/ Q로 쓸 수 있다. \/는 /\와 같이 오른쪽 결합을 따른다.

\/의 우선순위는 /\보다 약하다. 예를 들어 P \/ Q /\ R은 P \/ (Q /\ R)를 의미하며, P /\ Q \/ R은 (P /\ Q) \/ R을 의미한다.

① 고울 P \/ Q를 증명하려면 left 혹은 right 책략을 쓰면 된다. 이 둘은 고울 P \/ Q를 각각 P와 Q로 바꿔준다.

```
1 Lemma or_intro : forall A B : Prop, (A -> (A \/ B)) /\ (B -> (A \/ B)).
2   Proof.
3     intros. split.
4     - intros. left. exact H.
5     - intros. right. exact H. Qed.
```

라인 3을 실행한 후에 Coq Goals 화면은 다음과 같다. 고울 프랍이 논리곱이므로 두 개의 하위 고울로 분리된 것이다.

```
Goal 1
A, B : Prop
(1 / 2) ------------------------
A -> A \/ B

Goal 2
A, B : Prop
(2 / 2) ------------------------
B -> A \/ B
```

라인 4에서 먼저 intros로써 가설을 도입하고 이어서 left를 써서 고울 프랍의 왼쪽 합인자를 뽑아내면 Coq Goals 화면은 다음과 같게 된다.

```
A, B : Prop
H : A
(1 / 1) ------------------------
A
```

더 이상의 설명은 필요 없을 것이다.

② 가설 H : P \/ Q를 사용하려면 destruct H as ..를 사용하여 2개의 subgoal로 나눠 증명하도록 한다. 각 subgoal에서 고울은 원래 것과 동일하고 가설은 P \/ Q가 각각 P와 Q로 바뀐다.

destruct 책략을 논리곱 가설에 실행했을 때는 가설 2개가 컨텍스트에 추가로 도입되었는데, 논리합 가설에 실행했을 때는 고울이 두 개로 분리되고 가설은 각 하위고울에 하나씩 도입됨에 유의하라.[5] 또한 새로 도입되는 가설의 이름을 지정하는 형식은 .. as [H1 H2]가 아니라 .. as [H1 | H2]임에 주목해야 한다.

[5]destruct는 컨텍스트에 나타난, 귀납으로 정의된 모든 타입의 원소에 대해서 적용할 수 있다. destruct의 효과가 무엇인지는 타입의 생성자들의 정의를 보면 알 수 있지만 현재는 그냥 각 타입마다 암기해 두도록 한다.

```
1 Theorem or_commut : forall P Q : Prop,
2   P \/ Q  -> Q \/ P.
3 Proof.
4   intros P Q H.
5   destruct H as [HP | HQ].
6   - right. exact HP.
7   - left. exact HQ. Qed.
```

라인 4와 라인 5를 합쳐서 다음과 같이 쓸 수 있다.

```
intros P Q [HP | HQ].
```

다음의 연습문제를 풀어보라.

```
Theorem impl_add_disjunct : forall P Q R: Prop,
  (P -> Q) -> (P \/ R -> Q \/ R).
```

Falsehood and Negation

부정결합자 ~와 부등호 <>는 다음과 같이 정의된다.

```
Definition not (P: Prop) := P -> False.
Check not : Prop -> Prop.
Notation "~ x" := (not x) : type_scope.
Notation "x <> y" := (~(x = y)).
```

False: Prop은 Coq의 키워드인 프랍으로서 이것의 증명은 존재하지 않는다. 가설에 False가 있으면 그때의 고울이 무엇이든지 상관없이 destruct 책략을 써서 증명이 마무리된다.

```
Theorem ex_falso_quodlibet : forall (P:Prop),
  False -> P.
Proof.
  intros P contra.
  destruct contra. Qed.
```

그런데 가설에 모순이 있는 경우의 증명에서는 그동안 discriminate를 써 왔다.

```
Fact contra_to_any: 0 = S 0 -> 2 + 2 = 5.
Proof.
  intros contra.
  discriminate contra. Qed.
```

모순되는 가설을 이용하는 증명에서 destruct를 쓰는 경우와 discriminate를 쓰는 경우가 있으므로 이 둘을 잘 구별해서 사용해야 한다. 전자는 False에 적용하고 후자는 생성자의 단사성에 어긋나는 등식에 적용한다.

False로부터 무엇이든 다 증명할 수 있다는 사실은 다음과 같이 Coq에서 표현하고 증명할 수 있다.

```
1 Theorem not_implies_our_not : forall (P : Prop),
2   ~ P -> (forall (Q : Prop), P -> Q).
3 Proof.
4   intros P H. intros Q HP.
5   unfold not in H.
6   apply H in HP. destruct HP.
7   Qed.
```

라인 4를 실행하면 Coq Goals 화면은 다음과 같다.

```
P : Prop
H : ~ P
Q : Prop
HP : P
(1 / 1) ------------------------
Q
```

가설이나 고울 프랍에서 ~ P를 만나면 이것에 unfold not을 사용하여 P -> False로 바꾸는 것이 증명에 도움이 되는 경우가 많다. 이제 라인 5 이하를 읽어 가면서 증명을 이해할 수 있을 것이다.

증명 과정에서 t1 <> t2를 만났을 때도 마찬가지로 unfold가 도움되며, 이때는 t1 <> t2가 t1 = t2 -> False로 바뀐다.

컨텍스트에 모순인 가설이 존재하거나 혹은 가설들을 엮어서 모순을 만들 수 있는 경우에는 exfalso 책략을 사용하는 것이 편한 경우가 많다.

```
1  Theorem not_true_is_false' : forall b : bool,
2    b <> true -> b = false.
3  Proof.
4    intros. unfold not in H.
5    destruct b.
6    - (* b = true *)
7      exfalso.
8      apply H. reflexivity.
9    - (* b = false *)
10     reflexivity.
11 Qed.
```

라인 5를 실행하면 Coq Goals 화면은 다음과 같이 된다.

```
Goal 1
H : true = true -> False
(1 / 2) ------------------------
true = false

Goal 2
H : false = true -> False
(2 / 2) ------------------------
false = false
```

Goal 2는 reflexivity로 간단하게 증명된다.

Goal 1은 고울이 모순인데, Coq에서 모순 고울을 증명하는 방법은 단 하나, 컨텍스트 내에 모순을 만들어 내는 것이다. 그런데 이때 만일 고울이 False라면 H의 후건과 일치하므로 apply H. reflexivity.를 사용할 수 있다.

exfalso는 고울을 False로 바꿔주는 기능을 하는 아주 단순한 책략인데 이 예에서 그 유용함을 알 수 있다. 이 책략의 타당성은 기본적인 명제논리에 의하여 확인된다.

참고로 exfalso를 쓰지 않고 위의 Goal 1을 증명하는 방법은 다음과 같다.

```
  (* ... *)
  - (* b = true *)
```

```
    assert (HF: False). { apply H. reflexivity. }
    destruct HF.
  (* ... *)
```

Truth

True는 Coq의 키워드인 프랍으로서 그 의미는 ~ False, 즉 '참'인 것으로 생각하면 된다. 이것은 False 만큼 자주 쓰이지는 않지만 때때로 유용하다. 참고로 True의 증명은 다음과 같이 한다.

```
Lemma True_is_true : True.
Proof. apply I. Qed. (* or exact I. *)
```

True를 활용하는 증명의 예로 다음을 보자. 이 증명은 실은 라인 10 직후에 discriminate H를 사용하면 간단히 마무리 될 것인데, 일부러 discriminate를 쓰지 않고 True를 활용하여 증명하는 방법이다.

```
1  Definition disc_fn (n: nat) : Prop :=
2    match n with
3    | 0 => True
4    | S _ => False
5    end.
6
7  Theorem disc_example : forall n, 0 <> S n.
8  Proof.
9    intros n. unfold not. intros H.
10   assert (H2 : disc_fn 0). { simpl. apply I. }
11   rewrite H in H2. simpl in H2. exact H2.
12 Qed.
```

참고로 라인 9를 실행했을 때 Coq Goals 화면은 다음과 같다.

```
n : nat
H : 0 = S n
(1 / 1) ------------------------
False
```

라인 12에서 rewrite H in H2를 실행하면 H2는 H2: disc_fn (S n)이 된다. 이걸 simpl하면 H2: False가 된다. 이제 exact H2를 실행하면 증명이 끝난다. ✓

Logical Equivalence

논리적 동등을 나타내는 "if and only if" 결합자 <->는 다음과 같이 정의되어 있다.

```
Print iff.
(* iff = fun A B : Prop => (A -> B) /\ (B -> A)
    : Prop -> Prop -> Prop *)
```

<->를 사용하는 예를 하나만 보자.

```
Lemma not_true_iff_false : forall b,
  b <> true <-> b = false.
Proof.
```

```
  intros b. unfold iff. unfold not. split.
  - (* -> *) intros H. destruct b.
    { exfalso. apply H. reflexivity. }
    { reflexivity. }
  - (* <- *) intros H. destruct b.
    { discriminate H. }
    { intros H2. discriminate H2. }
Qed.
```

apply를 쌍조건문biconditional 가설 H에 적용하면 H를 구성하는 두 개의 조건문 중에 결론에 부합되는 것을 찾아서 적용한다. 구태여 H를 두 개의 조건문 가설로 destruct할 필요가 없다.

Setoids and Logical Equivalence

<->는 =와 비슷한 부분이 있다. 동등관계라는 것이다. 그래서 다음 3개의 책략 rewrite, reflexivity와 symmetry를 iff문에 대해 사용할 수 있다.

```
1  Theorem iff_refl : forall P : Prop,
2    P <-> P.
3  Proof.
4    intros P. reflexivity.
5  Qed.
6
7  Theorem iff_sym' : forall P Q : Prop,
8    (P <-> Q) -> (Q <-> P).
9  Proof.
10   intros P Q H. symmetry. exact H.
11 Qed.
12
13 Theorem iff_trans : forall P Q R : Prop,
14   (P <-> Q) -> (Q <-> R) -> (P <-> R).
15 Proof.
16   intros P Q R H1 H2.
17   rewrite H1. rewrite H2. reflexivity.
18 Qed.
```

동등관계가 구비된 집합을 *setoid*라고 한다. Setoid에서는 증명에 반사성, 대칭성과 추이성을 활용할 수 있다.

7.3 Existential Quantification and More

Existential Quantification

짝수임을 나타내는 술어 Even을 다음과 같이 정의한다. 이전에 정의했던 double : nat -> nat를 이 정의에서 사용한다.

```
Definition Even x := exists n : nat, x = double n.

Print double. (* fix double (n : nat) : nat :=
  match n with | 0 => 0 | S n' => S (S (double n')) end *)
```

4가 짝수임을 말하는 프랍은 Even 4이다. 이를 증명해 보자.

① 고울이 존재한정문일 때 exists *term* 책략을 사용할 수 있다. 이때 사용된 *term*을 *witness* 라고 한다.

```
1 Lemma four_is_Even : Even 4.
2 Proof.
3   unfold Even.
4   exists 2. simpl. reflexivity.
5 Qed.
```

라인 2를 실행하면 Coq Goals 화면은 다음과 같다.

```
Goal 1
(1 / 1) ----------------------
exists n : nat, 4 = double n
```

exists 2는 존재문 고울 프랍의 witness로 2를 사용한다는 뜻이다.[6] exists 2를 실행하면 고울 프랍에서 한정사가 사라지고 n := 2의 치환이 이루어져 고울 프랍은 다음과 같이 된다.

```
4 = double 2
```

그 다음은 설명을 생략한다.

exists 책략이 specialize 책략과 어떻게 다른지 생각해 보라.

② 가설이 존재한정문일 때는 이를 고울의 증명에 이용하기 위하여 destruct 책략을 사용할 수 있다. 그런데 많은 경우 존재문 가설은 고울 프랍의 전건에서 intro되어 컨텍스트에 도입되는데 이때 destruct가 intros에 의하여 암묵적으로$_{\text{implicitly}}$ 실행된다.

일단 예를 하나 보자.

```
1 Theorem exists_example_2 : forall n,
2   (exists m, n = 4 + m) -> (exists o, n = 2 + o).
3 Proof.
4   intros n H.
5   destruct H as [c Hc]. (* intros [c Hc]. *)
6   exists (2 + c).
7   rewrite Hm. simpl. reflexivity.
8 Qed.
```

라인 4를 실행한 후의 Coq Goals 화면은 다음과 같다.

```
n : nat
H : exists m : nat, n = 4 + m
(1 / 1) ----------------------
exists o : nat, n = 2 + o
```

존재문 가설 H에 Rule-C, 혹은 *exists elim*을 적용해야 한다. 이를 위하여 라인 5에서 destruct를 실행하였다. 존재문 가설 H : exists m : nat, n = 4 + m은 어떤 m에 대하여 n = 4 + m이 성립한다는 것이다. 이를 만족하는 m을 c라고 하고, 치환 m := c에 의하여 얻어지는 새로운 가설 n = 4 + c의 이름을(정확히 말하자면 inhabitant를) Hc라고 하는 것이 destruct H as [c Hc]가 하는 일이다. 이 책략에 의하여 Coq Goals 화면은 다음과 같이 바뀐다.

[6]나라면 이 책략의 이름을 exists가 아니라 witness로 지었을 것이다.

```
n, c : nat
Hc : n = 4 + c
(1 / 1) ------------------------
exists o : nat, n = 2 + o
```

그 다음 이어지는 라인 6과 라인 7을 보면 증명 과정을 이해할 수 있을 것이다. ✓

그런데 앞서 언급했듯이, 존재문 가설 `H : exists m, n = 4 + m`는 고울

`(exists m, n = 4 + m) -> (exists o, n = 2 + o)`

의 전건이었던 것이 컨텍스트로 `intro`된 것이므로, `destruct`를 명시적으로 쓰지 않고 `intros`에 의하여 Rule-C까지 한 번에 실행할 수 있다. 이를 위하여 라인 4와 5를 다음과 같이 한 줄로 대체할 수 있다.

```
intros n [c Hc].
```

다음의 정리는 한정사에 대한 기본 지식으로 반드시 알아 두어야 할 것이다.[7]

```
1 Theorem dist_not_exists : forall (X : Type) (P : X -> Prop),
2   (forall x, P x) -> ~ (exists x, ~ P x).
3 Proof.
4   intros X P H. unfold not.
5   intros [c Hc].
6   specialize H with (x := c).
7   apply Hc. exact H. Qed.
```

라인 4를 실행한 후에 `Coq Goals` 화면은 다음과 같다.

```
X : Type
P : X -> Prop
H : forall x : X, P x
(1 / 1) ------------------------
(exists x : X, P x -> False) -> False
```

여기서 라인 5를 실행하면 다음과 같이 되고,

```
..
H : forall x : X, P x
c : X
Hc : P c -> False
(1 / 1) ------------------------
False
```

이어서 라인 6, 라인 7을 실행하면 증명이 완료됨을 알 수 있을 것이다. ✓

다음의 정리도 역시 꼭 알아 두어야 하는 내용이다. 비워 놓은 부분을 채워 넣길 바란다.

```
1 Theorem dist_exists_or : forall (X : Type) (P Q : X -> Prop),
2   (exists x, P x \/ Q x) <-> (exists x, P x) \/ (exists x, Q x).
3 Proof.
```

---

[7]참고로 역방향은 배중률을 사용하면 증명할 수 있다.
역방향의 약한 버전인 `~ (exists x, P x) -> (forall x, ~ P x)`는 배중률 없이도 증명할 수 있다.

```
4    intros X P Q. split.
5    - intros [x [HP | HQ]].
6      (* intros [x H]. destruct H as [HP | HQ]. *)
7      + (* ... *)
8      + (* ... *)
9    - intros H. destruct H as [HP | HQ].
10     + destruct HP as [x HPx]. (* ... *)
11     + (* ... *)
12 Qed.
```

라인 5는 그 아래 라인 6의 코멘트 안에 있는 내용을 줄여 쓴 것이다. 이후의 증명은 그리 어렵지 않으므로 생략한다.

연습문제 ① $\forall n, m \in \mathbb{N},\ (n \leqslant m \rightarrow (\exists x \in \mathbb{N},\ m = n + x))$

```
1  Theorem leb_plus_exists :
2    forall n m : nat,
3    n <=? m = true -> exists x, m = n + x.
4  Proof. intros n. induction n as [|n' IHn'].
5    - (* n = 0 *) intros m. simpl.
6      (* ... *)
7    - (* n = S n' *) intros m.
8      intros H.
9      destruct m as [| m'] eqn:Eq.
10     + (* Eq : m = 0 *) simpl in H. (* .. *)
11     + (* Eq : m = S m' *) simpl in H.
12       specialize (IHn' m').
13       (* ... *)
14       rewrite H. simpl. exists x. reflexivity.
15 Qed.
```

induction n에서 기저단계 n = 0는 쉬우니까 설명을 생략하고, 귀납단계 n = S n' 경우를 보도록 하자. 다음은 라인 8을 실행한 후의 Coq Goals 화면이다.

```
IHn' : forall m : nat, (n' <=? m) = true -> exists x : nat, m = n' + x
H : (S n' <=? m) = true
(1 / 1) --------------------
exists x : nat, m = S n' + x
```

IHn'의 바디의 전건 (n' <=? m) = true가 성립하도록 하는 m을 찾아, 이때 바디의 후건이 고울프랍과 일치한다면 좋을 것이다.

H : (S n' <=? m) = true를 보면 m = 0일 수는 없고, m = S m'이어야 함을 알 수 있다. 그러므로 라인 9에서 destruct m을 실행하였다. 라인 12를 실행한 후 Coq Goals 화면은 다음과 같다.

```
IHn' : (n' <=? m') = true -> exists x : nat, m' = n' + x
H : (n' <=? m') = true
(1 / 1) --------------------
exists x : nat, S m' = S n' + x
```

라인 13에는 적당한 커맨드 2개(apply와 destruct)를 넣어 주면 된다. ✓

연습문제 ②

그 다음은 위 정리의 역이다. 라인 8과 라인 13을 채워 보라.

```
Theorem plus_exists_leb : forall n m,
  (exists x, m = n + x) -> n <=? m = true.
Proof.
intros n. induction n as [| n' IHn'].
- intros m. intros H. simpl. reflexivity.
- intros m. destruct m as [| m'].
  + intros. destruct H as [x H].
    (* ... *)
  + intros H.
    specialize IHn' with (m := m').
    simpl.
    apply IHn'.
    (* ... *)
    injection H as H.
    exists x. exact H.
Qed.
```

연습문제 ③

destruct H as [x H].를 elim H. intros.를 바꿔서 증명을 진행해 보라. 그리고 elim 책략이 어떤 기능을 하는 것인지 설명하여라.

Programming with Propositions

지금까지 공부한 Coq의 문법과 책략들로써 다양한 명제들을 자유롭게 표현하고 증명할 수 있게 되었다. 예를 들어 리스트에 원소가 속함을 뜻하는 2항술어 In은 다음과 같이 정의할 수 있다.

```
Fixpoint In {A : Type} (x : A) (l : list A) : Prop :=
  match l with
  | [] => False
  | x' :: l' => x' = x \/ In x l'
  end.
```

or를 쓰지 않고 In을 정의하는 것은, A의 inductive 정의를 알고 있다면 또 하나의 match .. end를 써서 가능하다. 그러나 여기서는 A가 변수이므로 안 된다. 그러므로 다형 In의 정의에는 반드시 or와 같은 프랍에 대한 연산을 이용하여야 한다.

구체적인 리스트와 원소에 대해서 In이 성립함을 보이는 것은, 다음의 두 예에서 보듯이 쉽고 재미없다.

```
Example In_example_1 : In 4 [1; 2; 3; 4; 5].
Proof.
  simpl. right. right. right. left. reflexivity.
Qed.

Example In_example_2 :
  forall n, In n [2; 4] ->
  exists n', n = 2 * n'.
Proof.
  simpl.
  intros n. intros [H | [H1 | H2]].
  - exists 1. rewrite <- H. simpl. reflexivity.
```

```
  - exists 2. rewrite <- H1. simpl. reflexivity.
  - destruct H2.
Qed.
```

① 조금 더 일반적이고 흥미있는 예를 보자.

```
1  Theorem In_map :
2    forall (A B : Type) (f : A -> B) (l : list A) (x : A),
3      In x l -> In (f x) (map f l).
4  Proof.
5    intros A B f l x.
6    induction l as [ | x' l' IHl'].
7    - (* l = [] *)
8      intros. destruct H.
9    - (* l = x' :: l' *)
10     simpl. intros [H | H].
11     + rewrite H. left. reflexivity.
12     + right. apply IHl'. apply H.
13 Qed.
```

라인 10의 `intros [H | H]`는 다음을 줄여 쓴 것이다.

```
intros H. destruct H as [H | H].
```

② 또 하나의 흥미있는 예를 보자.

```
1  Theorem In_map_iff :
2    forall (A B : Type) (f : A -> B) (l : list A) (y : B),
3           In y (map f l) <->
4           exists x, f x = y /\ In x l.
5  Proof.
6    intros A B f l y. split.
7    - (* -> *)
8      induction l as [|x l' IHl'].
9      + (* base case *)
10       (* ... *)
11     + (* induction step *)
12       simpl.
13       intros [H1 | H2].
14       { (* H1 : f x = y *)
15         exists x. split.
16         - (* ... *)
17         - (* ... *)
18       }
19       { (* H2 : In y (map f l') *)
20         apply IHl' in H2.
21         destruct H2 as [x' [H21 H22]].
22         exists x'. split.
23         - (* ... *)
24         - (* ... *)
25       }
26   - (* <- *)
27     intros [x [H1 H2]].
```

```
28      rewrite <- H1.
29      apply In_map. exact H2.
30  Qed.
```

라인 11을 실행한 후의 고울 프랍은 다음과 같다. P \/ Q -> R 형태이다.

```
f x = y \/ In y (map f l') ->
  exists x0 : A, f x0 = y /\ (x = x0 \/ In x0 l')
```

이런 경우에 intros [H1 | H2]를 실행하면 2개의 하위 고울 P -> R과 Q -> R이 생성된다. 그리고 전자의 컨텍스트에는 H1 : P가, 그리고 후자의 컨텍스트에는 H2 : Q가 들어간다.

라인 21은 destruct H as [x' [H1 H2]] 형태인데 이러한 책략은 다음과 같은 형식의 가설 H에 대해서 적용한다.

```
H : exists x, P x /\ Q x
```

이러한 경우 컨텍스트에서 원래의 가설 H는 사라지고, 새로운 두 가설 H1 : P x'와 H2 : Q x'가 도입된다.

라인 27은 intros [x' [H1 H2]] 형태인데 이러한 책략은 다음과 같은 형식의 고울 프랍에 대하여 적용한다.

```
(exists x, P x /\ Q x) -> R
```

이러한 경우 고울 프랍은 R이 되고, 컨텍스트에는 H1 : P x'와 H2 : Q x'가 도입된다.

라인 21과 라인 27의 구문은 간편하게 사용할 수 있다는 장점이 있지만, 이러한 구문들 대신 몇 개의 책략을 결합하여 사용하면 동일한 결과를 얻을 수 있으므로, 암기하고 있지 않다고 해도 별 문제는 없다고 생각한다. ✓

③ 다음 예는 흥미롭고 또한 유용하다. 먼저 함수 All을 정의하는데 이것은 일종의 polymorphic quantifier로 볼 수 있다. 이 함수는 인수로 1항 술어 P와 리스트 l을 받아서 P가 l의 모든 원소에 대하여 성립함을 나타내는 프랍을 만들어 리턴한다. All의 정의에서 /\를 사용하였음에 주목하라.

```
Fixpoint All {T : Type} (P : T -> Prop) (l : list T) : Prop :=
  match l with
  | [] => True
  | x :: l' => P x /\ All P l'
  end.
```

이제 우리의 의도대로 All P l이 forall x, In x l -> P x와 동등한지를 증명해야 한다. 다음의 증명에서 라인 18 이전의 부분은 충분히 쉬우므로 설명하지 않는다. 라인 19와 라인 20은 intros를 고울 프랍의 형태에 따라 어떻게 사용해야 하는지를 보여준다. 이때 intros가 암묵적으로 destruct를 실행한다는 것을 이해해야 하며, 이렇게 하지 않고 명시적으로 destruct를 사용하는 방법도 사용할 줄 알아야 한다.

라인 22의 apply는 후건이 조건문의 전칭문인 조건문일 때 사용한 것인데, 이런 경우에 암묵적으로 split이 실행되어 2개의 subgoal이 생성되는 것은 전에 설명한 바 있다.

```
Theorem All_In :
  forall (T: Type) (P : T -> Prop) (l : list T),
    (forall x, In x l -> P x) <-> All P l.
Proof.
  intros T P l. split.
  - (* forward direction of <-> *)
    induction l as [|x l' IH].
    + (* base case *)
      intros. simpl. apply I.
    + (* induction step *)
      simpl. intros H. split.
      * apply H. (* ... *)
      * apply IH. (* ... *)
  - (* backward direction of <-> *)
    induction l as [|x l' IH].
    + (* base case *)
    (* ... *)
    + (* induction step *)
      simpl. intros [H1 H2]. (* goal: P /\ Q -> R *)
      intros x0 [Hg1 | Hg2]. (* goal: forall x, P x \/ Q x -> R x *)
      * (* ... *)
      * apply IH. (* IH: P -> forall x (Q x -> R x), goal: R x0 *)
        { exact H2. }
        { exact Hg2. }
Qed.
```

Applying Theorems to Arguments

정리는 다른 정리의 증명에서 가설처럼 사용할 수 있다. 그런데 때로는 정리를 사용할 때 직접 사용하지는 못하고 약간의 추가 작업을 해야 하는 경우가 있다. 예를 들어

```
forall x y z: nat, x + (y + z) = (z + y) + x
```

는 add_comm 정리를 두 번 사용하여 증명할 수 있어야 할 것이다. 그런데 다음과 같은 증명은 안 된다.

```
Proof.
  intros x y z.
  rewrite add_comm. (* ==> y + z + x = z + y + x *)
  rewrite add_comm. (* ==> x + (y + z) = z + y + x *)
Abort.
```

라인 4는 라인 3이 해 놓은 것을 되돌려 놓는다. 증명이 진행되지 않는다. 이 문제는 전에 다음과 같이 해결한 바 있다.[8]

```
Lemma add_comm3_take2 :
  forall x y z, x + (y + z) = (z + y) + x.
Proof.
  intros x y z.
  rewrite add_comm. (* ==> y + z + x = z + y + x *)
```

[8] rewrite 책략의 적용 순서order of application가 outermost-leftmost이기 때문에 이러하다.

```
6   assert (H : y + z = z + y).
7     { rewrite add_comm. reflexivity. }
8   rewrite H. (* ==> z + y + x = z + y + x *)
9   reflexivity. Qed.
```

더 좋은 방법이 있다. 정리에 인수argument를 주어 그 정리를 상황에 맞게 활용할 수 있다. 즉, 정리를 함수처럼 사용하는 것이다.

```
1 Lemma add_comm3_take3 :
2   forall x y z, x + (y + z) = (z + y) + x.
3 Proof.
4   intros x y z.
5   rewrite add_comm.
6   rewrite (add_comm y z).
7   reflexivity. Qed.
```

증명할 때 사용하는 정리에 인수를 주어 함수처럼 사용하는 방법에는 여러가지가 있을 수 있다. 이를 설명하기 위한 예를 곧 보일 것인데, 이 예에서 증명에 사용할 정리는 다음과 같다.

```
1 Theorem in_not_nil :
2   forall (A: Type) (x : A) (l : list A), In x l -> l <> [].
3 Proof.
4   intros A x l H. unfold not. intro Hl.
5   rewrite Hl in H.
6   simpl in H. destruct H. Qed.
```

위의 정리를 이용하여 다음을 증명하고자 한다. 이것은 간단하게 말하자면 위의 정리에서 `A:= nat, x:= 42`인 경우이다. 먼저 실패하는 경우를 보이겠다.

```
1 Example in_not_nil_42 :
2   forall l : list nat, In 42 l -> l <> [].
3 Proof.
4   intros l H.
5   Fail apply in_not_nil.
6 Abort.
```

라인 4를 실행한 후의 `Coq Goals` 화면은 다음과 같다.

```
l : list nat
H : In 42 l
(1 / 1) ------------------------
l <> []
```

고울 프랍이 `is_not_nil`의 바디의 후건과 일치하므로 `apply`를 사용할 수 있어야 할 것 같지만 실패한다. 이유는 이 후건을 가지고는 전건 `In x l`에서 `x`에 어떤 값을 넣어야 할지 알 수 없기 때문이다. 이를 해결하는 데는 여러 방법이 있다.

① 첫 번째 해법은 `x:= 42`를 `apply .. with`를 써서 알려주는 것이다.

```
1 Example in_not_nil_42_take2 :
2   forall l : list nat, In 42 l -> l <> [].
3 Proof.
4   intros l H.
5   apply in_not_nil with (x:= 42).
6   exact H. Qed.
```

이 증명이 작동하는 이유는 라인 5를 실행한 후의 Coq Goals 화면이 다음과 같기 때문이다.

```
l : list nat
H : In 42 l
(1 / 1) -----------------------
In 42 l
```

② 그 다음 방법은 apply .. in H를 사용하는 것이다.

```
1  Example in_not_nil_42_take3 :
2    forall l : list nat, In 42 l -> l <> [].
3  Proof.
4    intros l H.
5    apply in_not_nil in H.
6    exact H. Qed.
```

라인 4를 실행한 후에 컨텍스트에는 H : In 42 l이 들어 있다. apply의 인수로 사용되는 정리 in_not_nil의 프랍은 전칭조건문 forall l, In 42 l -> l <> []이다. apply의 타겟이 가설일 때는 조건문의 전건이 가설과 매치되어야 하며, 이때 가설이 조건문의 후건으로 바뀌는 것을 상기하자.

따라서 라인 5를 실행한 후의 Coq Goals 화면은 다음과 같다.

```
l : list nat
H : l <> []
(1 / 1) -----------------------
l <> []
```

③ 세 번째 방법은 정리를 함수로 보아 필요한 인수를 넣어 주는 것이다.

```
1  Example in_not_nil_42_take4 :
2    forall l : list nat, In 42 l -> l <> [].
3  Proof.
4    intros l H.
5    apply (in_not_nil nat 42 l).
6    exact H. Qed.
```

이 경우 라인 5를 실행한 후의 Coq Goals 화면은 Example in_not_nil_42_take2의 경우와 같다. 그런데 실은 라인 5에서 인수 l은 생략해도 된다. Coq가 알아서 찾아주기 때문이다.

정확히 말하면 이 경우 in_not_nil의 인수 3개 중에 꼭 필요로 하는 것은 x:= 42 하나뿐이다. 따라서 라인 5는 다음 중 어느 것으로 바꿔 사용해도 된다.

```
apply (in_not_nil nat 42).
apply (in_not_nil _ 42).
apply (in_not_nil _ 42 l).
apply (in_not_nil _ 42 _).
```

마지막으로 라인 6이 apply H 였어도 되는 것을 이용하여 라인 5와 라인 6을 합쳐 다음의 (1), 또는 (2)와 같이 만들어도 된다.

```
apply (in_not_nil _ 42 _ H).  (* (1) *)
apply (in_not_nil _ _ _ H).   (* (2) *)
```

(2)에 42가 들어있지 않아도 되는 이유는 H에 x:= 42라는 정보가 들어있기 때문이다.

7.4 Working with Decidable Properties

이 절에는 정확하지 않은 표현을 잘 해석해서 읽어야 하는 부분이 많이 있다.

우선 *decidable*의 의미를 정의하지 않고 사용하고 있다. 역사적으로 볼 때 이 용어는 적어도 3개의 의미를 가지고 있다.

하나는 함수에 대해서 사용되는 것으로 *computable*, 혹은 *recursive*와 동의어이다. 이것은 computability와 algorithm의 영역에 속하는 개념이다.

다른 하나는 명제에 대한 것으로, 그것의 증명이 존재하거나 혹은 그것의 부정의 증명이 존재하면 *decidable*하다고 한다.[9] 예를 들어 "I am not provable."을 뜻하는 arithmetical sentence는 undecidable하다. Classical logic에서는 undecidable sentence를 찾는 것이 대단히 어렵다. 괴델은 불완전성 정리를 통하여 undecidable sentence의 존재를 최초로 보여주었다. 배중률을 사용하지 않는 증명시스템에서는 undecidable sentence를 찾기가 아주 쉽다. 예를 들어 `forall P : Prop, P \/ ~ P`는 undecidable 하다. 이것은 provability의 영역에 속하는 개념이다.

증명은 기호를 사용하는 계산이므로 이 두 개념은 깊게 연관되어 있다. 하지만 분명 구별되는 개념임에도 불구하고 이 둘이 혼동하여 사용되는 경우가 많다. SF에서도 그러하다.

세 번째 의미는 수리논리학에서 논리식들의 집합, 즉 이론theory에 대해서 사용되는 것이다. 주어진 논리식이 그 집합에 속하는지의 여부를 결정하는 알고리듬이 존재할 때 그 이론은 decidable하다고 한다. 이것은 첫 번째 의미에 포함된다고 볼 수 있다.

———— ○ ————

SF는 다음과 같이 말한다.

> We've seen two different ways of expressing logical claims in Coq: with booleans (of type `bool`), and with propositions (of type `Prop`).

Logical claim은 무엇인가? 모든 프랍을 뜻하는 것일 것이다. 그리고 boolean으로 logical claim을 한다는 것은 *term* = `true` 혹은 *term* = `false` 형태의 프랍을 사용한다는 것을 뜻하는 것으로 보인다.

이어서 SF는 다음과 같이 말한다.

```
Here are the key differences between [bool] and [Prop]:

                                   bool     Prop
                                   ====     ====
decidable?                         yes       no
useable with match?                yes       no
works with rewrite tactic?         no        yes
```

이걸 보면 이 절의 제목을 "Comparing `bool` and `Prop`"로 했다면 어땠을까 생각이 든다.

① 첫 줄부터 살펴보자. 우선 프랍들 중에는, 그것의 증명도 존재하지 않고 그것의 부정의 증명도 존재하지 않는 것이 많이 있으므로 `Prop`은 undecidable이라고 말할 수 있을 것이다.

---

[9]'decidable'이라는 용어는 이러한 의미로는 20세기 전반에 수리논리학 분야에서 널리 사용되었지만 요즘에는 별로 사용하지 않는다.

프랍 중에 특수한 형태인 *term* = true와 *term* = false에 대해서 말하자면, 둘 중의 (단) 하나에는 반드시 증명이 존재한다. 왜냐하면, SF는 다음과 같이 말하고 있기 때문이다. (SF가 증명을 제공하지는 않았지만 나는 이것이 참이라고 믿는다.)

> The crucial difference between the two worlds is decidability. Every (closed) Coq expression of type bool can be simplified in a finite number of steps to either true or false — i.e., there is a terminating mechanical procedure for deciding whether or not it is true.

Boolean *term*의 계산은 decidable하므로 *term* = true, 혹은 *term* = false 형태의 프랍은 decidable하다. 바로 이 문장에는 'decidable'이라는 용어가 두 번 사용되었는데, 첫 번째 사용은 algorithmic 여부의 의미로, 그리고 두 번째 사용은 증명의 존재 여부의 의미로 사용되었다.

② 둘째 줄, "Usable with match?"에 대해서 bool은 yes, Prop은 no라고 되어있는데 여기서 오해의 소지가 있다. 예를 들어 앞서 다음과 같은 함수를 사용한 적이 있다.

```
1 Definition disc_fn (n: nat) : Prop :=
2   match n with
3   | O => True
4   | S _ => False
5   end.
```

이 예에서 보면 dess_fn이라는 Prop에 대해서 match를 사용했다.

저자가 말하고자 하는 것은 term: bool일 때는 match term with ...를 쓸 수 있다는 말이다. 그리고 예를 들어 t1 = t2: Prop일 때 match (t1 = t2) with ...는 쓸 수 없다는 말이다. (라고 생각한다.) 등식이 아닌 프랍에 대해서도 매치를 사용할 수 없는 것은 당연하다.

match 관련하여 하나 알아두어야 할 것은 if-then-else 구문은 match-with-end의 다른 표현이므로 역시 bool에서만 사용할 수 있다는 사실이다. 즉 다음의 두 표현에서 첫 번째 것만 well-formed expression이라는 것이다.

> if *boolean_term exp1* else *exp2*
> if *proposition exp1* else *exp2*

③ 그 다음 "works with rewrite tactic?"는 bool에서는 no, Prop에서는 yes라고 하는데 이것의 의미는, rewrite 책략을 사용할 수 있는 가설은 t1 = t2 형태의 프랍뿐이라는 것이다.[10]

42가 짝수임을 증명(or 확인)하는 다음의 두 방법을 비교해서 보라.

```
1 Example even_42_bool : even 42 = true.
2 Proof. simpl. reflexivity. Qed.
3
4 Example even_42_prop : Even 42.
5 Proof. unfold Even. exists 21. simpl. reflexivity. Qed.
```

우리는 직관적으로 even 42 = true와 Even 42가 동등하다는 것을 안다. 이 사실을 이를 다음의 정리 even_bool_prop에서 증명해 보았다. 증명에서 사용한 보조정리 even_double_conv는 일단 admit 하여 사용하고, 조금 후에 증명할 것이다.

[10] H : (t1 =? t2) = true는 H : t1 = t2와 동등하나 전자에 직접 rewrite를 사용할 수는 없고, 일단 H : t1 = t2를 얻은 다음에야 rewrite를 사용할 수 있다.

```
Lemma even_double_conv : forall n, exists k,
  n = if even n then double k else S (double k).
Proof. Admitted.

Theorem even_bool_prop : forall n,
  even n = true <-> Even n.
Proof.
  intros n. split.
  - intros H. unfold Even.
    destruct (even_double_conv n) as [k Hk].
    rewrite H in Hk.
    exists k. exact Hk.
  - unfold Even. intros [k Hk].
    rewrite Hk. apply even_double.
Qed.
```

라인 9를 실행한 후의 Coq Goals 화면은 다음과 같다.

```
n : nat
H : even n = true
(1 / 1) ------------------------
exists n0 : nat, n = double n0
```

가설 H : even n = true는 even_double_conv의 쌍조건문의 좌변이므로 Even n이 성립할 것이므로, Even n의 정의에 의하여 n = double k인 k를 얻은 다음 이것을 고울명제의 n0로 사용하면 된다. 이 작업은 라인 10 한 줄로 충분하다. 여기서 배울 것은 정리의 프랍이 전칭문일 때, 그 묶인변수를 n으로 특정하여 나타나는 존재문을 (even_double_conv n)으로 얻고, 이를 가설 취급하여 destruct 책략을 사용하는 기법이다. 눈여겨 두면 앞으로 도움이 될 것이다.

그 다음의 고울 화면은 다음과 같다.

```
H : even n = true
Hk : n = (if even n then double k else S (double k))
(1 / 1) ------------------------
exists n0 : nat, n = double n0
```

다음은 Hk의 부분 표현 even n을 H를 이용하여 true로 rewrite한다. 라인 11로 이를 실행하면 자동으로 simpl까지 실행되어 Hk : n = double k를 얻게 된다. 그 다음의 라인 12는 설명이 필요 없을 것이다.

역 방향은 아주 간단하므로 설명하지 않겠다. ✓

다음은 아까 미루어 두었던 보조정리 even_double_conv의 증명이다. even_S라는 정리를 증명에 사용한다. 아래의 증명에서 비워 놓은 부분은 쉽게 채울 수 있을 것이다.

```
Check even_S : forall n : nat, even (S n) = negb (even n).

Lemma even_double_conv : forall n, exists k,
  n = if even n then double k else S (double k).
Proof.
  intros. induction n as [|n' IHn'].
  - exists 0. simpl. reflexivity.
  - destruct IHn' as [k IHn'].
```

```
9      destruct (even n') eqn: H.
10     + exists k. (* ... *)
11       (* ... *)
12     + exists (S k). rewrite even_S. rewrite H.
13       simpl. f_equal. exact IHn'.
14 Qed.
```

짝수임evenness을 bool로 나타내면 even n = true이고, Prop으로 나타내면 Even n인데 이 둘이 동등함을 방금 증명하였다.

이와 비슷한 맥락에서, 같음equality의 개념은 bool과 Prop에서 각각 (n =? m) = true와 n = m으로 나타낼 수 있는데, 이 둘이 동등함을 다음의 정리 eqb_eq에서 증명하겠다.

<->의 forward 방향은 이전에 eqb_true에서 증명한 바 있다.

```
Check eqb_true: forall n m : nat, (n =? m) = true -> n = m.

Check eqb_refl: forall n : nat, (n =? n) = true.

Theorem eqb_eq : forall n1 n2 : nat,
  n1 =? n2 = true <-> n1 = n2.
Proof.
  intros n1 n2. split.
  - apply eqb_true.
  - intros H. rewrite H. rewrite eqb_refl. reflexivity.
Qed.
```

bool과 Prop은 어떤 때는 둘 중 어느 것을 써도 상관없지만, 또 어떤 때는 어느 한 쪽이 더 편리하다든가, 아예 어느 한 쪽만 가능한 경우도 있다.

다음의 사실은 bool의 장점을 말해준다.

> Coq's core language is designed so that every function it can express is computable and total.

그러므로 명제를 bool을 이용하여 나타내는 것은 그것의 증명이 Coq 표현항의 계산으로 환원되므로 이해하기 쉬운 단순한 구조를 가질 수 있다는 장점이 있다.

1000이 짝수임을 증명하는 예를 보자. 먼저 프랍을 사용하는 방법이다.

```
Example even_1000 : Even 1000.
Proof. unfold Even. exists 500. simpl. reflexivity. Qed.
```

1000이 짝수라는 사실은 Prop으로 표현하면 존재문이 된다. 이걸 증명하는 가장 자연스러운 방법은 witness 500을 찾는 것이다. 이것은 쉬운 예이었지만 일반적으로 witness를 찾는 작업은 쉽지 않은 경우가 많다. 그런데 bool을 사용하면 증명이 쉬워진다.

```
Example even_1000' : even 1000 = true.
Proof. simpl. reflexivity. Qed.
```

또 하나의 방법이 있다. 우리는 even n = true와 Even n이 동등함을 정리 even_bool_prop에서 증명하였다. 이를 이용한 다음의 증명을 보라.

```
Example even_1000'' : Even 1000.
Proof. apply even_bool_prop. reflexivity. Qed.
```

이러한 증명기법을 *proof by reflection*이라고 한다.

bool 방식의 또 하나 좋은 점은 부정negation을 다루기가 무척 쉽다는 것이다. 다음의 예를 보라.

```
Example not_even_1001 : even 1001 = false.
Proof.
  simpl. reflexivity.
Qed.
```

엄청 쉽다. 그런데 Prop 방식으로는 어떻게 할까? 다음의 증명을 보라.

```
1 Example not_even_1001' : ~ (Even 1001).
2 Proof.
3   rewrite <- even_bool_prop. (* even 1001 <> true *)
4   unfold not. (* even 1001 = true -> False *)
5   intro H. simpl in H. (* H : false = true *)
6   discriminate H.
7 Qed.
```

보다시피 reflection을 이용하면 비교적 쉽다.

하지만 reflection을 사용하여 고울 프랍을 boolean expression으로 바꾸지 않고 직접 증명하기는 쉽지 않다. 다음과 같이 해야 할 것이다: 우리는 Even은 $0, 2, 4, 6, \ldots$에 대해서 성립하고 ~ Even은 (짝수 $+ 1$)에 대해서 성립함을 안다. 따라서 ~ (Even 1001)을 증명하기 위하여는 Even 1000을 증명하고 또한

```
Lemma even_n_odd_Sn : forall n, Even n -> ~ (Even (S n)).
```

를 증명하면 될 것이다. 이 보조정리를 가정하면 다음과 같은 증명이 가능하다.

```
Example not_even_1001'' : ~ (Even 1001).
Proof.
  assert (H : Even 1000).
  { unfold Even.  exists 500. simpl. reflexivity. }
  apply even_n_odd_Sn. exact H.
Qed.
```

보조정리 even_n_odd_Sn을 증명하기 위하여는 다시 그 이전에 다음을 증명해야 한다: $n+2$가 짝수이면 n이 짝수이다.

```
1  Theorem even_SSn_even_n : forall n,
2    Even (S (S n)) -> Even n.
3  Proof.
4    intros n. intros H. unfold Even in H.
5    destruct H as [m H]. unfold Even.
6    destruct m as [| m'] eqn:Eq.
7    + (* m = 0 *) simpl in H. discriminate H.
8    + (* m = S m' *) exists m'. simpl in H.
9      injection H as H. exact H.
10 Qed.
```

라인 5까지 실행한 후의 Coq Goals 화면은 다음과 같다.

```
n, m : nat
H : S (S n) = double m
(1 / 1) ------------------------
exists n0 : nat, n = double n0
```

라인 6에서 destruct m을 하면 m = 0 경우는 간단하게 처리되고, m = S m' 경우는 라인 8까지 실행한 후에 고울 화면이 다음과 같이 된다.

```
n, m, m' : nat
Eq : m = S m'
H : S (S n) = S (S (double m'))
(1 / 1) ------------------------
n = double m'
```

그 다음은 쉬우므로 설명을 생략한다.

마지막으로 even_n_odd_Sn의 증명을 설명없이 아래 보였다.

```
1  Theorem even_n_odd_Sn : forall n,
2    Even n -> ~ (Even (S n)).
3  Proof.
4    intros n. induction n as [|n' IHn'].
5    - (* base case, n = 0 *)
6      intros H. unfold not. intros H1.
7      unfold Even in H1. destruct H1 as [m H1].
8      destruct m as [|m'] eqn:Eq.
9      + simpl in H1. discriminate H1.
10     + simpl in H1. injection H1 as H1. discriminate H1.
11   - (* induction step, n = S n' *)
12     intros H. unfold not. intros H1.
13     apply even_SSn_even_n in H1. apply IHn' in H1.
14     unfold not in H1. apply H1. exact H.
15 Qed.
```

물론 항상 bool이 Prop보다 좋은 것은 아니다. 일반적으로 (n =? m) = true는 n = m과 동등하기는 하지만 후자는 rewrite 책략을 쓸 수 있는데 반해 전자는 그렇지 못하다는 단점이 있다. Lemma eqb_eq를 써서 양쪽을 오가면서 상황에 따라 더 나은 쪽을 택하면 된다.

다음의 예에서는 boolean expression을 eqb_eq를 써서 등식 프랍으로 바꾼 다음 rewrite를 쓰면 증명이 간단히 해결되지만, 이 방법이 아니고는 증명이 쉽지 않을 것이다.

```
Lemma plus_eqb_example : forall n m p : nat,
  n =? m = true -> n + p =? m + p = true.
Proof.
  intros n m p H.
  rewrite eqb_eq in H.
  rewrite H.
  rewrite eqb_eq.
  reflexivity.
Qed.
```

<u>문제</u>. 타입의 원소 간의 값이 같음을 비교하는 부울값 함수 eqb로부터, 그 타입의 두 리스트가 주어졌을 때 대응하는 원소들 간에 모두 eqb가 성립하는지를 판단하는 부울값 함수 eqb_list를 정의하여라. 그리고 eqb_list l1 l2가 l1 = l2와 동등함을 증명하여라.

```
Fixpoint eqb_list {A : Type} (eqb : A -> A -> bool)
                  (l1 l2 : list A) : bool :=
  match l1, l2 with
  | [], [] => true
  | [], _ => false
  | _, [] => false
  | x1 :: l1', x2 :: l2' =>
    if eqb x1 x2 then eqb_list eqb l1' l2'
             else false
  end.
```

이것으로 eqb_list의 정의는 되었고, 이제 정리 eqb_list_true_iff를 증명해 보자. 여기서 일종의 *double negation elimination*을 사용할 것이다. 원래 직관주의 논리intuitionistic logic에서는

```
NNPP: forall P : Prop, ~ ~ P -> P
```

를 사용할 수 없지만, 우리는 다음과 같은 방법으로 직관주의 논리를 벗어나지 않고 이 문제를 우회할 수 있다.

```
(* Following is a constructive(or intuitionistic) version of
   the double_neg elim rule. *)
Lemma not_false_is_true: forall b: bool,
  (b = false -> False) -> b = true.
Proof.
  intros. destruct b eqn: H1.
  - reflexivity.
  - exfalso. apply H. reflexivity.
Qed.
```

메인 정리의 증명은 l1에 대한 귀납 및 l2에 대한 destruct로 증명하는데, l1 = []인 경우와 l2 = []인 경우는 쉬우므로 코드에서 생략하였다. 실질적인 증명은 라인 12부터 시작한다.

```
1  Theorem eqb_list_true_iff :
2    forall (A : Type) (eqb : A -> A -> bool),
3      (forall (a1 a2 : A), eqb a1 a2 = true <-> a1 = a2) ->
4      forall (l1 l2 : list A), eqb_list eqb l1 l2 = true <-> l1 = l2.
5  Proof.
6    intros A eqb H l1. induction l1 as [|x1 l1' IHl1'].
7    - (* l1 = [] *) intros. destruct l2 as [|x2 l2'].
8      (* ... *)
9    - (* l1 = x1 :: l1' *) intros. destruct l2 as [|x2 l2'].
10     + (* l2 = [] *) split.
11       (* ... *)
12     + (* l2 = x2 :: l2' *) split.
13       { (* -> *)
14         simpl. intros H1.
15         (* Will show [x1 = x2], and show [l1' = l2'] using IHl1'. *)
16         assert (H2: eqb x1 x2 = false -> False).
17         { intros Hh. rewrite Hh in H1. discriminate H1. }
18         apply (not_false_is_true (eqb x1 x2)) in H2.
19         rewrite H2 in H1.
20         apply H in H2. rewrite H2.
21         apply IHl1' in H1. rewrite H1.
```

```
22          reflexivity. }
23        { (* <- *)
24          simpl. intros H1.
25          (* Will show [eqb x1 x2 = true], and
26             show [eqb_list eqb l1' l2' = true] using IHl1'. *)
27          injection H1 as H11 H12.
28          apply H in H11. rewrite H11.
29          apply IHl1' in H12. exact H12. }
30  Qed.
```

라인 12를 실행한 후의 Cq Goals 화면은 다음과 같다. (지면을 절약하기 위하여 컨텍스트에서 타입선언들은 모두 생략하고 가설만 보였다.) 두 개의 서브고울에서 컨텍스트는 동일하고 고울 프랍들은 조건문이며 서로의 역이다.

```
Goal 1
H : forall a1 a2 : A, eqb a1 a2 = true <-> a1 = a2
IHl1' : forall l2 : list A, eqb_list eqb l1' l2 = true <-> l1' = l2
(1 / 2) ----------------------------------
eqb_list eqb (x1 :: l1') (x2 :: l2') = true -> x1 :: l1' = x2 :: l2'

Goal 2
H : forall a1 a2 : A, eqb a1 a2 = true <-> a1 = a2
IHl1' : forall l2 : list A, eqb_list eqb l1' l2 = true <-> l1' = l2
(2 / 2) ----------------------------------
x1 :: l1' = x2 :: l2' -> eqb_list eqb (x1 :: l1') (x2 :: l2') = true
```

라인 18의 의미를 설명하겠다. 라인 17까지 실행한 상태에서 컨텍스트에는 (여러 가설들 중에) 다음의 두 가설이 존재한다.

```
H1 : (if eqb x1 x2 then eqb_list eqb l1' l2' else false) = true
H2 : eqb x1 x2 = false -> False
```

여기서 라인 18을 실행하면 `H2`는 다음과 같이 바뀐다.

```
H2 : eqb x1 x2 = true
```

왜냐하면 `not_false_is_true`의 묶인 변수 `b`가 부울리언 표현항인 `eqb x1 x2`로 특정되기 때문이다.

7.5 The Logic of Coq

`Coq`의 논리적 기반은, 이 책의 첫 부분에서 말했듯이 `CoIC`(Calculus of Inductive Constructions)이다. 수학자들이 전통적으로 사용하는 논리적 기반인 ZFC(제르멜로-프랑켈 집합론(with Axiom of Choice))와 많은 부분을 공유하지만 다른 점도 상당히 많다.

`Coq`의 핵심 엔진은 아주 작다. 따라서 `Coq`에서의 수학적 증명은 처리해야 할 소소한 일들을 대단히 많이 필요로 하며, 이 일들을 모두 사용자가 한다는 것은 현실적으로 힘들다. 우리는 알맞은 공리와 정리들을 추가하여 이 문제를 해결할 수 있다.

Functional Extensionality

Functional Extensionality를 직역하면 '함수의 외연성' 정도가 될 것인데 이것은 ZFC에서 말하는 외연공리Axiom of Extension의 특수한 형태이다. 외연공리란 직관적으로 말해서 집합은 그것의 외연extension에 의해서 결정된다는 것이다. 1계 논리식으로는 다음과 같이 쓸 수 있다.

$$(\forall X)(\forall Y)\left(X = Y \;\leftrightarrow\; (\forall z)(z \in X \leftrightarrow z \in Y)\right)$$

함수는 집합의 일종이며 함수의 정의에 외연공리를 적용하면 다음과 같이 쓸 수 있다.[11]

$$(\forall f : X \to Y)(\forall g : X \to Y)\Big(f = g \;\leftrightarrow\; (\forall x \in X)\big(f(x) = g(x)\big)\Big) \tag{7.1}$$

이것을 Coq에서 공리로 사용하려면 다음을 실행하면 된다.

```
Axiom functional_extensionality :
  forall {X Y: Type} {f g : X -> Y},
    (forall (x: X), f x = g x) -> f = g.
```

어떤 prop을 Axiom이라고 선언해 두면 그것을 증명에서 자유롭게 사용할 수 있다. Theorem과의 차이는 Axiom은 증명을 할 필요가 없다는 것이다.

다음은 functional_extensionality 사용의 예이다. 이 정리는 functional_extensionality 없이는 증명할 수 없다.

```
1 Example function_equality_ex2 :
2   (fun x => plus x 1) = (fun x => plus 1 x).
3 Proof.
4   apply functional_extensionality.
5   intros x. apply add_comm.
6 Qed.
```

라인 4를 실행한 후의 Coq Goals 화면이 다음과 같다는 것만 알면 이 증명을 쉽게 이해할 수 있을 것이다.

```
(1 / 1) ------------------------
forall x : nat, x + 1 = 1 + x
```

Axiom을 추가할 때는 무모순성consistency를 깨지 않도록 대단히 조심해야 한다. 참고로 공리계의 무모순성 여부를 판단하는 알고리듬은 없다는 것이 증명되어 있다.

정리의 증명에 어떤 공리와 보조정리들이 사용되었는지를 아는 간편한 방법이 있다.

```
Print Assumptions name_of_theorem.
```

name_of_theorem에 function_equality_ex2를 넣어 실행해 보아라.

함수의 외연성을 이용하는 문제를 하나 더 풀어 보자. 우리가 사용하는 리스트 뒤집기 함수 rev는 다음과 같이 정의되어 있다.

---

[11] 물론 이것은 ZFC에서의 얘기다. 하지만 ZFC에서 사용하는 함수의 엄격한 정의를 몰라도 (7.1)을 이해하는 데 지장이 없을 것이다.

```
Fixpoint rev {X : Type} (l : list X) : list X :=
  match l with
  | nil      => nil
  | cons h t => app (rev t) (cons h nil)
  end.
```

코드를 잘 보면 이건 대단히 비효율적이라는 것을 알 수 있다. 예를 들어 길이 100의 리스트를 뒤집을 때 rev는 길이 99의 리스트를 뒤집은 다음에 app을 길이 99의 리스트와 길이 1의 리스트에 대해서 실행한다. 길이 99의 리스트는 다시 rev를 호출하여 길이 98의 리스트를 뒤집은 다음에 ... 이런 식이다.

rev의 정의에 따른 계산이 비효율적인 문제는 정의 내에서 사용하는 함수 app의 첫 인수 rev t의 길이가 길기 때문에 발생한다. 다음의 정의가 더 낫다. 이것은 두 단계에 걸쳐 tr_rev를 정의한다.

```
1 Fixpoint rev_append X (l1 l2 : list X) : list X :=
2   match l1 with
3   | [] => l2
4   | x :: l1' => rev_append l1' (x :: l2)
5   end.
6
7 Definition tr_rev X (l : list X) : list X :=
8   rev_append l [].
```

rev_append는 app과 달리 인수들의 길이에 상관없이 실행에 걸리는 시간은 일정하게 짧다. 이제 우리가 해야 할 일은 rev와 tr_rev가 동일한 함수라는 사실을 증명하는 것이다.

다음의 정리는 이 문제를 해결한다.

```
1 Theorem tr_rev_correct : forall X, @tr_rev X = @rev X.
2 Proof.
3   intros. apply functional_extensionality.
4   intros l. (* use l instead of x for readability *)
5   unfold tr_rev.
6   rewrite -> rev_append_correct.
7   apply app_nil_r.
8 Qed.
```

라인 6과 라인 7에서 사용한 보조정리들은 다음과 같다.

```
1 Lemma rev_append_correct :
2   forall (X: Type) (l1 l2: list X),
3   rev_append l1 l2 = rev l1 ++ l2.
4 Proof.
5   intros.
6   generalize dependent l2.
7   induction l1 as [| h1 t1 IH1].
8   - (* l1 = nil *) intros.
9     simpl. reflexivity.
10  - (* l1 = h1 :: t1 *) intros.
11    (* ... *)
12    rewrite <- app_assoc.
13    (* ... *)
```

```
14 Qed.
15
16 Check app_nil_r : forall (X : Type) (l : list X), l ++ [ ] = l.
```

이 문제의 풀이에서 핵심이 되는 부분은 rev_append_correct라는 유용한 보조정리를 찾아내는 것이다. 일단 이 보조정리의 명제를 찾았다면 이를 증명을 하는 것은 그리 어렵지 않다. 이런 보조정리를 발견하기 전에 그럴듯한, 도움이 될 듯한 보조정리를 생각해 내고 증명했다 해도 실제 정리의 증명에는 도움이 되지 않는 경우가 많다. 이것이 바로 수학자가 업으로 삼고 하는 proof search의 과정이다.

Classical vs. Constructive Logic

고전 논리Classical logic와 생성형 논리Constructive logic(또는 직관주의 논리)는 다음의 프랍을 공리로 채택하는지의 여부에 따라 구분된다.[12]

```
Definition excluded_middle := forall P : Prop,
  P \/ ~ P.
```

excluded_middle은 흔히 LEM(*Law of Excluded Middle*)이라고 부른다.

Coq에서 증명을 하다보면 때로는 LEM이 꼭 필요한 것처럼 보이는 경우가 있다. Coq의 디폴트는 LEM을 사용하지 않으므로 이제 증명할 길이 막혔다고 생각할 수 있는데, 많은 경우 다음의 보조정리를 사용하여 이 문제를 우회할 수 있다.

```
1 Lemma restricted_excluded_middle : forall P b,
2   (P <-> b = true) -> P \/ ~ P.
3 Proof.
4   intros P [] H.
5   (* intros P b H. destruct b. *)
6   - left. rewrite H. reflexivity.
7   - right. rewrite H. intros contra. discriminate contra.
8 Qed.
```

라인 4는 라인 5를 줄여쓴 것이다.

LEM은 전에 한 번 다루었던 NNPP(double negation elimination)와 논리적으로 동등하다고 볼 수 있다. 이에 관련된 정리들을 조금 후에 살펴볼 것이다. NNPP의 restricted version은 이전에 not_false_is_true라는 이름의 보조정리에서 증명하였다.

LEM에 대한 restricted_excluded_middle은 NNPP에 대한 not_false_is_true와 같다고 볼 수 있다. SF는 restricted_excluded_middle을 활용하는 예로 다음을 제시하고 있다.

```
1 Theorem restricted_excluded_middle_eq : forall (n m : nat),
2   n = m \/ n <> m.
3 Proof.
4   intros n m.
5   apply (restricted_excluded_middle (n = m) (n =? m)).
6   symmetry.
7   apply eqb_eq.
8 Qed.
```

[12] Constructive logic과 Intuitionistic logic의 정의는 사람에 따라 조금씩 다르지만 이 책에서는 이 둘을 같은 것으로 여기기로 한다.

라인 5에서 restricted_excluded_middle의 묶인 변수들은 P:= (n = m), b: = (n =? m)로 매치된다. 고로 고울 프랍 n = m \/ n <> m는 restricted_excluded_middle의 전건인

```
n = m <-> (n =? m) = true
```

로 바뀌게 된다. 그 다음은 쉽다. ✓

그렇다면 restricted_excluded_middle을 써서 증명할 수 있는 명제는 반드시 P \/ ~ P 형태여야 하는가? 그렇지 않다는 것을 다음에 설명하였다.

술어predicate P가 인수를 (x: T) 하나만 가지는 경우를 예로 들어 보자. 우리가 증명하고자 하는 명제가 forall (x: T), Q x 형태라고 가정하자. 또한 (P x -> Q x) /\ (~ P x -> Q x)를 쉽게 증명할 수 있다고 하자. 이렇게 경우를 나누어 Q x를 증명하는 예는 수학에 아주 흔하다.

이런 조건이 갖추어졌을 때, 먼저 다음과 같은 성질을 가지는 부울값 함수 f: T -> bool를 찾아야 한다.

```
f(x) = true <-> P x
```

그 다음은 restricted_excluded_middle을 apply해서 forall x, P x \/ ~ P x를 얻는다. 그 다음은 설명을 생략한다.

LEM(Law of Excluded Middle)이라는 이름은 아마도 P \/ ~ P와 논리적으로 동등한 ~ (P /\ ~ P)에서 온 것이 아닌가 한다. P and not P를 P와 not P의 교집합middle이라고 본다면 이 교집합이 공집합이라고 말하는 것이 LEM이다. LEM은 흔히 우리말로 배중률이라고 하지만 이것보다는 양자택일이 낫다고 본다.

Constructive logic이라는 이름은 $\exists x,\ P(x)$의 증명으로 $P(t)$가 성립하는 표현항, 즉 *witness* t를 construct 하는 것만 받아들이도록 되어 있기 때문에 붙여진 것이다. Classical logic에서는 $\exists x,\ P(x)$가 주어졌을 때 $\neg(\exists x,\ P(x))$를 가정하고 모순을 유도하는 것도 증명으로 받아들이지만, 이 방법(부정을 가정하고 모순을 유도하여 긍정을 증명하는 NNPP, 혹은 이와 동등한 LEM)은 constructive logic에서는 사용할 수 없다.

참고로 P \/ ~ P는 증명할 수 없어도 이것의 double negation은 다음과 같이 증명이 가능하다.

```
1 Theorem excluded_middle_irrefutable : forall (P : Prop),
2   ~ ~ (P \/ ~ P).
3 Proof.
4   intros P. unfold not.
5   intros H.
6   apply H. right. intros H1.
7   apply H. (* ... *)
8 Qed.
```

이 정리를 이용하여 Coq에 LEM을 추가하여도 된다는 것, 즉 Coq이 원래 모순을 증명하지 않는다면, (Coq + LEM)도 모순을 증명하지 않는다는 것을 보일 수 있다.

만일 P \/ ~ P를 넣어서 모순이 발생한다면, 즉 False를 증명할 수 있다면 Coq는 ~ (P \/ ~ P)를 증명한다는 얘기다. 그런데 Coq은 excluded_middle_irrefutable에 의하여 ~ (~ (P \/ ~ P))를 증명한다. 이렇게 증명한 두 명제로부터 False를 얻을 수 있다. 즉 Coq는 본래 모순적이다. ✓

Equivalent forms of LEM

다음의 프랍들은 모두 동등하다.

```
Definition excluded_middle := forall P : Prop,
  P \/ ~ P : Prop

Definition peirce := forall P Q : Prop,
  ((P -> Q) -> P) -> P.

Definition double_negation_elimination := forall P : Prop,
  ~ ~ P -> P.

Definition de_morgan_not_and_not := forall P Q : Prop,
  ~ (~ P /\ ~ Q) -> P \/ Q.

Definition implies_to_or := forall P Q : Prop,
  (P -> Q) -> (~ P \/ Q).
```

증명은 특별할 것 없이 평이하다.

```
Theorem LEM_to_Peirce : excluded_middle -> peirce.
Proof.
  unfold excluded_middle. unfold peirce.
  intros H. intros  P Q.
  intros H1.
  specialize H with (P:= P).
  (* ... *)
  - exact H2.
  - apply H1. intros H4. unfold not in H3.
    (* ... *)
Qed.

Theorem Peirce_to_NNPP :
  peirce -> double_negation_elimination.
Proof.
  (* ... *)
  (* ... *)
  apply H with (Q:= False).
  intros H2. exfalso. apply H1.
  (* ... *)
Qed.

Theorem NNPP_to_de_morgan_not_and_not :
  double_negation_elimination -> de_morgan_not_and_not.
Proof.
  (* ... *)
  intros H. intros P Q H1.
  apply H. unfold not. intros H2.
  unfold not in H1. apply H1.
  split.
  - intros H3. (* ... *)
  - intros H3. (* ... *)
```

```
Qed.

Theorem de_morgan_not_and_not_to_implies_to_or :
  de_morgan_not_and_not -> implies_to_or.
Proof.
  (* ... *)
  intros H. intros P Q H1.
  apply H. unfold not. intros H2.
  destruct H2 as [H3 H4].
  apply H3. intros H5.
  (* ... *)
Qed.

Theorem implies_to_or_to_LEM :
  implies_to_or -> excluded_middle.
Proof.
  (* ... *)
  intros H. intros P.
  specialize H with (P:= P) (Q:= P).
  assert (~ P \/ P).
  { (* ... *) }
  destruct H0 as [H1 | H2].
  - (* ... *)
  - (* ... *)
Qed.
```

LEM과 동등한 3개의 프랍들을 소개한다.

```
Definition implies_to_or2 := forall P Q : Prop,
  (~ P -> Q) -> (P \/ Q).

Definition contrapositive2 := forall P Q : Prop,
  (~ P -> ~ Q) -> (Q -> P).

Definition de_morgan_not_or_not := forall P Q : Prop,
  ~ (~ P \/ ~ Q) -> P /\ Q.
```

위의 세 프랍이 모두 LEM과 동등함을 증명하는 것은 연습문제로 남겨 놓겠다.

이 프랍들의 역은 LEM 없이 증명할 수 있다. 이 점은 앞에 소개한 5개의 프랍들 중 전칭조건문 형태인 3개에 대해서도 마찬가지다. 즉, 모든 프랍 `P`, `Q`에 대해서 다음은 LEM을 사용하지 않고도 증명할 수 있다.

```
(P \/ Q) -> (~ P -> Q)
(Q -> P) -> (~ P -> ~ Q)
P /\ Q -> ~ (~ P \/ ~ Q)
P -> ~ ~ P
P \/ Q -> ~ ( ~ P /\ ~ Q)
( ~ P \/ Q) -> P -> Q
```

참고로, 수학자가 `P \/ Q` 형태의 명제를 증명할 때, 거의 모든 경우 `implies_to_or2`를 사용한다. 그러니까 실질적으로는 LEM을 사용하는 것이다.

Using LEM in Proofs

일반적으로 Q : Prop의 증명에 LEM을 사용하려면 excluded_middle -> Q를 증명하면 된다. 이와는 달리 Coq의 표준 라이브러리를 사용하는 방법도 있으며 이제 이 방법을 소개하겠다.

전에 다음의 프랍을 증명한 적이 있다.

```
Definition dist_not_exists : forall (X : Type) (P : X -> Prop),
    (forall x, P x) -> ~ (exists x, ~ P x).
```

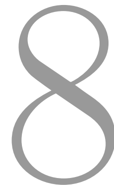
이것의 역은 LEM 없이는 증명할 수 없다. 다음은 LEM을 사용한 증명이다.

```
1  From Coq Require Import Classical_Prop.
2
3  Theorem dist_not_exists_inv : forall (X : Type) (P : X -> Prop),
4    ~ (exists x, ~ P x) -> (forall x, P x).
5  Proof.
6    intros X P H x.
7    unfold not in H.
8    destruct (classic (P x)) as [HPx | HNPx].
9    - exact HPx.
10   - exfalso. apply H. exists x. exact HNPx.
11 Qed.
```

라인 8에 의하여 서브고울 1에는 HPx : P x가, 서브고울 2에는 HNPx : ~ P x가 컨텍스트에 가설로 도입된다.

Classical_Prop을 Import한 다음에는 LEM 대신 NNPP를 사용할 수도 있다.

```
Check NNPP. (* NNPP : forall p : Prop, ~ ~ p -> p *)
```



Inductively Defined Propositions

8.1 Inductively Defined Predicates

The Collatz Conjecture

먼저 함수 div2: nat -> nat와 f: nat -> nat를 다음과 같이 정의하자.

```
Fixpoint div2 (n : nat) :=
  match n with
    0 => 0
  | 1 => 0
  | S (S n) => S (div2 n)
  end.

Definition f (n : nat) :=
  if even n then div2 n
  else (3 * n) + 1.
```

div2 n은 n을 2로 나눈 몫이라고 생각하면 된다. 주어진 n을 초항으로 하고 f를 반복적용하여 다음과 같은 수열을 얻을 수 있다.

$$n, f(n), f^2(n), f^3(n), \ldots$$

이런 수열의 예를 만들어 보기 위하여 다음의 함수를 사용한다.

```
(* Get a sequence of length n by repeatedly applying f to n0. *)
Fixpoint get_collatz_sequence (n0 n: nat) : list nat :=
  match n with
  | 0 => []
  | S n' => n0 :: get_collatz_sequence (f n0) n'
  end.
```

길이 10의 수열을 $n_0 = 1, 2, \ldots, 7$에 대하여 만들어 보면 다음과 같다.

```
Compute get_collatz_sequence 1 10.
(* ==> [1; 4; 2; 1; 4; 2; 1; 4; 2; 1] *)
Compute get_collatz_sequence 2 10.
(* ==> [2; 1; 4; 2; 1; 4; 2; 1; 4; 2] *)
```

```
Compute get_collatz_sequence 3 10.
(* ==> [3; 10; 5; 16; 8; 4; 2; 1; 4; 2] *)
Compute get_collatz_sequence 4 10.
(* ==> [4; 2; 1; 4; 2; 1; 4; 2; 1; 4] *)
Compute get_collatz_sequence 5 10.
(* ==> [5; 16; 8; 4; 2; 1; 4; 2; 1; 4] *)
Compute get_collatz_sequence 6 10.
(* ==> [6; 3; 10; 5; 16; 8; 4; 2; 1; 4] *)
Compute get_collatz_sequence 7 17.
(* ==> [7; 22; 11; 34; 17; 52; 26; 13; 40; 20; 10; 5; 16; 8; 4; 2; 1] *)
```

초항이 1∼6일 때는 수열의 항 중에 1이 나타나는 것을 볼 수 있다. 초항이 7일 때는 1이 길이 10의 수열에는 나타나지 않으며 길이 17에 이르러서야 1이 나타남이 관측되었다.

*Collatz Conjecture*는 모든 초항 $n_0 > 0$에 대해서, 위와 같은 방법으로 얻은 수열에는 1이 반드시 나타난다는 가설이다. (이 가설은 1937년에 공개되었고 아직까지 아무도 이것을 증명하거나 부정증명하지 못하였다.)

이 가설을 Coq에서 나타내는 명제를 얻으려면 get_collatz_sequence와 exists 및 In을 사용하면 될 것 같은데..., SF는 다음과 같은 함수를 제시하며 이렇게 하면 안 된다고 한다.

```
1 Fail Fixpoint reaches_1_in (n : nat) :=
2   if n =? 1 then true
3   else 1 + reaches_1_in (f n).
```

안 되는 이유는, 재귀호출에서 reaches_1_in의 인수가 'obviously smaller than n'이 아니기 때문이라고 한다. 그리고 해결책은 귀납적으로 프랍을 정의하는 것이라고 하였다. 물론 이것이 이 장에서 우리가 공부하려는 중요한 이슈이기는 하지만 앞서 말했듯이 다른 해결책도 있다.

그리고 이 함수의 리턴값의 타입이 bool인 것 같은데 그렇다면 1 + reaches_1_in (f n)에서 타입 오류가 발생하는 것도 문제다. 그리고 false를 리턴하는 조건은 존재하지도 않는다. 이 부분은 저자가 너무 성의없이 쓴 것 같다.

콜라츠 가설은 (함수가 아니라) 그 가설을 나타내는 프랍을 다음과 같이 귀납적으로 정의하여 나타낼 수 있다.

```
1 Inductive Collatz_holds_for : nat -> Prop :=
2   | Chf_done : Collatz_holds_for 1
3   | Chf_more (n : nat) : Collatz_holds_for (f n) -> Collatz_holds_for n.
4
5 Conjecture collatz : forall n, Collatz_holds_for n.
```

라인 5는 다음과 동일한 효과를 가진다. 다만 독자에게 전달되는 저자의 의도에서 차이가 있다.

```
Theorem collatz : forall n, Collatz_holds_for n.
Proof. Admitted.
```

Collatz_holds_for라는 술어를 사용하는 프랍에 대한 증명의 예를 보자.

```
1 Example Collatz_holds_for_5 : Collatz_holds_for 5.
2 Proof.
3   apply (Chf_more 5). unfold f. simpl. (* Collatz_holds_for 16 *)
4   apply (Chf_more 16). unfold f. simpl. (* Collatz_holds_for 8 *)
5   apply Chf_more. unfold f. simpl. (* Collatz_holds_for 4 *)
```

```
6    apply Chf_more. unfold f. simpl. (* Collatz_holds_for 2 *)
7    apply Chf_more. unfold f. simpl. (* Collatz_holds_for 1 *)
8    exact Chf_done. Qed.
```

비교적 간단한 증명이지만 지금까지 경험하지 못했던 증명 기법을 사용했으므로 이제부터 이것에 대하여 상세하게 설명하겠다. 우리가 공부할 새로운 기법은 다음과 같다.

> 귀납적으로 정의된 프랍의 생성자를 인수로 하여 apply 책략을 적용한다.

원래 apply 책략의 인수는 컨텍스트에 있는 가설, 혹은 정리가 되어야 한다.[1] 그런데 여기서는 생성자가 인수로 사용되었다.[2] 이럴 때는 그 생성자의 타입을 인수로 사용한다고 보면 된다. 예를 들어

```
Check Chf_more. (* forall n : nat,
                   Collatz_holds_for (f n) -> Collatz_holds_for n *)
```

이므로 apply Chf_more는 $\forall n \in \mathbb{N}$, Collatz_hods_for (f n) → Collatz_holds_for n을 apply하는 것으로 해석된다.

생성자 Chf_more의 타입은 전칭조건문이고, 전칭한정사의 묶인변수를 n으로 특정했하여 얻은 Chf_more n의 타입은 조건문 Collatz_holds_for (f n) -> Collatz_holds_for n이다. 그러므로 apply Chf_more n은 고울 프랍이 Collatz_holds_for n 일 때 적용할 수 있으며, 이렇게 했을 때 고울 프랍은 조건문의 전건인 Collatz_holds_for (f n)으로 바뀌게 된다.

이제 증명 스크립트의 라인 3을 들여다 보자. apply (Chf_more 5)를 실행하기 직전의 고울 프랍은 Collatz_holds_for 5이다. 여기에

```
Chf_more 5 : Collatz_holds_for (f 5) -> Collatz_holds_for 5
```

를 apply하면 이 조건문의 후건이 현재의 고울 프랍과 일치하므로, 고울 프랍은 새롭게, 조건문의 전건인 Collatz_holds_for (f 5)로 바뀐다. 이 표현항은 현 상태로는 simpl이 되지 않으므로 먼저 unfold f를 해 주어야 한다. 그 다음 simpl 하면 Collatz_holds_for 16이 얻어진다.

그 다음은 이런 과정의 반복이다. 그런데 apply (Chf_more n)에서 인수 n은, 이것의 값이 무엇이 되어야 하는지를 Coq이 스스로 알아서 찾아낼 수 있으므로, 라인 5, 6, 7에서 보듯이, 생략해도 된다.

마지막에 고울명제가 Collatz_holds_for 1이 되었을 때 이것이 생성자 Chf_done의 타입과 일치하므로 apply Chf_done, 혹은 exact Chf_done을 적용하여 증명을 마친다.

Example: Ordering

앞에서 본 Collatz_holds_for는 1항 술어이다. 2항 술어도 귀납을 써서 정의할 수 있다. 물론 애리티가 3 이상인 술어도 귀납을 써서 정의할 수 있지만 이러한 예는 애리티가 1, 2인 경우에 비해서 현저히 적다.

---

[1] 예를 들어 컨텍스트에 H: P -> Q가 있을 때, 그리고 고울 프랍이 Q일 때 apply H를 적용할 수 있다. 이 경우 프랍 H: P -> Q가 인수로 사용되었다고 말한다. 엄격히 말하자면 우리가 인수로 사용한 것은 프랍이 아니라 가설 H이고 이것은 P -> Q의 inhabitant이다. 프랍을 타입으로 보았을 때 그 타입의 inhabitant를 다른 말로 에비던스(*evidence*)라고 하며 이것이 앞으로 우리가 공부할 핵심적인 개념이다.

[2] 프랍(혹은 술어)의 Inductive 정의에서의 생성자는 가장 원시적인primitive 에비던스이다.

이 장의 제목은 Inductively Defined Propositions로 되어 있는데, 실은 이 장에서 배우는 것은 프랍의 구성요소인 술어를 Inductive를 써서 정의하는 것이다. 순수한 의미에서 Inductive*ly* 정의된 프랍은 True와 False밖에 없다. Inductive 정의는 인수를 받아들여 재귀를 사용할 때 비로소 무한히 많은 대상을 생성할 수 있게 되어 흥미를 끌게 된다. 인수를 받는 프랍은 곧 술어이므로 이 장의 제목은 Inductively Defined Predicates가 더 적절할 수 있을 것이다.

자연수 간의 대소 관계를 나타내는 부울값 함수 leb는 다음과 같이 정의되어 있다.

```
Fixpoint leb (n m : nat) : bool :=
  match n with
  | O => true
  | S n' =>
      match m with
      | O => false
      | S m' => leb n' m'
      end
  end.
```

한편 자연수들 간의 등호 관계는 부울값 함수 eqb를 다음과 같이 정의해서 나타낼 수 있지만

```
Fixpoint eqb (n m : nat) : bool :=
  match n with
  | O => match m with
         | O => true
         | S m' => false
         end
  | S n' => match m with
            | O => false
            | S m' => eqb n' m'
            end
  end.
```

더 간단하게, 직접 prop을 써서

```
n = m
```

으로 나타낼 수도 있다. 그렇다면 leb 대신으로 사용할 수 있는 술어는 어떻게 정의해야 하겠는가?

다음과 같이 귀납적으로 정의할 수 있다.

```
1 Inductive le : nat -> nat -> Prop :=
2   | le_n (n : nat)   : le n n
3   | le_S (n m : nat) : le n m -> le n (S m).
4
5 Notation "n <= m" := (le n m) (at level 70).
6
7 Example le_3_5 : 3 <= 5.
8 Proof.
9   apply le_S. apply le_S. apply le_n. Qed.
```

단, 라인 9 맨 뒤에 있는 apply le_n은 exact le_n으로 대체할 수 없다. 만일 exact를 쓰고 싶다면 exact (le_n 3)으로 써야 한다.

여기서 하나 Notation을 그 기호가 나타내고자 하는 술어(혹은 함수) 이전에 선언할 수도 있다. 단, 이때는 Reserved라는 키워드를 앞에 붙여 주어야 한다.

```
Reserved Notation "n <= m" (at level 70).

Inductive le : nat -> nat -> Prop :=
  | le_n (n : nat)   : n <= n
  | le_S (n m : nat) : n <= m -> n <= (S m)

  where "n <= m" := (le n m).
```

이 생성자들의 타입을 알아보자.

```
Check le_n. (* forall n : nat, le n n *)
Check le_S. (* forall n m : nat, le n m -> le n (S m) *)
```

결과로 나타난 타입을 보면 다음과 같은 정의도 가능해 보인다.

```
Inductive le : nat -> nat -> Prop :=
  | le_n : forall (n : nat), le n n
  | le_S : forall (n m : nat), le n m -> le n (S m).
```

실제로 이 두 정의는 정확히 같은 의미를 가진다. 어느 구문을 사용할지는 순전히 취향의 문제이다.

하나 주의할 것은 =는 모든 타입의 원소에 적용할 수 있지만 <=는 자연수들에만 적용할 수 있다는 점이다. 전통적 수학에서는 $\leqslant$가 갖추어야 할 성질들을 공리로 규정하고 이를 만족하는 임의의 집합 위의 임의의 관계를 순서관계라고 부르지만, Coq은 모든 관계를 구성하여 사용한다는 점에서 전통적 수학과 사뭇 다르다고 볼 수 있다.

한편 Coq에서도 관계의 성질, 혹은 술어의 술어, 예를 들어 반사성, 추이성, 반대칭성 등을 정의하여 전통적 수학과 유사한 접근 방식을 취하는 것이 가능하다. 이러한 내용은 나중에 공부하게 될 것이다.

Example: Transitive Closure

다음은 *transitive closure*(추이폐포)에 대한 것이다.

```
Inductive clos_trans {X: Type} (R: X -> X -> Prop) : X -> X -> Prop :=
  | t_step (x y: X) :
      R x y -> clos_trans R x y
  | t_trans (x y z: X) :
      clos_trans R x y -> clos_trans R y z -> clos_trans R x z.
```

그런데 이것이 정확히 무엇을 정의하고 있는 것인지를 생각해 보자. clos_trans는 2항관계를 받아서 그것의 추이폐포를 리턴하는가? 'yes'라고 답하는 것이 맞기는 하지만 그 의미에 대해서 조금 생각이 필요하다.

여기서 우리는 2항관계를 부울값 함수나 집합이 아닌 프랍의 형태로 다루고 있다. 2항관계는 그것의 인수로 타입 X의 원소 2개를 받는다. 그리고 어떤 프랍을 리턴한다. clos_trans는 이러한 2항관계를 인수로 받아서 이러한 2항관계를 리턴한다.

clos_trans의 입력 인수로 사용할 아주 작은 2항관계를 만들어 보자.

```
Inductive Person : Set := Sage | Cleo | Ridley | Moss.

Inductive parent_of : Person -> Person -> Prop :=
  po_SC : parent_of Sage Cleo
| po_SR : parent_of Sage Ridley
| po_CM : parent_of Cleo Moss.
```

이 2항관계 parent_of는 4개의 원소로 이루어진 집합 Person 위에 정의되었다. 이 2항관계를 트리로 나타내면 변이 딱 3개다. parent_of의 추이폐포 ancestor_of는 다음과 같이 정의할 수 있다. 이제 Sage가 Moss의 조상임을 증명해 보자.

```
1  Definition ancestor_of : Person -> Person -> Prop :=
2    clos_trans parent_of.
3
4  Example ancestor_of1 : ancestor_of Sage Moss.
5  Proof.
6    unfold ancestor_of. apply t_trans with Cleo.
7      (* with (x:= Sage) (y:= Cleo) (z:= Moss) *)
8    - apply t_step. apply po_SC.
9    - apply t_step. apply po_CM. Qed.
```

라인 6 맨 오른쪽의 with Cleo는, 원래는 라인 7과 같이 써야 할 것을, Coq의 추론 능력을 믿고 3개의 인수들 중에 두 번째 것만 인수로 공급해 준 것이다. 라인 8과 라인 9에서 apply po_SC와 apply po_CM은 각각 exact po_SC와 exact po_SC로 대채해도 된다.

프랍과 증명 스크립트와 증거항

프랍proposition의 의도된 의미는 명제이다. 흔히 명제는 진위를 구별할 수 있는 문장이라고 한다. Coq의 논리적 기반인 직관주의(혹은 구성주의) 논리에서는 명제의 참은 증명의 존재로써 판별한다. 증명이 존재하는 명제는 참이고 그렇지 않은 명제는 거짓이다.

증명이라는 개념을 명확하게 하기 위하여는 프랍과 증명을 구문론적 존재로 정의해야 한다. 프랍부터 보자.

> 프랍은 타입이 Prop인 표현이다. 즉 프랍은 Prop의 원소이다.

프랍의 예를 보자.

```
1  Check 0 = 0. (* 0 = 0 : Prop *)
2  Check 0 = 1. (* 0 = 1 : Prop *)
3  Check forall n: nat, n = n. (* forall n : nat, n = n : Prop *)
4
5  Check n = n. (* Error *)
6
7  Definition nat_refl: Prop := forall n: nat, n = n.
8  Check nat_refl. (* nat_refl : Prop *)
9  Definition nat_refl2 := forall n: nat, n = 0 -> S n = 1.
10 Check nat_refl2. (* nat_refl2 : Prop *)
```

라인 5에서 에러가 난 이유는 n = n은 증명의 대상이 될 수 없기 때문이다. n이 무엇인지 (타입과 값을 모두) 모르기 때문에 참, 거짓을 판별할 수 없다.

참고로 Prop은 Type의 원소이다. 이 점에서는 Set과 같다.

```
11 Check Prop. (* Prop : Type *)
12 Check Set. (* Set : Type *)
13 Check nat. (* nat : Set *)
14 Check nat -> nat. (* nat -> nat : Set *)
15 Check Type. (* Type : Type *)
16 Check forall P: Prop, P -> ~~P. (* forall P : Prop, P -> ~~P : Prop *)
```

아직 프랍에 대해서 할 말이 많지만 이쯤에서 증명에 대해서 얘기해 보자. 먼저 증명의 예를 하나 보자.

```
17 Theorem nat_refl_thm : nat_refl. (* nat_refl is defined in line 7 *)
18 Proof.
19   unfold nat_refl. intros n.
20   Check n = n. (* n = n : Prop *)
21   reflexivity.
22   Show Proof. (* fun n : nat => eq_refl : nat_refl *)
23 Qed.
```

라인 20은 라인 5와 같지만 결과는 다르다. 그 이유는 이번에는 n이 컨텍스트에 들어와 있기 때문이다. 즉, 자유변수이다. 값은 모르지만 특정되어 있으므로 프랍에 나타날 수 있다.

라인 22의 `Show Proof.`는 현재 구성되어 있는 증거항(*proof term*)을 메시지 창에 보여준다. 여기서 `eq_refl`은 `nat_refl`이라는 타입을 가지는 증거항이다. 증거항은 증명항이라고도 하는데 이 책에서는 proof term을 번역한 증명항보다는 evidence term을 번역한 증거항이라는 용어를 쓰기로 하겠다.

증명에는 증명 스크립트proof script와 증거항이 있다. 둘 다 구문론적 개념이다. 인간에게는 전자가 더 다루기 쉽다. (항상 그런 것은 아니지만 대부분의 경우 그렇다.)

프랍과 정리를 정확히 구별지어 말하기는 쉽지 않다. 챗지피티에 의하면 프랍은 아직 증명을 찾지 않은 상태이고 증명을 찾게 되면 그것은 정리가 된다고 했지만 이건 그냥 나이브하게, 느낌적인 느낌으로 하는 말이다.

```
Check nat_refl. (* Prop *)
Check nat_refl_thm. (* nat_refl *)
Print nat_refl. (* forall n : nat, n = n : Prop *)
Print nat_refl_thm. (* nat_refl_thm = fun n : nat => eq_refl : nat_refl *)
```

프랍의 타입은 `Prop`이다. 정리의 타입은 그것이 증명한 프랍이다. 정리의 증명은 통상 증명 스크립트로 나타내지만 증거항을 써도 된다. `Print` 커맨드는 원래 주어진 표현의 정의를 보여주는 것인데, 정리의 경우에는 증거항을 보여주도록 Coq에 구현되어 있다.[3]

증거항을 증명 스크립트에서 사용하는 것은 아주 쉽다. 아래의 라인 6와 라인 9에 사용 예가 있다.

```
1 Definition myprf1 : nat_refl := (fun n : nat => eq_refl).
2 Definition myprf2 := (fun n : nat => eq_refl) : nat_refl.
3 (* myprf1 and myprf2 are exactly the same. *)
4
```

[3]Coq에서는 객체object라는 용어를 잘 쓰지 않는다. 대신 표현expression을 사용한다. '표현'이 너무 일반적인 느낌이라면 '표현항term'도 좋다. 표현항, 프랍, 함수, 타입 등으로 용도에 딱 맞는 구체적인 용어를 상황에 맞추어 사용하는 것이 Coq의 어법이다.

```
5 Theorem nat_refl_thm1 : nat_refl.
6 Proof. apply myprf1. Qed. (* exact instead of apply is ok *)
7
8 Theorem nat_refl_thm2 : nat_refl.
9 Proof. apply myprf2. Qed. (* exact instead of apply is ok *)
```

eq_refl이 도데체 무엇인지 그동안 궁금했을 것이다. 여기 공개한다.

```
Check @eq_refl. (* @eq_refl : forall (A : Type) (x : A), x = x *)
Print eq_refl. (* Inductive eq (A : Type) (x : A) : A -> Prop :=
                  eq_refl : x = x. *)
```

eq_refl은 귀납을 써서 정의된 프랍(1항 술어)인 eq의 (유일한) 생성자이다. 그리고 =은 eq의 Notation이다. Coq에서는 등호조차도 이렇게 정의하여 사용한다!

증거항에 왜 fun이 들어가야 하는지는 나중에 BHK 해석을 설명할 때 알게 될 것이다. 여기서는 그냥 받아들이기로 하자. fun의 사용 예를 아래의 라인 4에 보였다.

```
1 Definition nnext (n: nat) := S n.
2 Check nnext. (* nat -> nat *)
3
4 Definition nnext2 := fun (n: nat) => S n.
5 Check nnext2. (* nat -> nat *)
```

프랍은 전통적 1계논리에서 말하는 closed formula와 비슷하다. 하지만 컨텍스트에 따라 자유변수를 포함할 수도 있다. Atomic 프랍에는 술어기호가 필요하다. 현재까지 사용해 본 모든 술어기호는 귀납을 써서 정의되었으며 앞으로도 그럴 것이다. 0항 술어 True와 False는 앞서 다음과 같이 정의하였다.

```
Print True. (* Inductive True : Prop :=  I : True *)
Check I. (* I : True *)
Print False. (* Inductive False : Prop :=  *)
Check True. (* True : Prop *)
Check False. (* False : Prop *)
```

True는 항상 참이라고 하는데, 여기서 '항상'은 인수를 받아들이지 않는다는 뜻이다. True의 정의에서는 1개의 생성자 I를 사용한다. True를 증명하려면 apply I 하면 된다.

False는 항상 거짓이다. 즉 인수를 받아들이지 않고, 생성자가 없어 *uninhabited* 이다.

Coq에서 흔히 사용되는 잘못된 표현으로 '프랍의 값은 참이거나 거짓이다.'가 있다. 프랍에는 값이 부여되지 않는다.

'프랍은 참이거나 거짓이다.'는 맞는 말이다. 왜냐하면 주어진 프랍은 증거항으로 inhabit 되거나 그렇지 않기 때문이다. 다만 거짓인 프랍의 부정도 거짓일 수 있다. 예를 들면 forall (A : Prop), A \/ ~ A가 그러하다.

Example: Permutations

*Permutation*은 두 리스트 간의 관계로 볼 수 있다. 같은 타입의 원소로 이루어진, 길이가 n으로 같은 두 리스트 l_1, l_2는 어떤 전단사 함수 $\sigma : n \to n$이 있어서 $(\forall i < n)\big(l_2(i) = l_1(\sigma(i))\big)$가 성립할 때 서로 permutation 관계에 있다고 한다.

논의를 쉽게 하기 위하여 길이가 3인 리스트들에 대한 permutation 관계를 다음과 같이 정의하자.

```
Inductive Perm3 {X : Type} : list X -> list X -> Prop :=
  | perm3_swap12 (a b c : X) :
      Perm3 [a;b;c] [b;a;c]
  | perm3_swap23 (a b c : X) :
      Perm3 [a;b;c] [a;c;b]
  | perm3_trans (l1 l2 l3 : list X) :
      Perm3 l1 l2 -> Perm3 l2 l3 -> Perm3 l1 l3.
```

간단한 연습문제로 다음을 증명해 보자.

```
Fact Perm3_fact1 : Perm3 [1;2;3] [3;2;1].
Proof.
  apply perm3_trans with [1;3;2].
  - apply perm3_swap23.
  - apply perm3_trans with [3;1;2].
    (* ... *)

Fact Perm3_fact2 : Perm3 [1;2;3] [1;2;3].
Proof.
  apply perm3_trans with [2;1;3].
  (* ... *)
```

Example: Evenness (yet again)

자연수의 짝수성에 대해서는 이미 다음과 같은 2개의 정의가 있다.

```
Fixpoint even (n: nat) : bool :=
  match n with
  | 0 => true | S 0 => false | S (S n') => even n'
  end.

Definition Even (n: nat) : Prop :=
        exists (m: nat), n = double m.

Fixpoint double (n: nat) :=
  match n with | 0 => 0 | S n' => S (S (double n'))
  end.
```

그러나 아래에 3번째 정의를 제시한다.

```
Inductive ev : nat -> Prop :=
  | ev_0 : ev 0
  | ev_SS (n : nat) (H : ev n) : ev (S (S n)).

Check ev. (* : nat -> Prop *)
Check ev_0. (* : ev 0 *)
Check ev_SS. (* : forall n : nat, ev n -> ev (S (S n)) *)
```

이 정의에는, 이전에도 `Collatz_holds_for`, `le` 등의 정의에서 보았지만 그때는 언급하지 않았던, 특이한 점이 있다. 아래 보인 `list`의 정의와 비교해 보자.[4]

[4]주의: 이것은 우리가 통상 사용하는 implicit argument를 사용하지 않은 정의이다.

```
1 Inductive list (X: Type) : Type :=
2   | nil
3   | cons (x: X) (l: list X).
```

list는 인수로 타입 변수를 받아들이는 다형 타입polymorphic type이다. ev는 인수로 타입의 원소를 받아들이는 의존 타입dependent type이다.

또한 Inductive list ...에서 list의 인수 X는 콜론의 왼쪽에 위치한다. 따라서 생성자 nil과 cons가 나타내는(생성하는) 것의 타입은 당연히 list X로 (X에 따라) 결정된다. 그렇기 때문에 라인 2와 라인 3에 있는 nil과 con .. 뒤에 : list X를 붙이지 않은 것이다. 다음과 같이 붙여도 문제는 없다.

```
Inductive list (X: Type) : Type :=
  | nil : list X
  | cons (x: X) (l: list X) : list X.
```

list의 정의에서 인수 X: Type은 콜론의 왼쪽에 위치한다. 이를 *parameter*라고 한다. 이에 반하여 ev의 인수는 콜론의 오른쪽에 위치한다. (정확히 말하면 ev의 인수는 콜론 오른쪽에 이름 없이 타입만 nat이라고 지정하여 나타난다.) 이를 *index*, 또는 *annotation*이라고 한다.

Inductive mytype의 인수가 (x: X): ...와 같은 파라메터 형식을 취할 때는 이것의 생성자가 만들어 내는 원소의 타입은 mytype x가 되어야 한다. 그러나 Inductive mytype의 인수가 ... : X -> Prop과 같은 인덱스 형식을 취할 때는 이것의 생성자가 만들어 내는 원소의 타입은 mytype (S (S x)) 등 다양한 값을 가질 수 있다.

예를 들어 다음과 같은 정의는 에러를 낸다.

```
Fail Inductive wrong_ev (n : nat) : Prop :=
  | wrong_ev_0 : wrong_ev 0
  | wrong_ev_SS (H: wrong_ev n) : wrong_ev (S (S n)).
(* Unable to unify "wrong_ev 0" with "wrong_ev n". *)
```

파라메터 형식을 유지하며 위 정의에서 에러를 제거하려면 각 생성자의 콜론 오른쪽에 wrong_ev n을 두는 수밖에 없다. 물론 이렇게 정의된 wrong_ev는 우리의 의도에서 한참 어긋난다.

짝수성을 정의하는 4번째 방법이 있다.

```
Fixpoint ev2 (n: nat) : Prop :=
  match n with
  | O => True
  | S O => False
  | S (S n') => ev2 n'
  end.

Compute ev2 27. (* = False: Prop *)
Compute ev2 400. (* = True: Prop *)
Fact even_400 : ev2 400.
Proof. simpl. exact I. Qed.
```

이것은 술어이지만 정의에 Inductive를 사용하지 않았으며 따라서 생성자도 사용하지 않았다.

술어를 정의할 때, `Definition`, `Fixpoint`, `Inductive` 중 어느 것을 사용해도 된다. 첫 두 경우에는 인수가 콜론의 왼쪽에 있어야 한다. 3 번째 경우에는 인수를 콜론의 오른쪽에 둔다. 그리고 생성자의 구문에도 상당히 특이한 부분이 있다.

```
ev_SS (n : nat) (H : ev n) : ev (S (S n)).
```

에서와 같이 재귀적으로 자기 자신을 불러서 사용할 때 콜론의 생성자 `ev_SS`의 두 번째 인수 (`H: ev n`)이 정말 특이하다. (`ev n: Prop`)이라고 써야 할 것 같기도 한데 그렇지 않다. 그 이유를 알아보자.

(참고로 `H`는 콜론의 오른쪽에 나타나지 않는다. 그러므로 (`H : ev n`)은 (`_ : ev n`)이라고 써도 된다. 즉, 이 두 구문은 동일한 의미를 가진다.)

프랍은 타입이고 이것의 원소(inhabitant)는 그 프랍의 증명이므로 `ev_SS n H : ev (S (S n))`는 `ev_SS n H`가 프랍 `ev (S (S n))`의 증명이라는 뜻이다.

직관적으로 볼 때 `ev (S (S n))`의 증명은 `n`과 `ev n`으로부터 얻는 것이 아니라 `n`과 <u>`ev n`의 증명</u>으로부터 얻는 것이 당연하다. 따라서 우리는 `ev_SS`의 두 번째 인수로 프랍 (`ev n: Prop`)이 아니라 증거항 (`H: ev n`)을 사용하는 것이다.[5]

SF는 `Coq`를 사용하는 법을 공부하기 위한 책이며 `Coq`의 구조와 원리에 대한 엄격한 설명은 충분하게 제공하지 않는다. 이런 부분은 [2, 3]을 참고하여 공부하면 된다.

———— ○ ————

SF에 다음과 같은 내용이 있다.

> We can think of <u>this</u> as defining a `Coq` <u>property</u> `ev : nat -> Prop`, together with *evidence constructors* `ev_0 : ev 0` and `ev_SS : forall n, ev n -> ev (S (S n))`.
>
> These evidence constructors can be thought of as *primitive evidence of evenness*, and they can be used just like proven theorems. In particular, we can use `Coq`'s `apply` tactic with the constructor names to obtain evidence for `ev` of particular numbers, or we can use function application syntax to combine several constructors.

대단히 중요한 내용이다. 예를 들어가면서 찬찬히 음미해 보자.

① 밑줄친 <u>this</u>는 문맥상 `ev`의 `Inductive` 정의를 의미하는 것으로 보인다.

② <u>property</u>는 내가 말하는 술어predicate, 혹은 1항 술어라고 부르는 개념이다.

③ 증거 생성자(*evidence constructor*)의 첫 번째 것 `ev_0 : ev 0`는 특별할 것이 없으니 됐고, 두 번째 것 `ev_SS : forall n, ev n -> ev (S (S n))`은 왜 정의대로

```
ev_SS (n : nat) (H : ev n) : ev (S (S n)).
```

와 같이 쓰지 않았을까? 그 이유는 `ev_SS`의 (정의가 아니라) 타입을 드러내고 싶었기 때문이다. 그렇다면 왜 타입이 이렇게 될까?

---

[5]정확히 말하면 (`H : ev n`)는 증거항이 아니라 증거항 `H`가 `ev n`의 에비던스라는 것을 나타내는 타입 선언type declaration이다. 그리고 `H`는 증거항을 나타내는 구문론적 변수이다. 예를 들어 `ev_0 : ev 0`에서 `ev_0`는 프랍 `ev 0`의 증거항이다. `H : ev 0`라고 쓴다면 `H`는 `ev_0`를 나타내는 구문론적 변수이다.

④ 두 번째 문단은 예를 통해서만 확실히 이해할 수 있다.

①과 ②에 대해서는 간단하게 설명되었으니 이제 ③의 증거 생성자 ev_SS의 타입에 대해서 말하겠다.

Inductive로 Set타입을 정의할 때는 생성자가 만들어 내는 원소의 타입은 파라메터를 포함한 하나의 표현식으로 결정된다. 그리고 생성자에 인수를 공급하면 그 표현 자체가 그 타입의 원소이다. 예를 들어 nil nat, cons nat 2 (nil nat) 등은 list nat: Type의 원소이고, 0, S 0 등은 nat: Type의 원소이다.

그러나 Inductive로 프랍, 예를 들어 ev n: Prop을 정의할 때는 파라메터를 사용하지 않으며, 생성자가 만들어 내는 원소(프랍의 증명)의 타입(프랍)은 인덱스(or annotation)를 포함한 여러 표현식, 예를 들어 ev 0, ev (S (S 0)) 등으로 나타나게 된다. 생성자에 인수를 공급하면 그 표현 자체가 원소인 것은 아까와 마찬가지다. 예를 들어 ev_0, ev_SS 0 ev_0, ev_SS 2 (ev_SS 0 ev_0) 등은 각각 ev 0, ev 2, ev 4의 원소이다. ev 1, ev 3 등의 원소는 존재하지 않는다.

ev_0는 인수를 받지 않는 상수형 증거 생성자이다. 그리고 이것의 타입은 eq 0이다. 즉 ev_0는 eq 0의 한줄짜리one-liner 증명이다. 공리라고 생각해도 좋다. 이를 증거항, 또는 에비던스evidence라고 부른다. 아래에서 라인 2를 보라.

```
1 Check ev 0. (* : Prop *)
2 Check ev_0. (* : ev 0 *)
3 Check ev_SS. (* : forall n : nat, ev n -> ev (S (S n)) *)
4 Check ev_SS 0. (* : ev 0 -> ev 2 *)
5 Check ev_SS 1. (* : ev 1 -> ev 3 *)
6 Check ev_SS 2. (* : ev 2 -> ev 4 *)
7 Check ev_SS 0 ev_0. (* : ev 2 *)
8 Check ev_SS 2 (ev_SS 0 ev_0). (* : ev 4 *)
9 Check ev_SS 4 (ev_SS 2 (ev_SS 0 ev_0)). (* : ev 6 *)
```

ev_0는 상수형 생성자이므로 증거 생성자인 동시에 증거evidence이다. ev_SS는 증거 생성자이지만 아직 증거는 아니다. 2개의 인수를 받아야 비로소 증거가 된다. ev_SS는 그 자체로는 아무것도 증명하지 않는다.

라인 7을 보자. ev_SS의 정의에 따르면 ev_SS n (H: ev n)은 ev (S (S n))의 증거라고 하는데, 라인 7은 여기서 n := 0로 특정된 형태 ev_SS 0 ev_0: ev 0이다. 따라서 이것은 ev (S (S 0)), 즉 ev 2의 증거이다.

라인 4의 ev_SS 0는 어떤가? 이것은 뒤에 인수 (H : ev 0)를 붙여 주었을 때 ev (S (S 0))의 증거, 즉 원소가 된다. 이때 H는 ev 0의 임의의 원소이므로 결국 ev_SS 0는 ev 0에서 ev (S (S 0))로 가는 함수이다. 즉 ev 0 -> ev 2 타입을 가지며 이를 ev_SS 0 : ev 0 -> ev 2로 쓸 수 있다.

라인 4에서 0를 n으로 바꿔 주면 ev_SS n : ev n -> ev (S (S n))이 성립함을 알 수 있다.

이제 ev_SS를 보면, 여기에 임의의 인수 n을 공급했을 때 ev n -> ev (S (S n)) 타입을 가지므로, ev_SS의 타입은 라인 3에 보였듯이 forall n, ev n -> ev (S (S n))라고 말하는 것이 타당함을 알 수 있다.

실제로 ev를 다음과 같이 정의해도 원래의 정의와 완전히 똑같은 효과를 가진다.

```
Inductive ev : nat -> Prop :=
  | ev_0 : ev 0
  | ev_SS : forall n, ev n -> ev (S (S n)).
```

이상으로써 ③의 설명을 마친다.

④가 말하는 두 번째 문단은 메우 중요한 사실에 대한 훌륭한 설명이기는 하나 경험이 적은 사람이 이해하기는 쉽지 않을 수 있다.

"use Coq's apply tactic with the constructor names to obtain evidence for ev of particular numbers"라고 한 것은, 생성자는 증명된 정리와 마찬가지로 apply의 인수로 사용할 수 있으므로 ev 2, ev 26 등 n에 특정 값을 주었을 때의 프랍에 대한 증명 스크립트를, apply *constructor name*을 반복 적용하여 얻을 수 있다는 뜻이다.

그리고 "function application syntax to combine several constructors"라고 하는 것은 생성자를 함수로 보아 함수의 합성function composition을 사용한다는 뜻이다. 이것은 증거항을 얻을 때 사용된다.

```
1  Theorem ev_4 : ev 4.
2  Proof.
3    apply ev_SS. (* ev 2 *)
4    apply ev_SS. (* ev 0 *)
5    apply ev_0.
6  Qed.
7
8  Theorem ev_4' : ev 4.
9  Proof. apply (ev_SS 2 (ev_SS 0 ev_0)). Qed.
```

라인 3에서 인수 n은 자동으로 2로 특정되었다. 고울 프랍이 ev 4이므로 Coq이 ev_SS의 조건문 ev n -> ev (S (S n))의 후건에 고울 프랍 ev 4를 매치한 것이다. 그러므로 라인 3은 apply ev_SS : ev 2 -> ev 4와 같다. 따라서 고울 프랍은 ev 2로 변경된다.

같은 원리로 라인 4에서 인수 n은 0으로 특정되었다. 증명 스크립트는 이렇게 apply *constructor*를 반복 적용하여 얻을 수 있다.

이제 증거항을 얻어 보자. 라인 8에서 생성자 ev_SS를 2항 함수로 보고 ev_0를 상수로 보아 합성하여 단번에 증거항을 얻었다. 여기서는 apply 대신 exact를 써도 된다.

증거항은 증명 스크립트로부터 쉽게 얻을 수 있다. 라인 5 아래에 Show Proof를 넣고 실행하면 증거항 (ev_SS 2 (ev_SS 0 ev_0))이 메시지 화면에 나타난다.

고울 프랍이 ev (S (S n)) 형태일 때는 항상 ev_SS를 쓰면 된다. 물론 이때 n이 홀수이면 마지막에 가서 증명을 마무리할 수 없을 것이다. 다음의 예를 보라.

```
Theorem ev_plus4 : forall n, ev n -> ev (4 + n).
Proof.
  intros n. simpl. intros Hn.
  apply ev_SS. apply ev_SS.
  exact Hn. (* apply Hn. *)
Qed.
```

다음은 쉬운 연습문제다. n에 대한 귀납을 사용한다.

```
Theorem ev_double : forall n,
  ev (double n).
Proof.
  intros n. induction n as [| n' IHn'].
  - (* n = 0 *) simpl. (* ... *)
```

```
  - (* n = S n' *) simpl. (* ... *)
Qed.
```

이 장에서 우리는 술어를 Inductive를 써서 정의하는 방법을 공부하고 있다. 이러한 정의에서 사용되는 생성자는, 적당한 타입의 인수들을 받아서 프랍의 원소, 즉 증거항을 만들어 낸다. Inductive definition of predicates의 정확한 의미를 알았으니, 이제 이러한 술어들로 이루어진 프랍을 증명하는 여러 기법들을 공부하도록 하자.

8.2 Using Evidence in Proofs

지금까지 관찰한 바에 의하면 각 n: nat에 대하여 ev n은 단 하나의 원소만을 가진다. 이 사실은 엄격하게 증명할 수도 있지만 일단 그냥 받아들여 사용하기로 하자. 이와 관련하여 SF는 다음과 같이 말하고 있다.

Defining ev with an Inductive declaration tells Coq not only that the constructors ev_0 and ev_SS are valid ways to build evidence that some number is ev, but also that these two constructors are the *only* ways to build evidence that numbers are ev.

In other words, if someone gives us evidence E for the assertion ev n, then we know that E must be one of two things:

- E is ev_0 (and n is 0), or
- E is ev_SS n' E' (and n is S (S n'), where E' is evidence for ev n').

Inversion on Evidence

증명 중 컨텍스트에 E : ev n을 가설로 가지고 있을 때 이를 이용하는 방법으로 destruct E가 있다. 이것은 다른 종류의 destruct와 마찬가지로 일종의 case analysis이다.

destruct 책략은 원래 컨텍스트에 있는 타입 선언, 예를 들어 n: nat를 destruct하여 n = 0 case와 n = S n' case로 나누어 각 경우에 대한 증명을 하는 것이다. 일반적인 destruct 책략은 T : Set가 귀납에 의하여 정의되었고 이 정의에서 생성자가 $c_1, \ldots, c_k$ 라면, 표현항 t의 값의 타입이 T일 때 destruct t를 실행하여 T의 생성자가 c_i, $(i = 1, \ldots, k)$ 인 경우로 나누어 각 경우에 대하여 증명하는 것이다.

이 아이디어를 에비던스 E: ev n에 적용할 수 있다. destruct E를 실행하면 E = ev_0 경우와 E = ev_SS n' E' 경우로 나누어 각 경우에 대하여 증명하는 것이다. E = ev_0 경우는 n = 0로, E = ev_SS n' E' 경우는 exists n', n = S (S n') /\ ev n'로 이어지는데 이 두 경우로 나누어 증명하는 기법을 *inversion*이라고 하며, 이를 사용할 때 다음의 보조정리가 유용하다.

```
1 Theorem ev_inversion : forall (n : nat),
2   ev n ->
3   (n = 0) \/ (exists n', n = S (S n') /\ ev n').
4 Proof.
5 intros n E.
6 destruct E as [ | n' E'] eqn:EE. (* destruct evidence *)
7 - (* E = ev_0 : ev 0 *)
8   left. reflexivity.
9 - (* E = ev_SS n' E' : ev (S (S n')) *)
```

```
10    right. exists n'.
11    split.
12    + reflexivity.
13    + exact E'.
14  Qed.
```

라인 5를 실행한 뒤 Coq Goals 화면은 다음과 같다.

```
n : nat
E : ev n
(1 / 1) ------------------------
n = 0 \/ (exists n' : nat, n = S (S n') /\ ev n')
```

라인 6에서 가설(혹은 에비던스) E: ev n을 destruct하면 다음과 같이 된다.

```
Goal 1
E : ev 0
EE : E = ev_0
(1 / 2) ------------------------
0 = 0 \/ (exists n' : nat, 0 = S (S n') /\ ev n')

Goal 2
n' : nat
E : ev (S (S n'))
E' : ev n'
EE : E = ev_SS n' E'
(2 / 2) ------------------------
S (S n') = 0 \/ (exists n'0 : nat, S (S n') = S (S n'0) /\ ev n'0)
```

그 다음은 Coq 화면을 보지 않고도 증명을 읽을 수 있을 것이다.

ev_inversion은 흔히 *inversion lemma*라고 부른다. 이 보조정리의 응용 예를 아래 보였다.

```
1  Theorem evSS_ev : forall n, ev (S (S n)) -> ev n.
2  Proof.
3    intros n H. apply ev_inversion in H. (* H : ev (S (S n)) changes to disjunction *)
4    destruct H as [H0 | H1]. (* destruct disjunction *)
5    - discriminate H0.
6    - destruct H1 as [n' [Hnm Hev]]. (* destruct existential conjunction *)
7      injection Hnm as Heq.
8      rewrite Heq. exact Hev.
9  Qed.
```

라인 3을 실행한 뒤 Coq Goals 화면은 다음과 같다.

```
n : nat
H : S (S n) = 0 \/ (exists n' : nat, S (S n) = S (S n') /\ ev n')
(1 / 1) ------------------------
ev n
```

그 다음은 새로울 것 없다.

Inversion 기법을 사용하기 위하여 매번(Inductive로 정의된 술어마다, 예를 들어 ev에 대하여) inversion lemma를 증명하는 것은 번거롭다. 이 문제의 해결을 위하여 Coq은 inversion이라는 책략을 제공한다. 위에 보인 evSS_ev의 증명은 inversion 책략을 쓰면 아주 간단해 진다.

```
Theorem evSS_ev' : forall n, ev (S (S n)) -> ev n.
Proof.
  intros n E.  inversion E as [| n' E' Heq].
  (* We are in the [E = ev_SS n' E'] case now. *)
  exact E'.
Qed.
```

inversion이 너무 많은 일을 한꺼번에 해 주기 때문에 증명과정의 세세한 부분을 이해하기 힘들 수도 있겠는데, evSS_ev의 증명의 각 단계를 해부해서 들여다 보면 거의 모두가 루틴routine 이며, 따라서 모든 단계들을 inversion 하나로 처리하는 것이 이상할 것 없다는 것을 알 수 있을 것이다.

증명에 inversion을 사용한다는 말은 ① 관련된 술어에 대한 inversion lemma를 증명하고 이를 이용하는 것을 의미할 수도 있고, ② inversion 책략을 사용하는 것을 의미할 수도 있는데 통상 ②의 의미로 많이 쓰인다.

inversion 책략은 방금 공부했듯이 귀납으로 정의된 술어로 이루어진 아톰 프랍 가설에 대해서 적용할 수 있는데, 등호도 귀납으로 정의된 술어이므로 H : S x = S y와 같은 가설에 대해서도 inversion을 쓸 수 있다.[6]

ev에 대한 inversion lemma를 쓰는 것과 inversion 책략을 쓰는 것의 차이를 보여주는 또 하나의 예를 보자.

```
Theorem one_not_even : ~ ev 1.
Proof.
  unfold not. intros H. apply ev_inversion in H.
  destruct H as [H1 | H2]. (* destruct disjunctive hyp *)
  - discriminate H1.
  - destruct H2 as [m Hm]. (* destruct existential hyp *)
    destruct Hm as [Hm1 Hm2]. (* destruct conjunctive hyp *)
    injection Hm1 as Hm1. discriminate Hm1.
Qed.

Theorem one_not_even' : ~ ev 1.
Proof.
  unfold not. intros H.
  inversion H. Qed.
```

---

inversion 책략은 여러 경우에 일어나는 많은 일을 자동으로 처리해 주므로 앞서 몇 개의 예에서 보았듯이 대단히 편리하다. inversion의 작업 과정을 충분히 세밀하게 이해하고 있다면, 이 책략을 쓰는 것은 증명의 효율을 높이는 데 큰 도움이 될 것이다. 하지만 제대로 이해하지 못한 상태에서 무작정 쓰는 것은 권장하지 않는다. inversion 책략이 하는 일은 SF에 다음과 같이 설명되어 있다.

The inversion tactic does quite a bit of work. For example, when applied to an equality assumption, it does the work of both discriminate and injection. In addition, it carries out the intros and rewrites that are typically necessary in the case of injection.

[6] 이런 가설에는 원래 injection과 rewrite를 사용하는 것으로 배웠지만 inversion을 쓰면 더 간편하다.

It can also be applied to analyze evidence for arbitrary inductively defined propositions, not just equality.

Here's how `inversion` works in general.

- Suppose the name `H` refers to an assumption `P` in the current context, where `P` has been defined by an `Inductive` declaration. (이것은 컨텍스트에 `H : P`, `H : P t1`, `H : P t1 t2`, … 같은 것이 들어 있다는 뜻이다.)
- Then, for each of the constructors of `P`, `inversion H` generates a subgoal in which `H` has been replaced by the specific conditions under which this constructor could have been used to prove `P`.
- Some of these subgoals will be self-contradictory ; `inversion` throws these away. (컨텍스트에 모순이 있는 서브고울은 버린다 — 고울 화면에 보여주지 않는다.)
- The ones that are left represent the cases that must be proved to establish the original goal. For those, `inversion` adds to the proof context all equations that must hold of the arguments given to `P`.

위의 설명은 틀린 부분은 없으나 이것만 가지고는 `inversion H`가 정확히 무엇을 하는지 알기가 쉽지 않다. `inversion` 책략을 이해하고 능숙하게 사용하기 위하여는 풍부한 예를 통한 연습이 필수적이다.

`inversion`은 결국 주어진 에비던스를 생성할 수 있는 모든 경로를 하나씩 조사하는 것이며, 원래는 `destruct`를 실행하고 그 다음에 수동으로 했어야 할 많은 작업들을 자동으로 해 주는 것이다. `inversion`을 사용하여 얻은 모든 증명은 원칙적으로 `inversion` 없이도 가능하다. 단, 이를 위해서는, 아주 간단한 경우를 제외하고는, inversion lemma for `P`를 `destruct`를 써서 증명하고, 그 다음에 이 lemma를 증명에 사용해야 할 것이다. 이런 방법으로 증명을 구성하는 것은 번거롭기는 하지만, `inversion`을 정확하게 이해하고 있는지를 확인하는 좋은 방법이다.

Induction on Evidence

자연수나 리스트 등 귀납적 타입의 원소에 대한 `destruct`보다 더 강력한 책략으로 `induction`이 있듯이, 에비던스에 대해서도 `destruct`보다 더 강력하며 `inversion`이 해결하지 못하는 경우에 사용할 수 있는 책략이 있다. 바로 `induction` (*on evidence*)이다.

`destruct E`와 `induction E`의 공통점은 생성자에 따른 case analysis를 한다는 것이고, 차이점은 `induction`이 `destruct`보다 더 많은 정보, 즉 귀납가설 `IH`를 제공한다는 것이다.

다음의 예를 가지고 설명하겠다.

```
1  Lemma ev_Even : forall n,
2    ev n -> Even n.
3  Proof.
4    intros n E.
5    induction E as [|n' E' IH].
6    - (* E = ev_0 with n = 0, goal is Even 0 *)
7      unfold Even. exists 0. simpl. reflexivity.
8    - (* E = ev_SS n' (Even n') with n = S (S n') and IH : Even n' *)
9      unfold Even in IH.
10     destruct IH as [k Hk].
```

```
11      rewrite Hk.
12      unfold Even. exists (S k). simpl. reflexivity.
13  Qed.
```

라인 5를 실행한 뒤 Coq Goals 화면은 다음과 같다.

```
(1 / 2) ------------------------
Even 0

Goal 2
n' : nat
E' : ev n'
IH : Even n'
(2 / 2) ------------------------
Even (S (S n'))
```

만일 라인 5가 destruct E as [|n' E']였다면 이를 실행했을 때 Coq Goals 화면은 다음과 같았을 것이다.

```
(1 / 2) ------------------------
Even 0

Goal 2
n' : nat
E' : ev n'
(2 / 2) ------------------------
Even (S (S n'))
```

차이점은 단 하나, 이번에는 IH가 없다는 것이다. 이제 증명 스크립트를 읽으면 진행 과정을 이해할 수 있을 것이다. ✓

역 방향 ev n <- Even n도 다음과 같이 쉽게 증명된다. 이번에는 destruct도 induction도 필요없다. 이 증명에서는 전에 증명해 두었던 보조정리 ev_double을 사용한다.

```
Check ev_double. (* forall n : nat, ev (double n) *)

Theorem ev_Even_iff : forall n,
  ev n <-> Even n.
Proof.
  intros n. split.
  - (* -> *) apply ev_Even.
  - (* <- *) unfold Even. intros [k Hk].
    rewrite Hk. apply ev_double.
Qed.
```

Exercises 1

다음은 ev에 대한 귀납적 증명 문제들이다.

① 두 짝수를 더하면 짝수를 얻게 됨을 보인다.

```
Theorem ev_sum : forall n m,
  ev n -> ev m -> ev (n + m).
```

```
Proof.
  intros n m Hn Hm.
  induction Hn as [| n' Hn' IH].
  - (* Hn = ev_0 *) (* ... *)
  - (* Hn = ev_SS n' Hn' *) (* ... *)
Qed.
```

② 다음은 짝수임을 말하는 술어 ev'를 ev와 다른 방법으로 정의하고 이 두 술어가 동등함을 보이는 문제이다.

이 문제에서 주목해야 할 것은 ev'의 정의에서 생성자가 3개라는 사실이다. 따라서 induction 책략을 사용하면 3개의 경우, 혹은 서브고울이 생긴다. 그리고 3번째 경우에는 변수가 n', m' 2개 필요하고 귀납가설도 IHn', IHm' 2개 필요하다.

```
1  Inductive ev' : nat -> Prop :=
2    | ev'_0 : ev' 0
3    | ev'_2 : ev' 2
4    | ev'_sum n m (Hn : ev' n) (Hm : ev' m) : ev' (n + m).
5
6  Theorem ev'_ev : forall n, ev' n <-> ev n.
7  Proof.
8    intros n. split.
9    - (* -> *) intros E.
10     (* there are 3 cases *)
11     induction E as [| | n' m' Hn' IHn' Hm' IHm'].
12     + (* E = ev'_0 *) apply ev_0.
13     + (* E = ev'_2 *) apply ev_SS. apply ev_0.
14     + (* E = ev'_sum n' m', Hn': ev n', Hm': ev m' *)
15       apply ev_sum.
16       (* ... *)
17   - (* <- *) intros E. induction E as [| n' Hn' IH].
18     + (* Hn = ev'_0 *) apply ev'_0.
19     + (* Hn = ev_SS n' Hn' *)
20       replace (S (S n')) with (2 + n').
21       (* ... *)
22 Qed.
```

③ 다음 문제는 컨텍스트에 2개의 에비던스가 있으므로 이들 중 어느 것에 induction을 적용할지가 문제이다. 하나를 선택하여 induction을 시도해 보고, 잘 되면 좋고, 잘 안 되면 다른 것에 대하여 induction을 시도해야 한다.

```
1  Theorem ev_ev__ev : forall n m,
2    ev (n+m) -> ev n -> ev m.
3  Proof.
4    intros n m Hnm Hn.
5    induction Hn as [| n' Hn' IH].
6    - (* Hn = ev_0 *) simpl in Hnm. exact Hnm.
7    - (* Hn = ev_SS n' Hn' *)
8      simpl in Hnm.
9      inversion Hnm as [| k Hnm' Eq].
10     (* ... *)
11 Qed.
```

이 증명에서 inversion Hnm을 썼는데, 이것 대신 apply ev_inversion in Hnm을 쓰는 방법도 있다.

④ 다음 문제는 별로 어렵지 않지만 끈기를 요구한다. 때로는 이런 문제도 풀어 보아야 한다.

```
Check plus_n_Sm. (* forall n m : nat, S (n + m) = n + S m *)

Lemma n_plus_n_ev : forall (n: nat),
  ev (n + n).
Proof.
  intros n. induction n as [| n' IH].
  - (* n = 0 *) simpl. apply ev_0.
  - (* n = S n' *) simpl. rewrite <- plus_n_Sm.
    apply ev_SS. exact IH.
Qed.

Theorem ev_plus_plus : forall n m p,
  ev (n+m) -> ev (n+p) -> ev (m+p).
Proof.
  intros n m p Hnm Hnp.
  assert (H: ev ((n+p) + (m+p)) -> ev (n+p) -> ev (m+p)).
  { apply ev_ev__ev with (n:=n+p) (m:=m+p). }
  apply H.
  - (* ev (n+p + m+p) *)
    replace (n + p + (m + p)) with ((n + m) + (p + p)).
    + apply ev_sum.
      { exact Hnm. }
      { apply n_plus_n_ev. }
    + rewrite add_assoc.
      replace (n + m + p) with (n + (m + p)).
      { replace (m + p) with (p + m).
        {
          (* .. tedious part .. *)
        }
        { rewrite add_comm. reflexivity. }
      }
      { rewrite add_assoc. reflexivity. }
  - (* ev (n+p) *) exact Hnp.
Qed.
```

8.3 Inductive Relations

이 섹션에서는 binary predicate, 즉 2항술어의 귀납적 정의에 대해서 공부하기로 한다.

$\leqslant$ relation on $\mathbb{N}$을 다음과 같이 정의한다.

```
Inductive le : nat -> nat -> Prop :=
  | le_n (n : nat)                : le n n
  | le_S (n m : nat) (H : le n m) : le n (S m).

Notation "n <= m" := (le n m).
```

Unit test를 위하여 다음을 실행해 보자.

```
1  Theorem test_le1 :
2    3 <= 3.
3  Proof.
4    apply le_n.  Qed.
5
6  Theorem test_le2 :
7    3 <= 6.
8  Proof.
9    apply le_S. apply le_S. apply le_S. apply le_n.
10 Qed.
11
12 Theorem test_le3 :
13   (2 <= 1) -> 2 + 2 = 5.
14 Proof.
15   intros H.
16   inversion H.
17   inversion H2.
18 Qed.
```

라인 9에서는 `apply le_S`들을 실행하면 고울 명제가 `3 <= 5`, `3 <= 4`, `3 <= 3`로 바뀌고, 마지막에 `apply le_n`을 실행하여 증명이 완료된다.

라인 15를 실행하면 `Coq Goals` 화면은 다음과 같이 된다. 고울 프랍은 증명할 수 없으므로 당연히 컨텍스트에서 모순을 얻어내야 한다. 이럴 때 `inversion` 책략이 특히 유용하다.

```
H : 2 <= 1
(1 / 2) ------------------------
2 + 2 = 5
```

그 다음에 `inversion H`를 실행하면 `Coq Goals` 화면은 다음과 같이 된다.

```
H : 2 <= 1
n, m : nat
H2 : 2 <= 0
H1 : n = 2
H0 : m = 0
(1 / 1) ------------------------
2 + 2 = 5
```

`H: 2 <= 1`은 `H: 2 <= S 0`이므로 `le_S (2 0) (H2: 2 <= 0)`에 의하여 생성되었을 것이기 때문에 `H2: 2 <= 0`가 컨텍스트에 추가된다. 다시 `H2: 2 <= 0`가 어떻게 생성되었는지를 찾아 `inversion` 해 보면 매치되는 증거 생성자가 존재하지 않으므로 증명이 완료된다. ✓

이번에는 $\mathbb{N}$ 위의 $<$ 관계를 다음과 같이 정의한다.

```
Definition lt (n m : nat) := le (S n) m.

Notation "n < m" := (lt n m).
```

Exercises 2

ⓐ Total relation. 모든 $n, m \in \mathbb{N}$에 대해서 `total_relation n m`이 성립함을 보이도록 해야 한다. 이 2항 술어의 (유일한) 생성자로 `tot`을 다음과 같이 정의하면 된다.

```
1  Inductive total_relation : nat -> nat -> Prop :=
2    tot (n m: nat) : total_relation n m.
3
4  Check tot. (* tot : forall (n m: nat), total_relation n m. *)
5
6  Theorem total_relation_is_total : forall n m,
7    total_relation n m.
8  Proof.
9    intros n m.
10   apply tot. Qed.
```

라인 2 대신 다음을 사용해도 된다.

```
tot : forall (n m: nat), total_relation n m.
```

ⓑ Empty relation.

모든 n m: nat에 대하여 empty_relation n m이 False를 함의imply해야 한다. 이를 위하여 empty_relation n m이 성립하기 위한 조건으로 불가능한 조건을 내 걸어야 한다.

```
1  Inductive empty_relation : nat -> nat -> Prop :=
2    empty (n m: nat) (H: n <> n) : empty_relation n m.
3
4  Theorem empty_relation_is_empty : forall n m,
5    ~ empty_relation n m.
6  Proof.
7    intros n m. unfold not. intros H.
8    inversion H. unfold not in H0.
9    apply H0. reflexivity. Qed.
```

Case Analysis, revisited

Coq에서의 case analysis에는 destruct, induction 및 inversion의 3가지가 있다. 이들은 모두 생성자에 따라 경우를 나누어 분석하는 것이므로 생성자별 분석이라고 부르기로 한다.

생성자별 분석 책략은 다음의 둘 중 하나의 형식을 취한다.

① case_anal term [as ...].

② case_anal evidence [as ...].

여기서 case_anal은 destruct, induction, inversion 중 어느 하나이며, term은 예를 들어 타입 선언 n : nat가 컨텍스트에 있을 때의 n이며, evidence는 예를 들어 가설 H : e1 <= e2가 컨텍스트에 있을 때의 H이다.

case_anal이 destruct일 때는 term이 변수가 아니라 표현항인 경우도 많이 쓰인다. evidence는 결합자나 한정사를 포함하지 않은 프랍, 즉 아톰 프랍이어야 한다.[7]

생성자별 분석 3개 중에 destruct와 induction은 term과 evidence 모두에 적용할 수 있지만 inversion은 evidence에만 적용할 수 있다.

[7]아톰 프랍은 Inductive를 써서 정의된 술어 하나만을 사용하여 만들어진 프랍이다.

생성자별 분석은 Inductive에 의해서 정의된 대상에 대하여 하는 것이므로, 예를 들어 함수 타입 대상에 대해서는 생성자별 분석이라는 개념이 존재하지 않는다. ①에서 term이 예를 들어 f x일 때 생성자별 분석은 f에 대해서 하는 것이 아니라 f x의 타입에 대해서 한다. ②에서 evidence가 예를 들어 P t일 때 생성자별 분성은 P에 대해서 한다.

이 하위절에서는 destruct, inversion, induction의 차이점을, 이들 책략을 E : e1 <= e2에 대하여 적용하는 예를 들어 설명하고자 한다.

H: e1 <= e2라는 에비던스가 컨텍스트에 있을 때 destruct H, inversion H, induction H에 의하여 각각 어떤 변화가 Coq Goals에 발생하는지를 상세히 설명하겠다. 여기서 e1, e2는 변수일 수도 있지만 일반적으로는 변수들을 포함하는 표현항이다. 세 책략 모두에 대하여 2개의 경우가 발생한다.

- 첫 번째는 H: e1 <= e2가 le_n에 의해서 생성된, e1 = e2인 경우이고,
- 두 번째는 H: e1 <= e2가 le_S에 의해서 생성된, e1 <= m인 어떤 m이 존재하여 e2 = S m인 경우이다.

destruct는 셋 중에 가장 기본이 되는 것이며 다른 2개는 destruct가 하는 일은 다 하고, 거기에 더해서 추가 작업을 한다.

- destruct H. 첫 번째 경우에는 e1 = e2이므로 컨텍스트와 고울에 나타나는 모든 e2를 e1으로 치환한다. 치환의 결과로 바뀐 가설들 중에 도움이 되지 않는 것들은, 예를 들어 n = n 같은 것은 삭제한다. 두 번째 경우에는 H: e1 <= e2에서 e2를 새로운 변수 m'으로 치환하여 얻은 H': e1 <= m'으로 (destruct의 대상이었던) H: e1: e1 <= e2를 대체한다. 그리고 고울 명제에서는 e2를 S m'으로 치환한다.
- inversion H. 각 경우, 즉 subgoal에 대하여, 컨텍스트에 있는 가설들이 생성자의 단사성 등에 의하여 모순을 함의하면 그 subgoal을 삭제한다. 각 subgoal에서, 생성자의 단사성 등에 의하여 성립해야 하는 등호들을 만들어서 컨텍스트에 넣는다.
- induction H. 고울에 나타난 모든 e2를 m'으로 치환하여 얻은 명제, 즉 귀납가설 IH'을 컨텍스트에 가설로 넣는다. 통상 induction H as [| m' H' IH']의 형식으로 사용된다.

  이것과 destruct의 차이는 귀납가설의 유무밖에 없으므로 일단 induction을 사용해 보고, ① 잘 되지 않으면 inversion을 사용하고, ② 증명은 잘 되었는데 귀납가설이 증명에 사용되지 않았다면 destruct로 바꾸는 것이 요령이다.

이상으로 귀납으로 정의된 술어 le와 이것으로 이루어진 에비던스 E: le t1 t2에 대한 destruct E, inversion E, induction E의 세 책략이 하는 일을 각각 설명하였다. 귀납으로 정의된 임의의 술어 P로 이루어진 에비던스에 대해서도 이런 식으로 분석·설명할 수 있어야 한다.

Exercises 3

① ⩽의 추이율. 이 문제는 (이 문제에 한해서 특별히) 아주 상세하게 설명하겠다.

```
1 Lemma le_trans : forall m n o,
2   m <= n -> n <= o -> m <= o.
```

```
3 Proof.
4   intros m n o. intros Hmn Hno.
5   induction Hno as [| n' Hno' IHno'].
6   - exact Hmn.
7   - apply le_S. exact IHno'.
8 Qed.
```

라인 4를 실행한 뒤 Coq Goals 화면은 다음과 같다.

```
m, n, o : nat
Hmn : m <= n
Hno : n <= o
(1 / 1) ------------------------
m <= o
```

우리가 증명에 사용할 수 있는 에비던스는 Hmn: m <= n과 Hno: n <= o의 2개가 있다. 이 두 에비던스 각각에 대해서 사용할 책략의 후보로 destruct, inversion, induction의 3개가 존재하므로 총 6개의 책략이 가능하다.

추이성을 증명할 때는 가운데에 있는 것을 대상으로 뭐든 작업을 하는 것이 일반적이므로 Hno를 선택하기로 한다. 그리고 세 책략들 중에 induction을 사용하는 것이 요령이라고 했으므로 라인 5에 있는 induction Hno as [| n' Hno' IHno']를 실행하면, Coq Goals 화면은 다음과 같이 기저 단계의 Goal 1과 귀납단계의 Goal 2를 보여준다. 이때 인트로 패턴 as [..]는 (Goal 2에만 해당되는 것인데) 생략해도 되지만 이럴 경우 Coq이 자동으로 변수와 가설들의 이름을 정해주는데, 이게 마음에 들지 않을 수도 있으므로 나는 주로 as [..]를 써서 이름을 지정해 주는 편이다.[8]

```
Goal 1
m, n : nat
Hmn : m <= n
(1 / 2) ------------------------
m <= n

Goal 2
m, n : nat
Hmn : m <= n
n' : nat
Hno' : n <= n'
IHno' : m <= n'
(2 / 2) ------------------------
m <= S n'
```

n'는 e2 = S n'를 만족하는 그 변수이고, Hno'는 Hno에서 e2를 n'로 치환하여 얻은 가설이다. 귀납가설 IHno'의 프랍은 원래의 고울 프랍에서 e2를 n'로 치환하여 얻은 프랍이다.

두 번째 고울 프랍에 m <= S n'가 보인다. 따라서 apply le_S를 사용할 수 있다. ✓

책략의 인수를 Hno 대신 Hmn으로 한다거나, and/or induction 대신 destruct, 혹은 inversion을 사용하면 어떻게 되며, 어디서 증명이 막히는지를 확인해 보는 것은 좋은 연습이 될 것이다.

[8]때로는 생성자의 개수가 많고 인수도 많아서 as [...]이 많이 복잡해 질 수 있으므로 이럴 때는 as [...]를 사용하지 않고 Coq이 디폴트로 제공하는 이름들을 사용한다.

② 쉬운 문제 3개. (라인 18의 Lemma n_le_Sn은 3번째 문제의 해결에 사용하려고 내가 만들어 넣은 것임.)

```
1  Theorem O_le_n : forall n,
2    0 <= n.
3  Proof.
4    intros n. induction n as [| n'].
5    - apply le_n.
6    - apply le_S. exact IHn'.
7  Qed.
8
9  Theorem n_le_m__Sn_le_Sm : forall n m,
10   n <= m -> S n <= S m.
11 Proof.
12   intros n m. intros Hnm.
13   induction Hnm as [| n' Hnm' IHnm'].
14   - apply le_n.
15   - apply le_S. exact IHnm'.
16 Qed.
17
18 Lemma n_le_Sn : forall n,
19   n <= S n.
20 Proof.
21   intros n. apply le_S. apply le_n. Qed.
```

다음 문제에 대해서는 약간의 설명을 덧붙였다.

```
1  Theorem Sn_le_Sm__n_le_m : forall n m,
2    S n <= S m -> n <= m.
3  Proof.
4    intros n m Hnm.
5    inversion Hnm.
6    - apply le_n.
7    - apply le_trans with (n:= S n).
8      + apply n_le_Sn.
9      + exact H0.
10 Qed.
```

induction으로는 증명이 잘 되지 않으므로 inversion Hnm을 사용하였다. 이때 Coq Goals는 다음과 같이 된다. (H0 : n = m은 원래 얻어지는 S n = S m에 injection이 자동으로 적용된 것이다.)

```
Goal 1
n, m : nat
Hnm : S n <= S m
H0 : n = m
(1 / 2) ------------------------
m <= m

Goal 2
n, m : nat
Hnm : S n <= S m
m0 : nat
H0 : S n <= m
```

```
H : m0 = m
(2 / 2) ------------------------
n <= m
```

Goal 2를 보면 n <= S n이 컨텍스트에 추가된다면 H0: S n <= m과 le_trans를 이용하여 고울 명제를 얻을 수 있음을 알 수 있다. 그래서 Lemma n_le_Sn을 미리 준비해 두었던 것이다. 그 다음은 쉽다. ✓

④ Dichotomy of $<$, 즉 $\forall n, m \in \mathbb{N}, (n < m) \vee (n \geqslant m)$. 이 문제에서 배워야 할 중요한 증명 기법이 있다. 고울 프랍이 논리합문disjunctive proposition일 때 직관론적 증명intuitionistic proof을 어떻게 하느냐이다. Classical logic에서는 $P \vee Q$를 증명할 때 대부분 이와 (classical logic 에서는) 동등한 $\neg P \to Q$를 증명한다. 하지만 직관론적 증명에서는 이런 식의 증명은 허용되지 않는다. 반드시 P를 증명하거나 혹은 Q를 증명해야 한다.

컨텍스트에 논리합문 $P' \vee Q'$이 있을 때는 고울 $P \vee Q$를 직관론적으로 증명할 수 있는 길이 열린다—P'인 경우에는 P를 증명하고, Q'인 경우에는 Q를 증명하는 것이다.

이 정리를 증명할 때 induction n을 실행하면 귀납가설에 논리합문이 나타나게 된다. 이를 이용한다.

이 정리에서 사용된 기호 >=는 ge의 가운데쓰기 기호이다. 이것을 <=로 바꾸려면 unfold ge를 하면 된다.

```
1  Theorem lt_ge_cases : forall n m,
2    n < m \/ n >= m.
3  Proof.
4    induction n as [| n' IHn'].
5    - intros m. destruct m as [| m'] eqn:Eq.
6      + right. unfold ge. apply O_le_n.
7      + left. unfold lt. apply n_le_m__Sn_le_Sm. apply O_le_n.
8    - intros m. destruct m as [| m'] eqn:Eq.
9      + right. (* ... *)
10     + specialize IHn' with m'.
11       destruct IHn' as [IHn'_l | IHn'_r].
12       { left. (* ... *) }
13       { right. (* ... *) }
14 Qed.
```

라인 4를 실행한 뒤 Coq Goals 화면은 다음과 같다.

```
Goal 1
(1 / 2)
forall m : nat, 0 < m \/ 0 >= m

Goal 2
n' : nat
IHn' : forall m : nat, n' < m \/ n' >= m
(2 / 2)
forall m : nat, S n' < m \/ S n' >= m
```

Goal 1의 프랍은 m = 0일 때와 m = S m'일 때로 나누어, 전자의 경우에는 오른쪽 합인자를 선택하고, 후자의 경우에는 왼쪽 합인자를 선택하여 증명할 수 있다.

Goal 2에서는 귀납가설에 논리합문이 있으므로 고울 프랍의 논리합문을 증명할 수 있다. 라인 8의 `intros m`을 실행한 뒤의 고울 화면은 다음과 같다.

```
n' : nat
IHn' : forall m : nat, n' < m \/ n' >= m
m : nat
(1 / 1) ------------------------
S n' < m \/ S n' >= m
```

이 상태에서는 `IHn'`의 `m`에 어떤 값을 주어도 고울 프랍을 얻을 수 없음을 알 수 있다. 이런 경우에 흔히 사용하는 방법은 `destruct m`을 실행하는 것이다. `m = 0`인 경우는 고울 프랍이 쉽게 증명되고, `m = S m'`인 경우에는 `IHn'`의 `m`을 `m'`로 특정하여 증명을 진행할 수 있다. ✓

⑤ 쉬운 문제 2개.

```
1  Theorem le_plus_l : forall (a b: nat),
2    a <= a + b.
3  Proof.
4    intros a b. induction b as [| b' IHb'].
5    - rewrite add_0_r. apply le_n.
6    -  rewrite <- plus_n_Sm. apply le_S. exact IHb'.
7  Qed.
8
9  Theorem plus_le : forall (n1 n2 m: nat),
10   n1 + n2 <= m ->
11   n1 <= m /\ n2 <= m.
12 Proof.
13   intros n1 n2 m H.
14   induction H as [| m' H' IH'].
15   - split.
16     + apply le_plus_l.
17     + rewrite add_comm. apply le_plus_l.
18   - destruct IH' as [IH'_l IH'_r].
19     split.
20     + (* ... *)
21     + (* ... *)
22 Qed.
```

두 번째 문제에서는 `induction` 대신 `inversion`을 써도 된다. 단, 증명은 조금 더 복잡해진다. ✓

⑥ 다음 문제는 좀 까다로와 보이는데, 우선 고울 프랍에 논리합이 있으므로 `lt_ge_cases`의 증명에서와 같은 아이디어를 사용한다. 그리고 `n`에 대해서 귀납을 하면서 `p`는 묶인변수로 놓아두고 싶으므로 `generlize dependent`를 사용한다.

```
1  Theorem add_le_cases : forall n m p q,
2    n + m <= p + q -> n <= p \/ m <= q.
3  Proof.
4    intros n m p q H.
5    generalize dependent p.
6    induction n as [| n' IHn'].
7    - intros. left. apply O_le_n.
```

```
8   - intros.
9     destruct p as [| p'].
10    + intros H. right. simpl in H. rewrite <- H.
11      replace (n' + m) with (m + n').
12      {
13        apply le_trans with (n:= m + n').
14        { (* ... *) }
15        { (* ... *) }
16      }
17      { (* ... *) }
18    + intros H. simpl in H. apply Sn_le_Sm__n_le_m in H.
19      apply IHn' with (p:= p') in H.
20      (* ... *)
21      { left. apply n_le_m__Sn_le_Sm. exact H_l. }
22      { (* ... *) }
23  Qed.
```

라인 8을 실행한 뒤 Coq Goals 화면은 다음과 같다.

```
IHn' : forall p : nat, n' + m <= p + q -> n' <= p \/ m <= q
(1 / 1) -----------------------
S n' + m <= p + q -> S n' <= p \/ m <= q
```

IHn'에서 p := pred p로 특정하면 될 듯 하지만 이렇게 하면 t_ge_cases의 증명에서와 마찬가지로 p = 0일 때가 문제가 된다. 이 문제는 destruct p로써 해결한다. ✓

⑦ 이제 평이한 문제들이다.

```
1   Theorem plus_le_compat_l : forall n m p,
2     n <= m ->
3     p + n <= p + m.
4   Proof.
5     intros n m p Hnm.
6     induction p as [| p' IHp'].
7     - simpl. exact Hnm.
8     - simpl. apply n_le_m__Sn_le_Sm. exact IHp'.
9   Qed.
10
11  Theorem plus_le_compat_r : forall n m p,
12    n <= m ->
13    n + p <= m + p.
14  Proof.
15    intros n m p Hnm.
16    rewrite add_comm. rewrite add_comm with (n:= m).
17    apply plus_le_compat_l. exact Hnm.
18  Qed.
19
20  Theorem le_plus_trans : forall n m p,
21    n <= m ->
22    n <= m + p.
23  Proof.
24    intros n m p Hnm.
25    apply le_trans with (n:= m).
```

```
26    - exact Hnm.
27    - apply le_plus_l.
28  Qed.
29
30  Theorem n_lt_m__n_le_m : forall n m,
31    n < m ->
32    n <= m.
33  Proof.
34    intros n m Hnm.
35    unfold lt in Hnm.
36    apply le_trans with (n:= S n).
37    - apply n_le_Sn.
38    - exact Hnm.
39  Qed.
40
41  Theorem plus_lt : forall n1 n2 m,
42    n1 + n2 < m ->
43    n1 < m /\ n2 < m.
44  Proof.
45    intros n1 n2 m H.
46    unfold lt in H.
47    split.
48    - unfold lt.
49      apply le_trans with (n:= S (n1 + n2)).
50      { apply n_le_m__Sn_le_Sm. apply le_plus_l. }
51      { exact H. }
52    - unfold lt.
53      apply le_trans with (n:= S (n1 + n2)).
54      { apply n_le_m__Sn_le_Sm. rewrite add_comm. apply le_plus_l. }
55      { exact H. }
56  Qed.
57
58  Theorem leb_complete : forall n m,
59    n <=? m = true -> n <= m.
60  Proof.
61    intros n m H.
62    generalize dependent n.
63    induction m as [| m' IHm'].
64    - intros n. destruct n as [| n'].
65      + intros H1. apply le_n.
66      + intros H1. unfold leb in H1. discriminate H1.
67    - destruct n as [| n'].
68      + intros H1. apply O_le_n.
69      + specialize IHm' with (n:= n').
70        intros H1.
71        apply n_le_m__Sn_le_Sm.
72        apply IHm'.
73        inversion H1. reflexivity.
74  Qed.
75
76  Theorem leb_correct : forall n m,
77    n <= m ->
```

```
78    n <=? m = true.
79  Proof.
80    intros n m H.
81    generalize dependent n.
82    induction m as [| m' IHm'].
83    - intros n. destruct n as [| n'].
84      + intros H1. simpl. reflexivity.
85      + intros H1. inversion H1.
86    - destruct n as [| n'].
87      + intros H1. simpl. reflexivity.
88      + specialize IHm' with (n:= n').
89        intros H1. inversion H1.
90        { apply leb_refl. }
91        { apply Sn_le_Sm__n_le_m in H1.
92          apply IHm' in H1.
93          simpl. exact H1. }
94  Qed.
95
96  Theorem leb_iff : forall n m,
97    n <=? m = true <-> n <= m.
98  Proof.
99    intros n m.
100   split.
101   - apply leb_complete.
102   - apply leb_correct.
103 Qed.
104
105 Theorem leb_true_trans : forall n m o,
106   n <=? m = true -> m <=? o = true -> n <=? o = true.
107 Proof.
108   intros n m o.
109   intros H1 H2.
110   apply leb_correct.
111   apply leb_iff in H1.
112   apply leb_iff in H2.
113   apply le_trans with (n:= m).
114   - exact H1.
115   - exact H2.
116 Qed.
```

Predicate Examples

다음은 자연수들의 3항 관계ternary relation의 예이다.

```
1 Inductive R : nat -> nat -> nat -> Prop :=
2   | c1                                      : R 0     0     0
3   | c2 m n o (H : R m     n     o        ) : R (S m) n     (S o)
4   | c3 m n o (H : R m     n     o        ) : R m     (S n) (S o)
5   | c4 m n o (H : R (S m) (S n) (S (S o))) : R m     n     o
6   | c5 m n o (H : R m     n     o        ) : R n     m     o
7 .
```

R은 덧셈을 나타내는 술어이다. 이는 다음과 같이 증명된다. 순방향에서는 R의 생성자가 5

개이므로 case analysis는 5가지 경우로 나누어서 진행하게 된다. 역방향에서는 nat의 생성자가 2개이므로 평소와 같이 2개의 경우로 나누어 생성자별 분석을 한다.

```
1  Definition fR : nat -> nat -> nat :=
2    fun (n m : nat) => n + m.
3
4  Theorem R_equiv_fR : forall m n o, R m n o <-> fR m n = o.
5  Proof.
6    intros m n o. split.
7    - intros H. induction H.
8      + simpl. reflexivity.
9      + simpl. rewrite IHR. reflexivity.
10     + unfold fR. rewrite <- plus_n_Sm.
11       rewrite <- IHR.
12       unfold fR. reflexivity.
13     + simpl in IHR. rewrite <- plus_n_Sm in IHR.
14       injection IHR as IHR.
15       exact IHR. (* unfold fR is automatic *)
16     + (* ... *)
17   - generalize dependent o. generalize dependent n.
18     induction m as [| m' IHm'].
19     + intros n o H. simpl in H. rewrite H.
20       generalize dependent o. (* See (1) below. *)
21       induction n as [| n' IHn'].
22       { intros. rewrite <- H. apply c1. }
23       { destruct o as [| o'].
24         - (* ... *)
25         - specialize IHn' with (o:= o'). (* (1) *)
26           intros H. inversion H. apply IHn' in H1.
27           (* ... *) }
28     + intros n o H.
29       destruct o as [| o'] eqn:Eqo.
30       { (* ... *) }
31       { apply c2. apply IHm'. simpl in H.
32         (* ... *) }
33 Qed.
```

순방향은 의외로 간단하다. 역방향은 조금 까다로운데, 생성자별 분석이 induction과 destruct가 각각 두 군데, inversion이 한 군데 쓰였다. ✓

귀납적으로 정의된 술어 R에 대한 귀납적 증명이란 정확히 무엇을 말하는 것인지 방금 공부했던 이 예를 통해서 알아 보자.

라인 7에서 (컨텍스트 내의 한 가설 H에 대하여) induction H를 실행하였다.

원래 증명하려던 명제는 $\forall x_1 x_2 x_3,$ R$(t_1, t_2, t_3) \rightarrow Q(x_1, x_2, x_3)$ 형태이다. 여기서 $t_i = t_i(x_1, x_2, x_3),\ i = 1, 2, 3$는 표현항이다.

생성자 c1, .., c5 각각에 대하여 서브고울이 하나씩 생성된다. 각 서브고울의 컨텍스트에는 ① 변수들의 타입선언과, ② H라는 가설과, ③ IHR이라는 귀납가설이 있다. 단, 생성자 c1에 대하여는, 이 생성자가 상수형(nullary)이라서 인수를 받아들이지 않으므로, 컨텍스트가 비어 있다.

변수들은 생성자의 변수 인수들이다.[9] 가설은 생성자의 인수 가설 R(s_1, s_2, s_3)이다. 귀납가설은 H가 R(t_1, t_2, t_3)과 매치되도록 x_i들을 치환했을 경우의 Q이다.

서브고울의 컨텍스트에 대한 설명은 이상으로 마쳤고, 서브고울의 고울 프랍은 생성자의 정의에서 콜론의 오른쪽에 있던 것이 R(t_1, t_2, t_3)과 매치되도록 x_i들을 치환했을 경우의 Q이다.

라인 7의 induction H에서 as [..]를 사용하여 변수들과 가설과 귀납가설에 이름을 줄 수도 있었지만 이 경우에는 생성자가 5개이고 각 생성자마다 변수가 3개씩이나 있으므로, 인트로 패턴이 너무 복잡해질 것이므로 그냥 Coq가 디폴트로 제공하는 이름들을 사용하였다.

다음은 n: nat, l: list nat를 인수로 받아들이는 2항 술어의 예이다. 정의를 보고 이 술어 R이 무엇을 나타내는지 생각해 보라.

```
1  Inductive R : nat -> list nat -> Prop :=
2      | c1                    : R 0     []
3      | c2 n l (H: R n     l) : R (S n) (n :: l)
4      | c3 n l (H: R (S n) l) : R n     l.
5
6  Fact myR1 : R 2 [1;0].
7  Proof.
8    apply c2. apply c2. apply c1. Qed.
9
10 Fact myR2 : R 1 [1;2;1;0].
11 Proof.
12   apply c3. apply c2. apply c3. apply c3.
13   apply c2. apply c2. apply c2. apply c1. Qed.
14
15 Fact myR3: R 2 [1;2;1;0].
16   Proof.
17     apply c2. apply c3. apply c3.
18   Abort.
```

Inductive Predicate for Subsequence Relation

부분수열subsequence 관계, subseq (l1: list nat) (l2: list nat)를 귀납을 써서 정의하고 이와 관련된 몇 개의 기본적인 정리를 증명해 보자.

```
1  Inductive subseq : list nat -> list nat -> Prop :=
2    | sbsq1 (l2: list nat) :
3        subseq [] l2
4    | sbsq2 (l1 l2: list nat) (x: nat) (H: subseq l1 l2) :
5        subseq l1 (x :: l2)
6    | sbsq3 (l1 l2: list nat) (x: nat) (H: subseq l1 l2) :
7        subseq (x :: l1) (x :: l2).
8
9  Theorem subseq_refl : forall (l : list nat), subseq l l.
10 Proof.
11   intros l. induction l as [| x l' IHl'].
12   - apply sbsq1.
```

[9]생성자의 인수에는 변수 인수와 가설 인수의 두 종류가 있다.

```
13    - apply sbsq3. exact IHl'.
14  Qed.
15
16  Theorem subseq_app : forall (l1 l2 l3 : list nat),
17    subseq l1 l2 ->
18    subseq l1 (l2 ++ l3).
19  Proof.
20    intros l1 l2 l3 H.
21    induction H.
22    - apply sbsq1.
23    - simpl. apply sbsq2. exact IHsubseq.
24    - simpl. apply sbsq3. exact IHsubseq.
25  Qed.
```

다음 정리는 추이성의 증명이다. 항상 하던 대로 가운데 있는 subseq l2 l3에 대해서 귀납을 적용하기로 한다. 증명을 두 부분으로 나누어 설명하겠다.

```
1  Theorem subseq_trans :  forall (l1 l2 l3: list nat),
2    subseq l1 l2 -> subseq l2 l3 -> subseq l1 l3.
3  Proof.
4    intros l1 l2 l3. intros Ha Hb.
5    generalize dependent Ha.
6    generalize dependent l1.
```

이 증명에서 특이한 것은 라인 5이다. generalize dependent를 Set 타입의 변수가 아니라 가설, 즉 Prop 타입의 변수인 Ha에 적용하였다. 이것이 어떤 기능을 하는 것인지 알기 위해서 먼저 라인 4를 실행한 후의 고울 화면을 보자.

```
Ha : subseq l1 l2
Hb : subseq l2 l3
(1 / 1) ------------------------
subseq l1 l3
```

Ha가 컨텍스트에 들어 있다. 여기서 generalize dependent Ha를 실행하면 Ha가 컨텍스트에서 고울 프랍의 전건으로 옮겨간다.[10]

```
Hb : subseq l2 l3
(1 / 1) ------------------------
subseq l1 l2 -> subseq l1 l3
```

이렇게 고울 프랍을 조건문으로 바꿔 놓은 이유는 나중에 사용하게 될 귀납가설을 더 강하게 만들기 위함이다. 이것은 generalize dependent를 Set 타입의 변수에 적용할 때 전칭한정사를 고울 프랍에 붙여서 귀납가설을 더 강하게 만드는 것과 같은 원리이다.

그 다음 generalize dependent l1을 실행하여 고울 프랍(나중에 귀납가설이 될)을 한층 더 강하게 만들어 놓고 나서 에비던스 Hb에 대한 귀납을 시작한다.

[10] 이것은 generalize dependent를 Set 타입 변수에 적용하면 그 변수의 타입 선언이 컨텍스트에서 고울 프랍의 전칭한정사로 옮겨가는 것과 비슷하다. 그러므로 대략 전칭한정사는 조건문의 전건과 같다고 볼 수 있다. 타입에서도, 한 예를 들면 nat -> nat : Type과 forall n: nat, nat: Type 사이에 미묘한 관련이 있었다. 이 이슈에 대해서 나중에 다시 설명하겠다.

```
7     induction Hb.
8     - intros.
9       inversion Ha.
10      (* ... *)
11    - intros.
12      apply sbsq2.
13      apply IHHb.
14      (* ... *)
15    - intros.
16      inversion Ha.
17      + (* ... *)
18      + (* ... *)
19      + (* ... *)
20  Qed.
```

이 문제는 귀납가설을 강화하기 위한 generalize dependent에 더하여 inversion을 적재적소에 사용해야 하므로 그리 쉽지 않다고 본다.

8.4 Case Study: Regular Expressions

정규표현식Regular Expression은 컴퓨터 과학에서 가장 많이 쓰이는 개념 중 하나이다. 정규표현식은 이 책의 용어 관례를 따르자면 표현항term이다. 나는 정규표현식을 레겍스(*Regex*)라고 부르겠다. 레겍스 대신 흔히 패턴pattern이라는 용어를 사용하기도 한다.

각 레겍스는 특정한 문자열의 집합을 어떤 방법으로 기술한다. 이 집합에 속하는 문자열은 그 레겍스에 매치(*match*)된다고 말한다.

이 절에서 우리는 레겍스를 Coq에서 어떻게 다룰 수 있는지 살펴보겠다. 하나 알아두어야 할 것은 여기서 다루는 레겍스는 실제 프로그래밍에서 쓰이는 레겍스의 기반을 제공하는 것으로서 실제 쓰이는 레겍스와 많이 달라 보일 것이라는 것이다. 프로그래밍에서 널리 사용되고 있는 레겍스의 구문은 사용의 편리성에 중점을 둔 것이고, Coq이 사용하는 레겍스 구문은 엄격성에 중점을 둔 것이라고 보면 될 것이다.

The reg_exp term and the exp_match relation

레겍스는 다형 타입이다. 인수로 받는 타입은 기호의 집합이라고 생각하면 된다. 통상 타입 T로는 상수 생성자만을 가진 유한집합을 사용한다. 예를 들어 $T := \{0, \ldots, 9, a, \ldots, z, A \ldots, Z\}$로 둘 수 있다.

```
Inductive reg_exp (T : Type) : Type :=
  | EmptySet
  | EmptyStr
  | Char (t : T)
  | App (r1 r2 : reg_exp T)
  | Union (r1 r2 : reg_exp T)
  | Star (r : reg_exp T).

Arguments EmptySet {T}.
Arguments EmptyStr {T}.
```

```
Arguments Char {T} _.
Arguments App {T} _ _.
Arguments Union {T} _ _.
Arguments Star {T} _.
```

레겍스 re: reg_exp와 문자열 s: list T의 관계, 즉 매치 관계는 다음 규칙에 의해 정의된다.

- EmptySet은 어떤 문자열도 매치하지 않는다. 즉 EmptySet에 매치되는 문자열은 없다.
- 공문자열 []은 EmptyStr에 매치된다.
- 길이 1인 문자열 [x]는 Char x에 매치된다.
- s_1이 re1: reg_exp에 매치되고 s_2가 re2 : reg_exp가에 매치되면 s_1++ s_2는 App re1 re2에 매치된다.
- s_1이 re1: reg_exp에 매치되고 s_2가 re2 : reg_exp가에 매치되면 s_1이 Union re1 re2에 매치되고 또한 s_2도 Union re1 re2에 매치된다.
- $s_1, \ldots, s_k$가 모두 re에 매치되면 $s_1 ++ \cdots ++ s_k$는 Star re에 매치된다. $k = 0$인 경우에는 공문자열이 Star re에 매치된다.

이상의 정의를 귀납적 술어 exp_match로 다음과 같이 나타낼 수 있다.

```
Reserved Notation "s =~ re" (at level 80).

Inductive exp_match {T} : list T -> reg_exp T -> Prop :=
  | MEmpty
      : [] =~ EmptyStr
  | MChar x
      : [x] =~ (Char x)
  | MApp s1 re1 s2 re2 (H1 : s1 =~ re1) (H2 : s2 =~ re2)
      : (s1 ++ s2) =~ (App re1 re2)
  | MUnionL s1 re1 re2 (H1 : s1 =~ re1)
      : s1 =~ (Union re1 re2)
  | MUnionR re1 s2 re2 (H2 : s2 =~ re2)
      : s2 =~ (Union re1 re2)
  | MStar0 re : [] =~ (Star re)
  | MStarApp s1 s2 re (H1 : s1 =~ re) (H2 : s2 =~ (Star re))
      : (s1 ++ s2) =~ (Star re)

  where "s =~ re" := (exp_match s re).
```

레겍스 re가 문자열 s를 매치할 때 s가 re에 매치된다고 말한다. s =~ re를 s ∈ re로 생각하는 것도 괜찮은 방법이라고 본다.

reg_exp의 생성자 6개 각각에 대해서 exp_match의 생성자 7개 중 0개, 1개, 혹은 2개가 대응하는 것에 주목하라.

문자열은 리스트이므로 원래 [1;2;A;b]의 형식으로 써야 하지만 이를 간단히 12Ab와 같이 나타내기로 하자. 그러면 다음의 매치 관계들이 성립한다.

```
1 =~ Char 1
12 =~ App (Char 1) (Char 2)
```

```
3 a =~ Union (Char a) (App (Char A) (Char B))
4 AB =~ Union (Char a) (App (Char A) (Char B))
5 abab =~ Union (Star (App (Char a) (Char b))) (App (Char A) (App (Char B) (Char C)))
```

라인 3과 라인 4에서 사용한 레젝스는 동일하다. 이 레젝스는 딱 2개의 문자열과 대응된다. 라인 5의 레젝스는 무한히 많은 문자열들과 대응되는데, 그 중 몇 개를 나열해 보면

```
ABC, [], ab, abab, ababab, ...
```

등이 있다.

위에서 예로 든 관계들이 성립함을 하나씩 증명해 보자.

① 첫 번째 예의 증명은 exp_match의 정의로부터 즉시 얻어진다.

```
1 Example reg_exp_ex1 : [1] =~ Char 1.
2 Proof.
3 apply (MChar 1). (* or apply MChar. *)
4 Qed.
```

라인 3에서 MChar의 인수 1은 생략해도 Coq이 알아서 찾아준다.

② 두 번째 예의 증명은 여러 버전으로 보여 주겠다.

```
Example reg_exp_ex2 : [1;2] =~ App (Char 1) (Char 2).
Proof.
  apply (MApp [1] (Char 1) [2] (Char 2) (MChar 1) (MChar 2)).
Qed.
```

첫 버전에서는 생성자 MApp의 인수 6개를

s1 := [1], re1 := Char 1, s2 := [2], re2 := Char 2, H1 := MChar 1, H2 := MChar 2

로 두어 라인 하나에서 한꺼번에 다 사용하였다.

다음은 MApp의 인수를 4개만 사용하는 버전이다.

```
1 Example reg_exp_ex2' : [1;2] =~ App (Char 1) (Char 2).
2 Proof.
3   apply (MApp [1] (Char 1) [2] (Char 2)).
4   - apply (MChar 1).
5   - apply (MChar 2).
6 Qed.
```

귀납적 술어의 생성자의 인수에는 값을 나타내는 인수와 에비던스를 나타내는 인수의 두 종류가 있다. 다시 말하면 n : nat, l : list nat 등 Set 타입의 원소와, H: 1 = 1, E : ~ ev 3 등 Prop 타입의 원소가 있다.

이 버전의 증명 스크립트에서는 MApp의 인수들 중 값을 나타내는 인수 4개를 라인 4에 사용하였다. 그리고 남은 2개의 에비던스를 나타내는 인수들은 각각 하나의 서브고울에서 처리하도록 넘겼다. 이렇게 하는 것이 증명 스크립트를 작성하는 통상적인 방법이다.

증명의 세 번째 버전은 다음과 같다.

```
Example reg_exp_ex2'' : [1;2] =~ App (Char 1) (Char 2).
Proof.
  apply (MApp [1]).
  - apply MChar.
  - apply MChar.
Qed.
```

이것은 생성자의 인수를 Coq이 자동으로 채워주는 기능을 이용한 것이다.

라인 3에서 MApp의 인수를 아예 한 개도 사용하지 않는 다음과 같은 방법도 있다.

```
Example reg_exp_ex2''' : [1;2] =~ App (Char 1) (Char 2).
Proof.
  replace [1;2] with ([1] ++ [2]).
  { apply MApp.
    - apply MChar.
    - apply MChar. }
  { reflexivity. }
Qed.
```

③ 세 번째 예 이후에는 알파벳으로 {a,b,A,B,C}을 사용한다.

```
Inductive my_alphabet : Type := a | b | A | B | C.

Example reg_exp_ex_my3 :
  [a] =~ Union (Char a) (App (Char A) (Char B)).
Proof.
  apply MUnionL.
  apply MChar.
Qed.

Example reg_exp_ex_my4 :
  [A;B] =~ Union (Char a) (App (Char A) (Char B)).
Proof.
  apply MUnionR.
  apply (MApp [A]).
  - apply MChar.
  - apply MChar.
Qed.
```

세 번째와 네 번째 예를 보면 Union을 사용하는 레겍스에 대한 매치의 증명은 어려울 것 없다는 것을 알 수 있을 것이다.

④ Star의 경우는 조금 더 복잡하다. 다음의 예는 답을 보지 않고 스스로 한 번 풀어보기를 권한다.

```
Example reg_exp_ex_my5 :
  [a;b;a;b] =~ Union (Star (App (Char a) (Char b)))
                     (App (Char A) (App (Char B) (Char C))).
Proof.
  apply MUnionL.
  apply (MStarApp [a;b]).
  - apply (MApp [a]).
```

```
8      { apply MChar. } { apply MChar. }
9    - apply (MStarApp [a;b]).
10     + apply (MApp [a]).
11     { apply MChar. } { apply MChar. }
12     + apply MStar0.
13 Qed.
```

Star를 포함하는 레겍스에 대한 매치를 증명할 때는 반드시 MStarApp을 0번 이상 사용하고 마지막에 MStar0를 한 번 사용해야 함에 주목하라.

다음 예는 부정문의 증명에서 inversion의 유용함을 보여준다. inversion 없이도 이 증명은 (더 복잡하기는 하지만) 가능함을 나중에 p179에서 보게 될 것이다.

```
Example reg_exp_ex3 : ~ ([1; 2] =~ Char 1).
Proof.
  intros H.
  inversion H.
Qed.
```

문자열을 받아서 그것에 매치되는 레겍스를 만들어 리턴하는 함수를 다음과 같이 정의할 수 있다. 이 함수가 만들어 주는 레겍스와 매치되는 문자열은 원래 받았던 문자열 하나밖에 없다.

```
Fixpoint reg_exp_of_list T (l : list T) :=
  match l with
  | [] => EmptyStr
  | x :: l' => App (Char x) (reg_exp_of_list l')
  end.
```

위에서 정의한 함수에 대한 유닛 테스트 하나를 보자. 모든 유닛 테스트가 그러하듯이 이것도 참 재미없는 증명이다.

```
1  Example reg_exp_ex4 : [1; 2; 3] =~ reg_exp_of_list [1; 2; 3].
2  Proof.
3    simpl.
4    apply (MApp [1]).
5    { apply MChar. }
6    { apply (MApp [2]).
7      { apply MChar. }
8      { apply (MApp [3]).
9        { apply MChar. }
10       { apply MEmpty. } }
11   }
12 Qed.
```

참고로 라인 3을 실행한 뒤 Coq Goals 화면은 다음과 같다.

```
[1; 2; 3] =~ App (Char 1) (App (Char 2) (App (Char 3) EmptyStr))
```

위 식의 우변은 Compute reg_exp_of_list [1;2;3]을 실행하여 얻을 수 있다.

다음은 Star re는 re의 초집합이라는 사실의 증명이다. (앞으로는 레겍스를 문자열들의 집합으로 취급하겠다.) 이 증명에서 Poly에서 증명했던 정리 app_nil_r을 사용했다. 이것의 목적은 고울 프랍 등식 s =~ Star re의 좌변 s를 s ++ []로 대체하기 위함이다. 이렇게 해야만 라인 9에서 MStarApp의 인수가 제대로 infer 될 수 있기 때문이다.

```
Check app_nil_r. : forall (X : Type) (l : list X), l ++ [ ] = l

Lemma MStar1 :
  forall (T: Set) (s: list T) (re: reg_exp T),
    s =~ re -> s =~ Star re.
Proof.
  intros T s re H.
  rewrite <- (app_nil_r T s).
  apply MStarApp.
  - apply H.
  - apply MStar0.
Qed.
```

다음은 평이한 사실 2개의 증명이다.

```
Lemma empty_is_empty : forall T (s : list T),
  ~ (s =~ EmptySet).
Proof.
  intros T s H. (* H: s =~ EmptySet, Goal: False *)
  inversion H. Qed.

Lemma MUnion' : forall T (s : list T) (re1 re2 : reg_exp T),
  s =~ re1 \/ s =~ re2 ->
  s =~ Union re1 re2.
Proof.
  intros T s re1 re2 H.
  (* H: s =~ re1 \/ s =~ re2, Goal: s =~ Union re1 re2 *)
  destruct H as [H1 | H2].
  - (* ... *)
  - apply MUnionR. exact H2.
Qed.
```

다음은 re의 원소들의 연접concatenation은 Star re의 원소라는 사실의 증명이다. 정리에 앞서 여기서 사용하는 fold와 In의 사용 예를 보였다.

```
Compute (fold app [[1;2];[3];[4;5]] []). (* = [1;2;3;4;5] *)
Compute (In 1 [1;2]). (* = 1 = 1 \/ 2 = 1 \/ False *)

Lemma MStar' : forall T (ss : list (list T)) (re : reg_exp T),
  (forall s, In s ss -> s =~ re) ->
    fold app ss [] =~ Star re.
Proof.
  intros T ss re H.
  induction ss as [| s ss' IH].
  - (* ss = [] case. Show [] =~ Star re *)
    simpl. apply MStar0.
  - (* ss = s :: ss' case. Show s :: ss' =~ Star re *)
    simpl.
    apply MStarApp.
    + apply H. (* ... *)
    + apply IH. intros s' H'.
      apply H. (* ... *)
Qed.
```

Exercises 4

① 다음의 사실을 증명해 보자.

> 만일 s =~ re라면 s의 모든 각각의 원소는 re의 어딘가에 character literal로 나타난다.

먼저 re에 나타나는 모든 char literal 들의 (중복) 리스트를 얻는 함수를 정의한다.

```
Fixpoint re_chars {T} (re : reg_exp T) : list T :=
  match re with
  | EmptySet => []
  | EmptyStr => []
  | Char x => [x]
  | App re1 re2 => re_chars re1 ++ re_chars re2
  | Union re1 re2 => re_chars re1 ++ re_chars re2
  | Star re => re_chars re
  end.
```

지금까지는 가설이 Hm: s =~ regex 형태인 경우에 이에 적용할 책략으로 항상 inversion Hm을 사용해 왔다. 다음의 정리는 inversion Hm 대신 induction Hm을 사용하는 첫 예이다. 내용은 많지 않으나 이러한 유형은 처음이므로 증명을 몇 부분으로 나누어 상세히 설명하겠다.

```
1  Theorem in_re_match : forall T (s : list T) (re : reg_exp T) (x : T),
2    s =~ re -> In x s -> In x (re_chars re).
3  Proof.
4    intros T s re x Hm Hin.
5    induction Hm (* Hm: s =~ re, Hin : In x s, Goal: In x (re_chars re) *)
6      as [(* 1.MEmpty *)
7          (* 2.MChar *)    | x'
8          (* 3.MApp *)     | s1 re1 s2 re2 Hm1 IH1 Hm2 IH2
9          (* 4.MUnionL *)  | s1 re1 re2 Hm IH
10         (* 5.MUnionR *)  | re1 s2 re2 Hm IH
11         (* 6.MStar0 *)   | re
12         (* 7.MStarApp *) | s1 s2 re Hm1 IH1 Hm2 IH2].
13   - (* MEmpty, Hin: In x [ ], Goal: In x (re_chars EmptyStr) *)
14     unfold In in Hin. destruct Hin.
15   - (* MChar, Hin: In x [x'], Goal: In x (re_chars (Char x')) *)
16     simpl. simpl in Hin.
17     exact Hin.
```

exp_match라는 술어는 7개의 생성자를 써서 Inductively 정의되었으므로 인트로 패턴이 상당히 길다. 이 패턴 as [..]가 라인 6 ~ 12처럼 됨은 정의로부터 쉽게 확인된다.

라인 13 ~ 17을 설명하겠다. 첫 두 생성자의 정의를 보면 다음과 같다.

```
| MEmpty : [] =~ EmptyStr
| MChar x : [x] =~ (Char x)
```

콜론의 오른쪽 부분이 s =~ re와 매치 되어야 하므로 1.MEmpty에서는 s := [], re := EmptyStr이어야 한다. 그러므로 컨텍스트의 Hin은 In x []이어야 한다. 이것은 정의를 보면 False로 계산된다. 이것으로써 라인 14까지 설명되었다. (unfold ..는 생략해도 된다.)

2.MChar에서는 매치 조건이 s := [x'], re := Char x'를 함의한다. 따라서 고울 프랍은 In x (re_chars (Char x'))가 된다. 이 표현항의 값은 re_chars의 정의를 보면 In x [x']로 계산됨을 알 수 있다. 따라서 고울 프랍은 In x [x']이 되고 때마침 이것은 Hin과 일치한다. 이것으로써 라인 17까지 설명되었다.

그 다음 생성자는 조금 더 복잡한 분석을 요한다. 이 증명에서는 이전에 증명해 두었던 정리 In_app_iff를 사용한다. 이 정리는 고울 프랍에도 적용하고 또한 HIn에도 적용해야 하므로 라인 21에서 rewrite In_app_iff in *를 사용한다. 그러면 Hin이 논리합문이 되므로 destruct Hin을 사용하여 두 경우로 나누어 진행한다.

```
18    - (* MApp *)
19      simpl.
20      (* In_app_iff : In a (l ++ m) <-> In a l \/ In a m *)
21      rewrite In_app_iff in *.
22      destruct Hin as [Hin | Hin].
23      + (* Hin : In x s1 *)
24        left. apply (IH1 Hin).
25      + (* Hin : In x s2 *)
26        right. apply (IH2 Hin).
```

라인 29와 라인 32에서 apply (H1 H2)를 사용했는데 이 구문은 H1: P -> Q, H2: P, Goal: Q일 때 편리하게 사용할 수 있으며 두 개의 명령. apply H1 in H2. apply H2.를 대체한다.

그 다음은 특별히 새로운 것은 없다. 단, MStarApp 경우에는 다시 MApp와 비슷하게 논리합문 가설을 destruct해야 한다.

```
27    - (* MUnionL *)
28      simpl. rewrite In_app_iff.
29      (* ... *)
30    - (* MUnionR *)
31      simpl. rewrite In_app_iff.
32      (* ... *)
33    - (* MStar0 *)
34      destruct Hin.
35    - (* MStarApp *)
36      simpl.
37      rewrite In_app_iff in Hin.
38      destruct Hin as [Hin | Hin].
39      + (* ... *)
40      + (* ... *)
41  Qed.
```

② 다음 문제는 레겍스에 매치되는 문자열이 존재한다는 것을 뜻하는 부울값 함수를 재귀적으로 정의하고 이 함수의 값이 true라는 것이 exists s, s =~ re와 동등함을 증명하는 것이다.

```
Fixpoint re_not_empty {T: Type} (re: reg_exp T) : bool :=
  match re with
  | EmptySet => false
  | EmptyStr => true
  | Char _ => true
  | App re1 re2 => andb (re_not_empty re1) (re_not_empty re2)
  | Union re1 re2 => orb (re_not_empty re1) (re_not_empty re2)
```

```
  | Star re => true
  end.
```

Star re가 항상 공아닌집합이 되는 이유는 그것이 re가 어떻든 간에 공문자열에 매치되기 때문이다.

아래 제시한 정리의 증명은 길이가 제법 길지만 평이하다. 다만 induction이 두 번 사용되는데, 첫 번째 것은 exp_match 술어에 대한 것이고 두 번째 것은 reg_exp 표현에 대한 것이라는 것 정도만 알면 되겠다.

```
1  Lemma re_not_empty_correct : forall T (re : reg_exp T),
2    (exists s, s =~ re) <-> re_not_empty re = true.
3  Proof.
4    intros T re. split.
5    - intros [s H].
6      induction H (* 7 cases *)
7        as [ (* 1. MEmpty *)
8            |(* 2. MChar *) x'
9            |(* 3. MApp *) s1 re1 s2 re2 Hm1 IH1 Hm2 IH2
10           |(* 4. MUnionL *) s1 re1 re2 Hm IH
11           |(* 5. MUnionR *) re1 s2 re2 Hm IH
12           |(* 6. MStar0 *) re
13           |(* 7. MStarApp *) s1 s2 re Hm1 IH1 Hm2 IH2].
14     + (* MEmpty *)
15       simpl. reflexivity.
16     + (* MChar *)
17       simpl. reflexivity.
18     + (* MApp *)
19       simpl. rewrite IH1. rewrite IH2. simpl. reflexivity.
20     + (* MUnionL *)
21       simpl. rewrite IH. simpl. reflexivity.
22     + (* MUnionR *)
23       simpl. rewrite IH. destruct (re_not_empty re1).
24       * (* ... *)
25       * (* ... *)
26     + (* MStar0 *)
27       (* ... *)
28     + (* MStarApp *)
29       (* ... *)
30   - intros H. induction re. (* 6 cases *)
31     + (* EmptySet *) simpl in H. discriminate H.
32     + (* EmptyStr *) exists []. apply MEmpty.
33     + (* Char t *) exists [t]. apply MChar.
34     + (* App re1 re2 *) simpl in H. apply andb_true_iff in H.
35       (* ... *)
36     + (* Union re1 re2 *) simpl in H. apply orb_true_iff in H.
37       (* ... *)
38     + (* Star re *) exists []. apply MStar0.
39 Qed.
```

The remember Tactic

induction 책략은 훌륭하고 유용하다. 하지만 다음과 같은 경우를 제대로 처리하지 못한다.

```
Lemma star_app: forall T (s1 s2 : list T) (re : reg_exp T),
  s1 =~ Star re ->
  s2 =~ Star re ->
  s1 ++ s2 =~ Star re.
Proof.
  intros T s1 s2 re H1.
  (* H1 : s1 =~ Star re *)
  (* Goal : s2 =~ Star re -> s1 ++ s2 =~ Star re *)
  induction H1
    as [(* MEmpty *)
        |(* MChar *) x'
        (* 5 remaining cases *)].
  - (* MEmpty *)
    simpl. intros H. apply H.
  - (* MChar *)
    intros H. simpl. (* Stuck... *)
Abort.
```

MEmpty 경우에는 s1 =~ Star re가 [] =~ EmptyStr에 매치되어야 하는데 이것은 (우변에서) 불가능하다. 하지만 이걸 잡아내지 못하고 s1 := []로 특정되어 진행된다. 그러면 고울 프랍은 s2 = EmptyStr -> [ ] ++ s2 = EmptyStr가 되어 어쨌든 증명이 되기는 한다.

MChar 경우에는 라인 15에서 고울 프랍이 다음과 같다.

```
s2 =~ Char x' -> [x'] ++ s2 =~ Char x'
```

이것은 참이 아니므로 당연히 증명할 수 없을 것이다. 이렇게 된 것은 애초에 s1 =~ Star re가 [x'] =~ Char x'와 매치되지 않는 것을 잡아내지 못해고 그대로 진행했기 때문이다.

이 문제에 대하여 다음과 같은 해결책이 있긴 하다. 좀 길지만 어렵지 않으니 차분히 읽어보자. 이 해법은 증명해야 할 프랍을 내용면에서는 동등하지만 다른 구문으로 바꾸어 놓고 증명하는 것이다.

```
Lemma star_app': forall T (s1 s2 : list T) (re re' : reg_exp T),
  re' = Star re ->
  s1 =~ re' ->
  s2 =~ Star re ->
  s1 ++ s2 =~ Star re.
Proof.
  intros T s1 s2 re re' H0 H1.
  induction H1
    as [(* MEmpty *)
        (* MChar *) x'
        (* 5 remaining cases *)].
  - (* MEmpty *)
    discriminate H0.
  - (* MChar. *)
    discriminate H0.
  - (* MApp. *)
```

```
17      discriminate H0.
18    - (* MUnionL. *)
19      discriminate H0.
20    - (* MUnionR. *)
21      discriminate H0.
22    - (* MStar0. *)
23      simpl. intros H. exact H.
24    - (* MStarApp. *)
25      rewrite <- app_assoc. (* ++ is right associative *)
26      intros H.
27      apply MStarApp.
28      + injection H0 as H0. rewrite <- H0. exact Hm1.
29      + apply IH2.
30        * exact H0.
31        * exact H.
32  Qed.
```

라인 7을 실행한 후 Coq Goals 화면은 다음과 같다.

```
H0 : re' = Star re
H1 : s1 =~ re'
(1 / 1) ------------------------
s2 =~ Star re -> s1 ++ s2 =~ Star re
```

H1: s1 =~ re'의 우변은 변수이므로 생성자의 프랍, 예를 들어 [x'] =~ Char x' 와 매치시킬 때 일단은 아무런 문제가 없다. 그리고 증명과정에서 컨텍스트에, 이전에는 없었던 중요한 정보를 가지게 된다.

예를 들어 라인 12의 경우 고울 화면은 다음과 같게 되는데 이때 H0가 바로 그 중요한 정보이다.

```
H0 : EmptyStr = Star re
(1 / 1) ------------------------
s2 =~ Star re -> [ ] ++ s2 =~ Star re
```

여기서 discriminate H0로써 간단하게 이 서브고울을 마무리 할 수 있다. 이런 식으로 해서 해결되지 않는 경우는 당연히 MStar0와 MStarApp의 두 경우뿐이며, 둘 다 여렵지 위의 스크립트에서 보듯이 쉽게 해결된다.

이러한 해결방법이 있기는 하지만 좀 깔끔하지 못하다는awkward 단점이 있다.

이러한 문제에 대하여 remember 책략은 아주 훌륭한 해결책이 된다. 이 책략은 수학에서 흔히 쓰는 '복잡한 표현에 적당한 이름을 주어 사용하기'와 일맥상통한다. 다음에 보인 remember 책략을 이용한 해법은 아주 자연스러워서 설명 없이 그냥 코드를 읽기만 해도 이해할 수 있을 것이다. 이 증명에서는 컨텍스트에 지난 번과는 달리 Eq : re' = Star re라는 훌륭한 정보가 들어 있다. 이것은 대부분의 생성자에 대하여 모순되는 등식으로 탈바꿈 되어 우리의 증명에 편리하게 이용할 수 있다.

```
1  Lemma star_app: forall T (s1 s2 : list T) (re : reg_exp T),
2    s1 =~ Star re ->
3    s2 =~ Star re ->
4    s1 ++ s2 =~ Star re.
```

```
5  Proof.
6    intros T s1 s2 re H1.
7    remember (Star re) as re' eqn:Eq.
8    (* Coq Goals is as follows. *)
9    Eq : re' = Star re
10   H1 : s1 =~ re'
11   (1 / 1) ------------------------
12   s2 =~ re' -> s1 ++ s2 =~ re'
13   induction H1
14     as [|x'|s1 re1 s2' re2 Hm1 IH1 Hm2 IH2
15         |s1 re1 re2 Hm IH|re1 s2' re2 Hm IH
16         |re''|s1 s2' re'' Hm1 IH1 Hm2 IH2].
17   - (* MEmpty *)  discriminate Eq.
18   - (* MChar *)   discriminate Eq.
19   - (* MApp *)    discriminate Eq.
20   - (* MUnionL *) discriminate Eq.
21   - (* MUnionR *) discriminate Eq.
22   - (* MStar0 *)
23     intros H. simpl. apply H.
24   - (* MStarApp *)
25     intros H1. rewrite <- app_assoc.
26     apply MStarApp.
27     + apply Hm1.
28     + apply IH2.
29       { exact Eq. }
30       { exact H1. }
31 Qed.
```

지금까지 induction 책략을 적용할 때 정보의 손실을 막기 위하여 remember .. as .. eqn:.. 책략을 사용하는 것에 대하여 설명하였는데, 이 책략은 induction 뿐만 아니라 destruct의 경우에도 중요한 정보의 손실을 막아 준다.

앞서 보았던 증명 하나를 예로 들어 보겠다.

```
1  Example reg_exp_ex3 : ~ ([1; 2] =~ Char 1).
2  Proof.
3    intros H.
4    inversion H.
5  Qed.
```

이 증명은 라인 4의 inversion 한 방으로 완료되는데 디테일을 볼 수 없어서 답답한 면이 있었는데 이제 inversion 대신 더 기본적인 책략인 destruct를 사용하여 증명해 보기로 한다. 라인 4의 inversion을 destruct로 대체하여 실행하면 고울 화면이 다음과 같이 된다.

```
Goal 1
(1 / 7) ------------------------
False

Goal 2
x : nat
(2 / 7) ------------------------
False
...
```

두 고울에서 모두, 컨텍스트에 아무런 가설이 없으니 없으니 증명을 진행할 수 없다. 컨텍스트에 들어가야 할 소중한 정보를 destruct하면서 잃어버린 것이다. 이것을 방지하기 위하여 destruct 이전에 remember를 사용하면 된다.

```
Example reg_exp_ex3' : ~ ([1; 2] =~ Char 1).
Proof.
  intros Hm.
  remember (Char 1) as re eqn:Eq_re.
  remember [1; 2] as s eqn:Eq_s.
  destruct Hm.
  - (* MEmpty *) discriminate Eq_re.
  - (* MChar *) injection Eq_s as Eq_s. discriminate H.
  - (* MApp *) discriminate Eq_re.
  - (* MUnionL *) discriminate Eq_re.
  - (* MUnionR *) discriminate Eq_re.
  - (* MStar0 *) discriminate Eq_re.
  - (* MStarApp *) discriminate Eq_re.
Qed.
```

물론 inversion을 쓰는 것보다 불편하다. 다만 증명의 과정을 더 자세히 볼 수 있다는 점에서는 이 방법이 더 좋다고 볼 수 있다.

다음 예는 좀 더 이전에 나왔던 것이다. 부정문을 증명할 때 주로 inversion을 간편하게 많이 쓰는데 이것도 destruct와 remember로 대체할 수 있다.

```
Example not_le_1_0 : ~ (1 <= 0).
Proof.
Proof.
  intros H.
  remember 1 as n eqn:E. remember 0 as m eqn:F.
  destruct H.
  - (* H = le_n *) rewrite F in E. discriminate E.
  - (* H = le_S *) discriminate F. Qed.
```

다음 문제는 앞서 나왔던 MStar', 즉 're의 원소들을 이어 붙이면 Star re의 원소가 된다.'의 역으로 볼 수 있다. 이번 증명은 remember를 써야 한다는 점에서 전과 다르다.

```
Lemma MStar'' : forall T (s : list T) (re : reg_exp T),
  s =~ Star re ->
  exists ss : list (list T),
    s = fold app ss [] /\
    forall s', In s' ss -> s' =~ re.
Proof.
  intros T s re Hm.
  remember (Star re) as re' eqn:Eq.
  induction Hm
    as [|x'|s1 re1 s2 re2 Hm1 IH1 Hm2 IH2
        |s1 re1 re2 Hm IH|re1 s2 re2 Hm IH
        |re''|s1 s2 re'' Hm1 IH1 Hm2 IH2].
  - discriminate Eq.
  - discriminate Eq.
  - discriminate Eq.
  - discriminate Eq.
```

```
17    - discriminate Eq.
18    - (* MStar0 *)
19      injection Eq as Eq. exists []. split.
20      + (* ... *)
21      + (* ... *)
22    - (* MStarApp *)
23      specialize (IH2 Eq). destruct IH2 as [ss [H1 H2]].
24      exists (s1 :: ss). split.
25      + (* ... *)
26      + intros s' H. destruct H as [H_hd | H_tl].
27        { (* ... *) }
28        { (* ... *) }
29  Qed.
```

다음은 공아닌 문자열 s가 Star re에 매치되면 어떤 공아닌 sa =~ re와 sb =~ Star re에 대하여 s = sa ++ sb임을 보이는 것이다. 이 증명에서도 remember를 사용한다.

이 증명의 핵심은 MStarApp 생성자 경우에 s = s1 ++ s2로 두었을 때, ① s1 = []일 때는 귀납가설을 사용하고, ② s1 <> []일 때는 sa := s1, sb := s2로 두는 것이다. 아주 간단한 증명이지만 길이는 좀 된다.

```
1   Check Gt.gt_Sn_O. (* forall n : nat, S n > 0 *)
2
3   Lemma MStarApp1 : forall T (s : list T) (re : reg_exp T),
4     length s > 0 -> s =~ Star re ->
5     exists (sa sb: list T),
6       length sa > 0 /\ sa =~ re /\ sb =~ Star re /\ s = sa ++ sb.
7   Proof.
8     intros T s re H1 H2.
9     remember (Star re) as re' eqn:Eq.
10    induction H2
11      as [|x'|s1 re1 s2 re2 Hm1 IH1 Hm2 IH2
12          |s1 re1 re2 Hm IH|re1 s2 re2 Hm IH
13          |re''|s1 s2 re'' Hm1 IH1 Hm2 IH2].
14    - (* MEmpty *) discriminate Eq.
15    - (* MChar *) discriminate Eq.
16    - (* MApp *) discriminate Eq.
17    - (* MUnionL *) discriminate Eq.
18    - (* MUnionR *) discriminate Eq.
19    - (* MStar0 *) inversion H1.
20    - (* MStarApp *)
21      destruct s1 as [| x1 s1'] eqn:Eq_s1.
22      + (* s1 = [] case *) simpl in H1.
23        specialize (IH2 H1).
24        specialize (IH2 Eq).
25        simpl. exact IH2.
26      + (* s1 = x1 :: s1' case *)
27        injection Eq as Eq. rewrite -> Eq in *.
28        exists s1, s2.
29        repeat split.
30        { rewrite Eq_s1. simpl. apply Gt.gt_Sn_O. }
31        { rewrite Eq_s1. exact Hm1. }
32        { exact Hm2. }
```

```
33       { rewrite Eq_s1. reflexivity. }
34 Qed.
```

라인 29에 repeat split 책략을 사용했는데 이것은 고울 프랍이 여러 개의 곱인자로 이루어진 논리곱문일 때 여러 개의 서브고울을 만들어 주는 책략이다.

Pumping Lemma for Regular Languages

여기서 증명할 정리는 *pumping lemma*라고 불리는, 오토마타automata 이론에서 배우게 되는 중요한 정리로서 어느 정도의 수학적 컨텐츠가 있다. 지금까지 이 책에서 우리가 다룬 거의 모든 정리들은 직관적으로 명확하여, 수학 논문에서 엄격하게 증명하지 않고 그냥 trivial하다고 여기고 넘어갈 만한 것들이었지만 이번에는 다르다.

펌핑 레마는 여러 언어 클래스들 각각에 대한 버전이 있으며, 이 보조정리는 주어진 언어가 그 언어 클래스에 속하지 않음을 증명할 때 주로 사용된다. 여기서는 당연히 정규언어regular language에 대한 버전을 다룰 것이다.

정규언어란 특정한 하나의 레겍스에 매치되는 모든 문자열들의 집합을 뜻한다. 펌핑 레마의 내용은 다음과 같다.

> L이 레겍스 re에 의해 규정되는 정규언어이면 어떤 자연수 N이 존재하여 길이가 N 이상인 모든 $s \in L$에 대해서 $s = s_1 + s_2 + s_3$이고 $s2 \neq []$이며 $\forall k \geqslant 0, s_1 + s_2^k + s_3 \in L$인 s_1, s_2, s_3가 존재한다.

이것은 weak version의 명제이며 strong version은 여기에 $s_1 + s_2$의 길이가 N 이하라는 조건이 추가된 것이다.

주어진 레겍스 re에 대한 문자열 길이의 한계 N을 계산하는 함수를 다음과 같이 정의하자.

```
1  Fixpoint pumping_constant {T} (re : reg_exp T) : nat :=
2    match re with
3    | EmptySet => 1
4    | EmptyStr => 1
5    | Char _ => 2
6    | App re1 re2 =>
7        pumping_constant re1 + pumping_constant re2
8    | Union re1 re2 =>
9        pumping_constant re1 + pumping_constant re2
10   | Star re => pumping_constant re
11   end.
```

펌핑 레마 weak_pumping의 증명에는 다음과 같은 보조정리들이 도움될 수 있다.

```
1  Lemma pumping_constant_ge_1 :
2    forall T (re : reg_exp T),
3      pumping_constant re >= 1.
4  Proof.
5    intros T re. induction re.
6    - (* EmptySet *)
7      simpl. apply le_n.
8    - (* EmptyStr *)
9      simpl. apply le_n.
```

```
10     - (* Char *)
11       simpl. apply le_S. apply le_n.
12     - (* App *)
13       simpl.
14       apply le_trans with (n:=pumping_constant re1).
15       { apply IHre1. } { apply le_plus_l. }
16     - (* Union *)
17       simpl.
18       apply le_trans with (n:=pumping_constant re1).
19       { apply IHre1. } { apply le_plus_l. }
20     - (* Star *)
21       simpl. apply IHre.
22   Qed.
```

라인 14에서 `le_trans with (n:= pumping_constant re1)`을 사용하기 직전의 화면은 다음과 같았다.

```
IHre1 : pumping_constant re1 >= 1
IHre2 : pumping_constant re2 >= 1
(1 / 1)
pumping_constant re1 + pumping_constant re2 >= 1
```

이 상태에서 라인 14를 실행하면 어떻게 될지를 마음 속에 쉽게 그릴 수 있어야 한다.

다음의 보조정리는 아주 쉽다.

```
1  Lemma pumping_constant_0_false :
2    forall T (re : reg_exp T),
3      pumping_constant re = 0 -> False.
4  Proof.
5    intros T re H.
6    assert (Hp1 : pumping_constant re >= 1).
7    { apply pumping_constant_ge_1. }
8    { rewrite H in Hp1. inversion Hp1. }
9  Qed.
```

문자열을 여러 번 반복하여 긴 문자열을 얻는 함수를 다음과 같이 정의한다.

```
1  Fixpoint napp {T} (n : nat) (l : list T) : list T :=
2    match n with
3    | 0 => []
4    | S n' => l ++ napp n' l
5    end.
```

다음은 간단한 보조정리들이다.

```
1  Lemma napp_star :
2    forall T m s1 s2 (re : reg_exp T),
3      s1 =~ re -> s2 =~ Star re ->
4      (napp m s1) ++ s2 =~ Star re.
5  Proof.
6    intros T m s1 s2 re Hs1 Hs2.
7    induction m.
8    - simpl. exact Hs2.
```

```
9     - simpl. rewrite <- app_assoc.
10      apply MStarApp.
11      + exact Hs1.
12      + exact IHm.
13  Qed.
14
15  Lemma le_0_is_0 : forall n, n <= 0 -> n = 0.
16  Proof.
17    intros n H. destruct n as [| n'].
18    - reflexivity.
19    - inversion H.
20  Qed.
21
22  Lemma app_assoc0 : forall T (s1 s2 s3 s4: list T),
23    s1 ++ (s2 ++ s3) ++ s4 = s1 ++ s2 ++ s3 ++ s4.
24  Proof.
25    intros T s1 s2 s3 s4.
26    rewrite <- app_assoc.
27    reflexivity.
28  Qed.
29
30  Lemma app_assoc1 : forall T (s1 s2 s3 s4: list T),
31    (s1 ++ s2 ++ s3) ++ s4 = s1 ++ s2 ++ s3 ++ s4.
32  Proof.
33    intros T s1 s2 s3 s4.
34    rewrite <- app_assoc.
35    apply app_assoc0.
36  Qed.
```

The Pumping Lemma, Weak version

이제 메인 정리이다. 통상 이 정리의 (Coq를 사용하지 않은) 증명은 주어진 정규언어(혹은 정규표현식)에 대응되는 Finite State Automaton의 state의 개수를 N으로 두고 진행한다.[11]

우리는 FSA를 사용하지 않으므로 상태들의 개수 대신 pumping_constant를 사용한다. 증명이 상당히 긴데 그 이유는 =~의 생성자들에 대한 경우를 일일이 하나씩 다루기 때문이다. 마지막 경우인 MStarApp 경우를 제외하고는 모두 평이하게 증명된다.

증명이 상당히 길므로 몇 개의 구역으로 나누어서 보겠다. 증명하고자 하는 weak pumping lemma의 명제를 WPL로 나타내기로 하자. 이 명제는 조금 복잡해 보일 수 있는데, 이것이 무엇을 의미하는지를 정확히 이해하고 있기만 하면, 이 증명은 길기만 했지 어떻게 작동하는지를 이해하기는 어렵지 않다.

```
1  Lemma weak_pumping : forall (T: Type) (re: reg_exp T) (s: list T),
2    s =~ re ->
3    pumping_constant re <= length s ->
4    exists s1 s2 s3,
5      s = s1 ++ s2 ++ s3 /\
```

[11] 길이가 N 이상인 문자열을 받아들이는 FSA가 이 문자열을 받아들이는 계산과정의 상태state들의 열 $\langle q_0, q_{1i}, q_{2i}, \ldots, q_{Ni} \rangle$ 중에는 중복된 상태가 존재한다. 그러므로 $q_{ji} = q_{ki}$ 인 $j < k$를 취했을 때 ... (skip).

```
6       s2 <> [] /\
7       forall (m: nat), s1 ++ napp m s2 ++ s3 =~ re.
8   Proof.
9     intros T re s Hm.
10    induction Hm as [
11            | x | s1 re1 s2 re2 Hm1 IH1 Hm2 IH2
12            | s1 re1 re2 Hm IH | re1 s2 re2 Hm IH
13            | re | s1 s2 re Hm1 IH1 Hm2 IH2 ].
14    - (* MEmpty *)
15      simpl. intros contra. inversion contra.
16    - (* MChar *)
17      simpl. intros contra. inversion contra. inversion H0.
```

MEmpty와 MChar 경우에는 length s가 각각 0과 1이고 pumping_constant re가 각각 1과 2이므로 라인 3의 조건이 성립하지 않아 간단히 처리된다.

이어서 MApp의 경우를 보겠다.

```
18    - (* MApp *)
19      simpl. intros H.
20      replace (length (s1 ++ s2)) with (length s1 + length s2) in H.
21      {
22        assert(Ha: pumping_constant re1 <= length s1 \/
23                   pumping_constant re2 <= length s2).
24        { apply add_le_cases. exact H. }
25        destruct Ha as [Ha_left | Ha_right].
26        { (* Ha_left : pumping_constant re1 <= length s1 *)
27          specialize (IH1 Ha_left).
28          destruct IH1 as [s2' [s3' [s4' [IH1a [IH1b IH1c]]]]].
29          exists s2', s3', (s4' ++ s2).
30          split.
31          - rewrite IH1a. apply app_assoc1.
32          - split.
33            + exact IH1b.
34            + intros m.
35              replace (s2' ++ napp m s3' ++ s4' ++ s2) with
36                      ((s2' ++ napp m s3' ++ s4') ++ s2).
37              { apply MApp. apply IH1c. exact Hm2. }
38              { apply app_assoc1. }
39        }
40        { (* Ha_right : pumping_constant re2 <= length s2 *)
41          (* ... *)
42        }
43      }
44      { rewrite app_length. reflexivity. }
```

라인 19를 실행한 후의 화면은 다음과 같다.

```
Hm1 : s1 =~ re1
Hm2 : s2 =~ re2
IH1 : pumping_constant re1 <= length s1 -> exists s2 s3 s4 : list T,
  s1 = s2 ++ s3 ++ s4 /\ s3 <> [ ] /\
  (forall m : nat, s2 ++ napp m s3 ++ s4 =~ re1)
IH2 : pumping_constant re2 <= length s2 -> exists s1 s3 s4 : list T,
```

```
    s2 = s1 ++ s3 ++ s4 /\ s3 <> [ ] /\
    (forall m : nat, s1 ++ napp m s3 ++ s4 =~ re2)
  H : pumping_constant re1 + pumping_constant re2 <= length (s1 ++ s2)
  (1 / 1) ------------------------
  exists s0 s3 s4 : list T,
    s1 ++ s2 = s0 ++ s3 ++ s4 /\ s3 <> [ ] /\
    (forall m : nat, s0 ++ napp m s3 ++ s4 =~ App re1 re2)
```

s1 =~ re1과 s2 =~ re2가 각각 WPL, 즉 IH1과 IH2를 만족할 때 s1 ++ s2 =~ App re1 re2가 WPL, 즉 H -> exists s0 s3 s4 ...을 만족함을 보여야 한다.

먼저 H로부터 라인 22의 assertion을 증명한다. 이 assertion을 가설로 사용할 때, 이것이 논리합문이므로 2가지 경우로 나누어 증명을 진행한다. 첫 번째 경우는 라인 26에서 시작하고 두 번째 경우는 라인 40에서 시작한다. 각 경우에 존재문의 witness로부터 고울 프랍의 witness를 만들어 내는 것이 증명의 핵심인데, 이것은 WPL이 무엇을 말하는지를 정확히 이해하고 있기만 하면 아주 쉽다. (힌트). 두 번째 경우에 라인 29에 해당하는 것은 exists (s1 ++ s1'), s3', s4'이다.

이어서 MUnionL과 MUnionR의 경우를 보겠다. 후자는 전자와 비슷하므로 전자만 설명하겠다.

```
45    - (* MUnionL *)
46      simpl. intros H.
47      assert (Ha: pumping_constant re1 <= length s1).
48      { apply le_trans
49          with (n:= pumping_constant re1 + pumping_constant re2).
50        - apply le_plus_l.
51        - exact H. }
52      { specialize (IH Ha).
53        destruct IH as [s2' [s3' [s4' [IHa [IHb IHc]]]]].
54        exists s2', s3', s4'.
55        split.
56        - exact IHa.
57        - split.
58          + exact IHb.
59          + intros m. apply MUnionL. apply IHc. }
60    - (* MUnionR *)
61      (* ... *)
```

라인 46을 실행한 후의 화면은 다음과 같다.

```
  Hm : s1 =~ re1
  IH : pumping_constant re1 <= length s1 -> exists s2 s3 s4 : list T,
    s1 = s2 ++ s3 ++ s4 /\ s3 <> [ ] /\
    (forall m : nat, s2 ++ napp m s3 ++ s4 =~ re1)
  H : pumping_constant re1 + pumping_constant re2 <= length s1
  (1 / 1) ------------------------
  exists s0 s2 s3 : list T,
    s1 = s0 ++ s2 ++ s3 /\ s2 <> [ ] /\
    (forall m : nat, s0 ++ napp m s2 ++ s3 =~ Union re1 re2)
```

MApp 경우와 어떻게 다른지 들여다 보기 바란다. 고울 프랍을 얻기 위하여 우리가 필요로 하는 것은 라인 47의 Ha인데 컨텍스트에는 이것보다 강한 H가 있으므로 아무 문제가 없다. 그 다음은 MApp와 비슷하다. Witness를 귀납가설로부터 찾아내는 것은 오히려 더 쉽다.

다음은 MStar0와 MStarApp 경우인데 전자는 쉬우므로 후자만 설명하겠다.

```
62    - (* MStar0 *)
63      intros H. simpl in H. exfalso. inversion H.
64      apply pumping_constant_0_false in H1. exact H1.
65    - (* MStarApp *)
66      simpl in *. intros H.
67      (* to be continued *)
```

라인 66을 실행한 후의 화면은 다음과 같다.

```
Hm1 : s1 =~ re
Hm2 : s2 =~ Star re
IH1 : pumping_constant re <= length s1 ->
  exists s2 s3 s4 : list T,
    s1 = s2 ++ s3 ++ s4 /\ s3 <> [ ] /\
    (forall m : nat, s2 ++ napp m s3 ++ s4 =~ re)
IH2 : pumping_constant re <= length s2 ->
  exists s1 s3 s4 : list T,
    s2 = s1 ++ s3 ++ s4 /\ s3 <> [ ] /\
    (forall m : nat, s1 ++ napp m s3 ++ s4 =~ Star re)
H : pumping_constant re <= length (s1 ++ s2)
(1 / 1) ------------------------
exists s0 s3 s4 : list T,
  s1 ++ s2 = s0 ++ s3 ++ s4 /\ s3 <> [ ] /\
  (forall m : nat, s0 ++ napp m s3 ++ s4 =~ Star re)
```

MStarApp가 다른 생성자와 다른 부분은 H: pumping_constant re <= length (s1 ++ s2)를 가지고 IH1과 IH2의 전건을 얻기가 쉽지 않다는 것이다.

이 문제는 s1의 길이가 pumping_constant re보다 작은 경우와 그렇지 않은 경우로 나눔으로써 해결할 수 있다. 후자의 경우는 IH1을 사용하면 되고, 전자의 경우에는 다시 s1 = []인 경우와 그렇지 않은 경우로 나누어야 한다. s1 = []일 때는 IH2를 사용하면 되고, 그렇지 않을 때는 (라인 85에서 보듯이) s1 자신을 고울 프랍에서의 펌핑 문자열 s3로 두면 된다.[12]

```
68  (* continued from line 67 *)
69      assert (Ha: length s1 < pumping_constant re \/
70                  length s1 >= pumping_constant re).
71      { apply lt_ge_cases. }
72      destruct Ha as [Ha_left | Ha_right].
73      + (* Ha_left : length s1 < pumping_constant re *)
74        destruct s1 as [| x s1tl].
75        { (* s1 = empty string *)
76          simpl in *. specialize (IH2 H).
77          destruct IH2 as [s1' [s3' [s4' [IH2a [IH2b IH2c]]]]].
78          exists s1', s3', s4'.
79          repeat split.
80          - exact IH2a.
81          - exact IH2b.
82          - exact IH2c.
83        }
84        { (* s1 = x :: s1tl *)
```

[12]실은 이것보다 조금 더 간단한 경우별 분석 방법이 있으나, 이 방법은 나중에 strong version을 증명할 때 확장하여 사용할 수 있다는 장점을 가진다.

```
85          exists [], (x :: s1tl), s2.
86          simpl. repeat split.
87          - unfold not. intros H'. discriminate H'.
88          - intros m. apply napp_star.
89            + exact Hm1.
90            + exact Hm2.
91        }
92        + (* Ha_right : length s1 >= pumping_constant re *)
93          unfold ge in Ha_right.
94          specialize (IH1 Ha_right).
95          destruct IH1 as [s1' [s3' [s4' [IH1a [IH1b IH1c]]]]].
96          exists s1', s3', (s4' ++ s2).
97          repeat split.
98          { rewrite IH1a. apply app_assoc1. }
99          { exact IH1b. }
100         { intros m. specialize IH1c with (m:=m).
101           replace (s1' ++ napp m s3' ++ s4' ++ s2) with
102                   ((s1' ++ napp m s3' ++ s4') ++ s2).
103           - apply MStarApp. exact IH1c. exact Hm2.
104           - apply app_assoc1.
105         }
106 Qed.
```

이상으로 펌핑레마의 weak version의 증명은 끝났다.

The Pumping Lemma, Strong version

Strong version은 weak version의 조건에 `length (s1 ++ s2) <= N`를 추가하여 얻는다. Strong Pumping Lemma의 명제를 *SPL*로 나타내기로 하자.

SPL의 증명은 이전 버전에서와 비슷한 부분이 많기는 하지만 `exp_match`의 생성자들 중에 다음 두 경우는 상당히 까다롭다.

① `MApp`에서 `s1 =~ re1`, `s2 =~ re2`가 주어졌을 때, 펌핑되는 문자열이 `s1`에 속할 때는 문제가 없으나 `s2`에 속할 때는 조금 까다로운 문제가 발생한다. 이를 해결하기 위하여 명제논리의 토톨로지

$$P \vee Q \leftrightarrow (P \vee (\neg P \wedge Q))$$

의 constructive version을 보조정리로 만들어 사용한다.

② `MStarApp`에서 `s1 =~ re`, `s2 =~ Star re`가 주어졌을 때, 펌핑되는 문자열이 `s1`에 속할 때는 문제가 없으나 `s2`에 속할 때는 조금 까다로운 문제가 발생한다. 이를 해결하기 위하여 보조정리 `MStarApp1`을 사용한다.

지금까지 설명한 것들을 코드로 옮기면 다음과 같다.

Lemmas for MApp case

```
1 Check or_commut. (* forall P Q : Prop, P \/ Q -> Q \/ P *)
2 Check lt_ge_cases. (* forall n m : nat, n < m \/ n >= m *)
```

```
3
4  Lemma le_nat_or_iff : forall n1 n2 m1 m2: nat,
5    (n1 <= n2 \/ m1 <= m2) <->
6      (n1 <= n2 \/ (n1 > n2 /\ m1 <= m2)).
7  Proof.
8    intros n1 n2 m1 m2. split.
9    - intros H.
10     destruct H as [H | H].
11     + left. exact H.
12     + assert (Ha: n1 <= n2 \/ n1 > n2).
13       { apply or_commut. apply lt_ge_cases. }
14       destruct Ha as [Ha_left | Ha_right].
15       { left. exact Ha_left. }
16       { right. split.
17         - exact Ha_right.
18         - exact H. }
19   - intros H.
20     destruct H as [H | H].
21     + left. exact H.
22     + destruct H as [H_left H_right].
23       right. exact H_right.
24 Qed.
25
26 Lemma plus_le_compat : forall n1 n2 m1 m2: nat,
27   n1 <= n2 -> m1 <= m2 -> n1 + m1 <= n2 + m2.
28 Proof.
29   intros n1 n2 m1 m2 H1 H2.
30   assert (Ha: n1 + m1 <= n2 + m1).
31   { apply plus_le_compat_r. exact H1. }
32   assert (Hb: n2 + m1 <= n2 + m2).
33   { apply plus_le_compat_l. exact H2. }
34   apply le_trans with (n:= n2 + m1).
35   - exact Ha.
36   - exact Hb.
37 Qed.
```

SPL의 증명은 다음 스크립트의 라인 1부터 시작하여 두 부분으로 나누어 제시한다. 첫 부분은 `Star re` 이전까지다. 이 증명 스크립트는 라인 26까지는 이전의 WPL과 똑같다. 라인 31의 `clear`는 처음 나오는 책략이다. 이 책략이 하는 일은 아주 간단하다 — 컨텍스트의 가설 중에 더 이상 필요가 없어진 것을 제거하는 데 쓰인다.

이후에도 대동소이하나 라인 54에서 새로 추가된 `Ha_right1`에 대해서는 주목할 필요가 있다. SPL에서 추가된 조건 `length s1 + length s2 <= pumping_constant App re1 re2`는 이 상황에서 `length s1 + (length s1' + length s3') <= pumping_constant re1 + pumping_constant re2`가 되며, 이 부등식은

① `length s1 <= pumping_constant re1`과

② `length s1' + length s3'<= pumping_constant re2`

로 나누어 증명한다. 이때 ②는 귀납가설로부터 쉽게 얻어지고, ①은

`Ha_right1 : pumping_constant re1 > length s1`

로부터 얻는다.

MUnionL은 이전 WPL 코드의 라인 45 ~ 58까지는 똑 같다. 이전 코드의 라인 59에 있는 intros m. apply MUnionL. apply IHc. }를 새 코드의 라인 85 ~ 88로 대체하면 된다.

MUnionR은 생략한다.

```
(* Strong Pumping Lemma, Part 1 of 2 *)
Lemma pumping : forall T (re : reg_exp T) s,
  s =~ re ->
  pumping_constant re <= length s ->
  exists s1 s2 s3,
    s = s1 ++ s2 ++ s3 /\
    s2 <> [] /\
    length s1 + length s2 <= pumping_constant re /\
    forall m, s1 ++ napp m s2 ++ s3 =~ re.
Proof.
  intros T re s Hm.
  induction Hm
    as [ | x | s1 re1 s2 re2 Hm1 IH1 Hm2 IH2
       | s1 re1 re2 Hm IH | re1 s2 re2 Hm IH
       | re | s1 s2 re Hm1 IH1 Hm2 IH2 ].
  - (* MEmpty *)
    simpl. intros contra. inversion contra.
  - (* MChar *)
    simpl. intros contra. inversion contra. inversion H0.
  - (* MApp *)
    simpl. intros H.
    replace (length (s1 ++ s2)) with (length s1 + length s2) in H.
    {
      assert(Ha: pumping_constant re1 <= length s1 \/
                 pumping_constant re2 <= length s2).
      { apply add_le_cases. exact H. }
      assert(Ha': pumping_constant re1 <= length s1 \/
                  (pumping_constant re1 > length s1 /\
                   pumping_constant re2 <= length s2)).
      { apply le_nat_or_iff. exact Ha. }
      clear Ha. (* remove unnecessary hypothesis *)
      destruct Ha' as [Ha_left | Ha_right].
      (* Ha_right case is the harder one. *)
      { (* Ha_left : pumping_constant re1 <= length s1 *)
        specialize (IH1 Ha_left).
        destruct IH1 as [s2' [s3' [s4' [IH1a [IH1b [IH1c IH1d]]]]]].
        exists s2', s3', (s4' ++ s2).
        split.
        - rewrite IH1a. apply app_assoc1.
        - split.
          + exact IH1b.
          + split.
            * apply le_trans with (n:= pumping_constant re1).
              { exact IH1c. }
              { apply le_plus_l. }
            * intros m.
              replace (s2' ++ napp m s3' ++ s4' ++ s2) with
```

```
48                      ((s2' ++ napp m s3' ++ s4') ++ s2).
49                { apply MApp. apply IH1d. exact Hm2. }
50                { apply app_assoc1. }
51        }
52        { (* Ha_right : pumping_constant re1 > length s1 and
53                      pumping_constant re2 <= length s2 *)
54          destruct Ha_right as [Ha_right1 Ha_right2].
55          specialize (IH2 Ha_right2).
56          destruct IH2 as [s1' [s3' [s4' [IH2a [IH2b [IH2c IH2d]]]]]].
57          exists (s1 ++ s1'), s3', s4'.
58          split.
59          - rewrite IH2a. apply app_assoc.
60          - split.
61            + exact IH2b.
62            + split.
63              * replace (length(s1 ++ s1')) with
64                  (length(s1) + length s1').
65                { rewrite <- add_assoc.
66                  unfold gt in Ha_right1.
67                  apply n_lt_m__n_le_m in Ha_right1.
68                  apply plus_le_compat.
69                  { exact Ha_right1. }
70                  { exact IH2c. }
71                }
72                { rewrite app_length. reflexivity. }
73              * intros m.
74                replace ((s1 ++ s1') ++ napp m s3' ++ s4') with
75                        (s1 ++ (s1' ++ napp m s3' ++ s4')).
76                { apply MApp. exact Hm1. apply IH2d. }
77                { rewrite app_assoc. reflexivity. }
78        }
79      }
80      {
81        rewrite app_length. reflexivity.
82      }
83    - (* MUnionL *)
84      (* ... *)
85          + destruct IHc as [IHc_l IHc_r]. split.
86            { apply le_plus_trans. exact IHc_l. }
87            { intros m. apply MUnionL. apply IHc_r. }
88      }
89    - (* MUnionR *)
90      (* to be continued *)
```

MStarApp case

MStar0와 MStarApp의 라인 76, simpl in *. specialize (IH2 H)까지는 이전의 weak version에서와 동일하다. 라인 76은 다음의 스크립트에서 라인 99에 대응된다. 이후에는 repeat split에서 경우의 수가 3에서 4로 늘어난 것을 제외하고는 특별한 것이 없다.

```
91  (* Strong Pumping Lemma, Part 2 of 2, continued from line 90 *)
92    - (* MStar0 *)
```

```
93      (* ... *)
94    - (* MStarApp *)
95      (* ... *)
96      + (* Ha_left : length s1 < pumping_constant re *)
97        destruct s1 as [| x s1tl].
98        { (* s1 = empty string*)
99          simpl in *. specialize (IH2 H).
100         destruct IH2 as [s1' [s3' [s4' [IH2a [IH2b [IH2c IH2d]]]]]].
101         exists s1', s3', s4'.
102         repeat split.
103         - exact IH2a.
104         - exact IH2b.
105         - exact IH2c.
106         - exact IH2d.
107       }
108       { (* s1 = x :: s1tl *)
109         exists [], (x :: s1tl), s2.
110         simpl. repeat split.
111         - unfold not. intros H'. discriminate H'.
112         - simpl in Ha_left. unfold lt in Ha_left.
113           apply le_trans with (n:= S (S (length s1tl))).
114           + apply n_le_Sn.
115           + exact Ha_left.
116         - intros m. apply napp_star.
117           + exact Hm1.
118           + exact Hm2.
119       }
120     + (* Ha_right : length s1 >= pumping_constant re *)
121       unfold ge in Ha_right.
122       specialize (IH1 Ha_right).
123       destruct IH1 as [s2' [s3' [s4' [IH1a [IH1b [IH1c IH1d]]]]]].
124       exists s2', s3', (s4' ++ s2).
125       repeat split.
126       { rewrite IH1a. apply app_assoc1. }
127       { exact IH1b. }
128       { exact IH1c. }
129       { intros m. specialize IH1d with (m:=m).
130         replace (s2' ++ napp m s3' ++ s4' ++ s2) with
131                 ((s2' ++ napp m s3' ++ s4') ++ s2).
132         - apply MStarApp.
133           + exact IH1d.
134           + exact Hm2.
135         - apply app_assoc1.
136       }
137 Qed.
```

8.5 Case Study: Improving Reflection

Reflection은 bool과 Prop을 넘나들며 증명하는 기법을 뜻한다. 다음의 예를 보라. 라인 12에서 eqb_eq를 사용하여 bool에서 Prop으로 이동하여 증명을 진행하였다. 이것은 일종의 reflection

이다.

```
1  Theorem filter_not_empty_In : forall (n: nat) (l: list nat),
2    filter (fun x => n =? x) l <> [] ->
3    In n l.
4  Proof.
5    intros n l. induction l as [|m l' IHl'].
6    - (* l = [] *)
7      simpl. intros H. unfold not in H. apply H. reflexivity.
8    - (* l = m :: l' *)
9      simpl. destruct (n =? m) eqn:H.
10     + (* n =? m = true *)
11       intros _.
12       rewrite eqb_eq in H. rewrite H.
13       left. reflexivity.
14     + (* n =? m = false *)
15       intros H'. right. apply IHl'. exact H'.
16 Qed.
```

참고로 라인 11에서 보듯이 intros _와 같이 intros의 인수를 _로 하면, 인수에 _ 대신 이름을 주었다면 컨텍스트로 올라왔어야 할 것(이 경우에는 가설)이 올라오지 않는다.

Reflection을 사용하는 증명은 별 내용이 없는 경우에도 스크립트가 좀 지루하게 길어지기 쉽다. 코드를 줄이는 방법을 소개하겠다.

```
1  Inductive reflect (P : Prop) : bool -> Prop :=
2    | ReflectT (H :   P) : reflect P true
3    | ReflectF (H : ~ P) : reflect P false.
4
5  Check ReflectT. (* forall P : Prop, P -> reflect P true *)
6  Check ReflectF. (* forall P : Prop, ~ P -> reflect P false *)
7
8  Theorem iff_reflect : forall P b, (P <-> b = true) -> reflect P b.
9  Proof.
10   intros P b H. destruct b eqn:Eq.
11   - apply ReflectT. rewrite H. reflexivity.
12   - apply ReflectF. unfold not. rewrite H.
13     intros H'. discriminate H'.
14 Qed.
```

라인 10을 실행하면 다음과 같은 결과가 나온다.

```
Goal 1
P : Prop
b : bool
Eq : b = true
H : P <-> true = true
(1 / 2) ------------------------
reflect P true

Goal 2
P : Prop
b : bool
Eq : b = false
```

```
H : P <-> false = true
(2 / 2) ________________________
reflect P false
```

다음은 iff_reflect의 역방향 정리이다.

```
Theorem reflect_iff : forall P b, reflect P b -> (P <-> b = true).
Proof.
  intros P b H. destruct H as [H | H].
  - split.
    + intros _. reflexivity.
    + intros _. exact H.
  - split.
    + intros H'. unfold not in H. apply H in H'. destruct H'.
    + intros H'. discriminate.
Qed.
```

이 증명에서 특이한 점은 destruct H이다. H가 논리곱이나 논리합이 아니라 reflect P b인 경우에도 이 책략을 쓸 수 있는데 이때 어떤 일이, 그리고 왜 벌어지는지를 정확히 알아야 한다.

일반적으로 destruct *term*, 혹은 destruct H를 실행하면 *term*, 혹은 H의 타입의 생성자의 개수만큼의 서브고울이 생긴다. H의 타입(즉 프랍)이 reflect P b일 경우에는 생성자가 2개이므로 2개의 서브고울이 생기게 되며, 이 프랍은 reflect P true, 또는 reflect P false이다. 그리고 각 프랍에 따라 그것의 원인이 되는 가설, 즉 P 또는 ~ P가 컨텍스트에 들어가게 된다.

팩트 8.1 iff_reflect와 reflect_iff에 의하여 P <-> b = true를 쓸 자리에는 항상 reflect P b를 대신 쓸 수 있고, 이것의 역방향도 가능하다. ⊣

eqb_eq의 다른 버전 eqbP를 보자.

```
Check eqb_eq.
(* forall n1 n2 : nat, (n1 =? n2) = true <-> n1 = n2 *)

Lemma eqbP : forall n m, reflect (n = m) (n =? m).
Proof.
  intros n m. apply iff_reflect.
  rewrite eqb_eq. reflexivity.
Qed.
```

이제 reflect를 이용하여 앞서 보았던 filter_not_empty_In를 (아주 조금) 더 간편하게 증명할 수 있다.

```
Theorem filter_not_empty_In' : forall n l,
  filter (fun x => n =? x) l <> [] ->
  In n l.
Proof.
intros n l. induction l as [|m l' IHl'].
- (* l = [] *)
  simpl. intros H. unfold not in H. apply H. reflexivity.
- (* l = m :: l' *)
  simpl. destruct (eqbP n m) as [H | H].
  + (* n = m *)
```

```
11      intros _.
12      rewrite H.
13      left. reflexivity.
14    + (* n <> m *)
15      intros H'. right. apply IHl'. apply H'.
16  Qed.
```

이전의 스크립트와 비교하면 커맨드 1개가 줄어들었음을 알 수 있다.

연습문제를 풀어 보자.

```
1   Fixpoint count n l :=
2     match l with
3     | [] => 0
4     | m :: l' => (if n =? m then 1 else 0) + count n l'
5     end.
6
7   Theorem eqbP_practice : forall n l,
8     count n l = 0 -> ~ (In n l).
9   Proof.
10    intros n l Hcount. induction l as [| m l' IHl'].
11    - (* l = [] *)
12      simpl. unfold not. intros H. exact H.
13    - (* l = m :: l' *)
14      simpl. unfold not. intros H.
15      destruct (eqbP n m) as [H' | H'].
16      + (* n = m *)
17        rewrite H' in Hcount.  simpl in Hcount.
18        rewrite eqb_refl in Hcount.
19        discriminate Hcount.
20      + (* n <> m *)
21        simpl in Hcount.
22        destruct H as [H | H].
23        * rewrite H in H'. unfold not in H'. apply H'. reflexivity.
24        * destruct (n =? m).
25          { simpl in Hcount. discriminate Hcount. }
26          { simpl in Hcount. apply IHl' in Hcount.
27            unfold not in Hcount. apply Hcount in H. destruct H. }
28  Qed.
```

8.6 More Exercises

① nostutter는 리스트에 연이어 동일한 원소가 나오는는지를 검사하는 다형 술어이다.

```
Inductive nostutter {X: Type} : list X -> Prop :=
  | nostutter_nil : nostutter []
  | nostutter_one : forall x, nostutter [x]
  | nostutter_cons : forall x1 x2 l,
      x1 <> x2 -> nostutter (x2 :: l) -> nostutter (x1 :: x2 :: l).
```

유닛 테스트는 역시 재미없다.

```
Example test_nostutter_1: nostutter [3;1;4;1;5;6].
Proof.
  apply nostutter_cons.
  - unfold not. intros H. inversion H.
  - apply nostutter_cons.
    + unfold not. intros H. inversion H.
    + apply nostutter_cons.
      * unfold not. intros H. inversion H.
      * apply nostutter_cons.
        { unfold not. intros H. inversion H. }
        { apply nostutter_cons.
          - unfold not. intros H. inversion H.
          - apply nostutter_one. }
Qed.
```

이것을 좀 더 간단하게 증명할 수 있는 방법이 있다. 아직 배우지 않은 책략들을 사용해야 한다.

```
1  Example test_nostutter_1' : nostutter [3;1;4;1;5;6].
2  Proof. repeat constructor; apply eqb_neq; auto. Qed.
3
4  Example test_nostutter_2 :  nostutter (@nil nat).
5  Proof. apply nostutter_nil. Qed.
6
7  Example test_nostutter_3 :  nostutter [5].
8  Proof. apply nostutter_one. Qed.
9
10 Example test_nostutter_4 :      not (nostutter [3;1;1;4]).
11 Proof.
12   intro.
13   repeat match goal with
14     h: nostutter _ |- _ => inversion h; clear h; subst
15   end.
16   contradiction; auto.
17 Qed.
```

② 주어진 리스트가 두 리스트를 리스트 내에서의 순서를 유지하며 합쳐서 얻을수 있는지를 의미하는 다형 술어 merge를 만들어 보자.

```
1  Inductive merge {X : Type} : list X -> list X -> list X -> Prop :=
2    | merge_nil : merge [] [] []
3    | merge_left (x: X) (l1 l2 l3: list X) (H: merge l1 l2 l3):
4        merge (x :: l1) l2 (x :: l3)
5    | merge_right (x: X) (l1 l2 l3: list X) (H: merge l1 l2 l3):
6        merge l1 (x :: l2) (x :: l3).
7
8  Example test_merge_1 : merge [1;6;2] [4;3] [1;4;6;2;3].
9  Proof.
10   apply merge_left. apply merge_right. apply merge_left.
11   apply merge_left. apply merge_right. apply merge_nil.
12 Qed.
```

다음 문제에는 Logic.v에서 정의했던 All이라는 술어를 사용하는데 이게 이상하게도 제대로

작동하지 않는다. 그래서 All을 대신하여 사용할 All'을 정의하였다. 이름만 다르고 원래의 정의와 똑같다. 그리고 기본적인 보조정리 2개를 증명했다.

```
Fixpoint All' T : Type (P : T -> Prop) (l : list T) : Prop :=
  match l with
  | [] => True
  | x :: l' => P x /\ All' P l'
  end.

Fact All_hd : forall (X: Type) (P: X -> Prop) (x: X) (l: list X),
  All' P (x :: l) -> P x.
Proof.
  intros X P x l H.
  simpl in H. destruct H as [H1 H2]. exact H1.
Qed.

Fact All_tl : forall (X: Type) (P: X -> Prop) (x: X) (l: list X),
  All' P (x :: l) -> All' P l.
Proof.
  intros X P x l H.
  simpl in H. destruct H as [H1 H2]. exact H2.
Qed.
```

discrimnate를 2군데에서 사용했다. 고울프랍을 증명할 수 없을 때는 컨텍스트에 모순이 있는지를 살펴야 하는 것을 잊지 말아야 한다.

```
Theorem merge_filter : forall (X: Set) (test: X -> bool) (l l1 l2 : list X),
  merge l1 l2 l ->
  All' (fun n => test n = true) l1 ->
  All' (fun n => test n = false) l2 ->
  filter test l = l1.
Proof.
  intros X test l l1 l2 Hm.
  induction Hm as [ | x l1' l2' l' Hm' IHm'
                  | x l1' l2' l' Hm' IHm'].
  - (* merge_nil *)
    intros H1 H2. simpl. reflexivity.
  - (* merge_left *)
    intros H1 H2. simpl.
    destruct (test x) eqn:Hx.
    + f_equal. apply IHm'.
      { apply All_tl with (x := x). exact H1. }
      { exact H2. }
    + apply All_hd in H1. rewrite H1 in Hx. discriminate Hx.
  - (* merge_right *)
    intros H1 H2. simpl.
    destruct (test x) eqn:Hx.
    + apply All_hd in H2. rewrite H2 in Hx. discriminate Hx.
    + apply IHm'.
      { exact H1. }
      { apply All_tl with (x := x). exact H2. }
Qed.
```

③ Among all subsequences of `l` with the property that test evaluates to true on all their members, `filter test l` is the longest. Formalize this claim and prove it.

버전 2개를 준비했다. 'longest'에서 순서관계를 부분수열로 정의한 버전과 길이의 대소로 정의한 버전이다. 먼저 부분수열로 정의한 버전을 보자.

```
Example filter_challenge_2' : forall (test: nat -> bool) (l l1: list nat),
  (subseq l1 l) -> (forallb test l1) = true -> subseq l1 (filter test l).
Proof.
  intros test l l1 Hsub Htest.
  induction Hsub as [l | l1' l' x Hsub' IHsub' | l1' l' x Hsub' IHsub'].
  - (* sbsq1 *)
    apply sbsq1.
  - (* sbsq2 *)
    simpl. destruct (test x) eqn:Hx.
    + apply sbsq2. apply IHsub'. exact Htest.
    + apply IHsub'. exact Htest.
  - (* sbsq3 *)
    simpl. destruct (test x) eqn:Hx.
    + apply sbsq3. apply IHsub'.
      simpl in Htest. rewrite Hx in Htest. simpl in Htest.
      exact Htest.
    + simpl in Htest. rewrite Hx in Htest. simpl in Htest.
      discriminate Htest.
Qed.
```

그 다음은 길이의 대소로 정의한 버전이다. 이 버전은 `subseq_length`라는 보조정리를 사용한다.

```
Example subseq_length : forall (l1 l2: list nat),
  subseq l1 l2 -> length l1 <= length l2.
Proof.
  intros l1 l2 Hsub. induction Hsub
    as [l | l1' l2' x Hsub' IHsub' | l1' l2' x Hsub' IHsub'].
  - (* sbsq1 *) apply le_0_n.
  - (* sbsq2 *) simpl. apply le_S. exact IHsub'.
  - (* sbsq3 *) simpl.  apply n_le_m__Sn_le_Sm. exact IHsub'.
Qed.

Example filter_challenge_2 : forall (test: nat -> bool) (l l1: list nat),
  (subseq l1 l) -> (forallb test l1) = true ->
  length l1 <= length (filter test l).
Proof.
  intros. apply subseq_length.
  apply filter_challenge_2'.
  { exact H. } { exact H0. }
Qed.
```

④ Palindrome은 앞으로 읽으나 뒤로 읽으나 같은 단어를 말한다. 이것을 Coq에서 다음과 같이 구성적으로 정의한다.

```
Inductive pal {X: Type} : list X -> Prop :=
  | pal_nil : pal []
  | pal_one : forall x, pal [x]
  | pal_cons : forall x l, pal l -> pal (x :: l ++ [x]).
```

우선 쉬운 문제 2개부터 풀어보자.

```
Theorem pal_app_rev : forall (X: Type) (l: list X),
  pal (l ++ (rev l)).
Proof.
  intros X l. induction l as [| x l' IHl'].
  - (* l = [ ] *) simpl. apply pal_nil.
  - (* l = x :: l' *)
    simpl. rewrite app_assoc. apply pal_cons. exact IHl'.
Qed.

Theorem pal_rev : forall (X: Type) (l: list X),
  pal l -> l = rev l.
Proof.
  intros X l H. induction H as [| x | x l Hl IHl].
  - (* pal_nil *) simpl. reflexivity.
  - (* pal_one *) simpl. reflexivity.
  - (* pal_cons *) simpl. rewrite rev_app_distr.
    simpl. rewrite <- IHl. reflexivity.
Qed.
```

pal_rev는 palindrome에 대한 우리의 정의가 원래의 정의를 함의한다는 것을 보여준다. 이제 이것의 역을 증명할 것인데, 이 방향은 더 어렵다. 2개의 보조정리를 먼저 증명하겠다.

```
Lemma pal_lemm1 : forall (X: Type) (l1 l2: list X) (a b: X),
  a :: l1 = l2 ++ [b] ->
  (a = b /\ l1 = []) \/ exists l3, l1 = l3 ++ [b].
Proof.
  induction l2 as [| x l2' IHl2'].
  - (* l2 = [] *)
    intros a b H. inversion H.
    left. split.
    { reflexivity. } { reflexivity. }
  - (* l2 = x :: l2' *)
    intros a b H. inversion H.
    right. exists l2'. reflexivity.
Qed.

Lemma pal_lemm2 : forall (X: Type) (l1 l2 : list X) (a b: X),
  l1 ++ [a] = l2 ++ [b] -> a = b / l1 = l2.
Proof.
  intros X l1 l2 a b H.
  assert (Ha: rev (l1 ++ [a]) = rev (l2 ++ [b])).
  { rewrite H. reflexivity. }
  rewrite rev_app_distr in Ha. rewrite rev_app_distr in Ha.
  simpl in Ha. injection Ha as Ha1 Ha2.
  split.
  - rewrite <- Ha1. reflexivity.
  - assert (Ha: rev (rev l1) = rev (rev l2)).
    { rewrite Ha2. reflexivity. }
    repeat rewrite rev_involutive in Ha. exact Ha.
Qed.
```

메인 정리 palindrome_converse는 다음과 같다. 이 증명에서 중요한 것은 리스트에 대한 귀납으로는 증명이 잘 안 된다는 것이다. 그래서 리스트의 길이에 대한 귀납을 사용하여 보조정리 palindrome_converse_length를 먼저 증명한 다음에 본 정리를 증명하였다.

```
1  Theorem palindrome_converse : forall X: Type (l: list X),
2    l = rev l -> pal l.
3  Proof.
4    intros X l H. apply palindrome_converse_length with (n:= length l).
5    { apply le_n. } { exact H. }
6  Qed.
7
8  Lemma palindrome_converse_length: forall (X: Type) (n: nat) (l: list X),
9    length l <= n -> l = rev l -> pal l.
10 Proof.
11   intros X n.
12   induction n as [| n' IHn'].
13   - (* n = 0 *) destruct l as [| x l'] eqn:Eq.
14     + intros. apply pal_nil.
15     + intros Hlen H. simpl in Hlen. inversion Hlen.
16   - (* n = S n' *) destruct l as [| x l'] eqn:Eq.
17     + (* l = [ ] *) intros. apply pal_nil.
18     + (* l = x :: l' *) intros Hlen Hrev.
19       simpl in Hrev.
20       (* Hrev : x :: l' = rev l' ++ [x] *)
21       pose proof Hrev as Hrev_orig.
22       apply pal_lemm1 in Hrev.
23       destruct Hrev as [Hrev_l | Hrev_r].
24       { destruct Hrev_l as [Hrev_l1 Hrev_l2].
25         clear Hrev_l1. rewrite Hrev_l2. apply pal_one. }
26       { destruct Hrev_r as [l3 Hrev_r].
27         rewrite Hrev_r. apply pal_cons.
28         apply IHn' with (l := l3).
29         - simpl in Hlen. apply le_S_n in Hlen.
30           apply le_trans with (n:= length l').
31           + rewrite Hrev_r.
32             replace (length (l3 ++ [x])) with (S (length l3)).
33             { apply le_S. apply le_n. }
34             { rewrite app_length. simpl.
35               rewrite add_comm. simpl. reflexivity. }
36           + exact Hlen.
37         - rewrite Hrev_r in Hrev_orig.
38           replace (rev (l3 ++ [x])) with (x :: rev l3) in Hrev_orig.
39           { apply pal_lemm2 with (l1:= x :: l3) in Hrev_orig.
40             destruct Hrev_orig as [_ Hrev_orig].
41             injection Hrev_orig as Hrev_orig.
42             exact Hrev_orig. }
43           { rewrite rev_app_distr. simpl. reflexivity. }
44       }
45 Qed.
```

라인 20의 Hrev는 나중에 라인 37 이하에서 다시 써야 하는 중요한 정보를 담고 있다. 그런데 라인 22의 apply pal_lemm1 in Hrev에 의해서 이 정보를 잃게 된다. 그래서 라인 21에서 Hrev를

Hrev_orig라는 이름으로 저장해 둔 것이다. 이때 사용하는 명령이 pose proof .. as .. 이다.

⑤ 두 리스트에 공통 원소가 없음을 나타내는 술어 disjoint를 귀납을 써서 정의하고, 이것과 NoDup과 ++에 대한 관계를 나타내는 정리를 증명하라.

```
1  Inductive disjoint {X: Type} : list X -> list X -> Prop :=
2    | disjoint_nil : disjoint [] []
3    | disjoint_cons1 : forall x l1 l2,
4        ~ In x l2 -> disjoint l1 l2 -> disjoint (x :: l1) l2
5    | disjoint_cons2 : forall x l1 l2,
6        ~ In x l1 -> disjoint l1 l2 -> disjoint l1 (x :: l2).
7
8  Inductive NoDup X: Type : list X -> Prop :=
9    | NoDup_nil : NoDup []
10   | NoDup_cons : forall x l,
11       ~ In x l -> NoDup l -> NoDup (x :: l).
12
13 Theorem NoDup_disjoint : forall (X: Type) (l1 l2: list X),
14   NoDup (l1 ++ l2) -> disjoint l1 l2.
15 Proof.
16   intros X l1 l2 H. induction l1 as [| x l1' IHl1'].
17   - (* l1 = [] *)
18     induction l2.
19     - apply disjoint_nil.
20     - simpl in *. apply disjoint_cons2.
21       + unfold not. intros H'. inversion H'.
22       + apply IHl2. inversion H. exact H3.
23   - (* l1 = x :: l1' *)
24     inversion H as [| x' l' H1 H2 H3].
25     apply disjoint_cons1.
26     + unfold not. intros H'. apply H1.
27       apply In_app_iff. right. exact H'.
28     + apply IHl1'. exact H2.
29 Qed.
```

⑤ 다음 보조정리는 비둘기집 원리의 증명에 사용된다.

```
Lemma in_split : forall (X: Type) (x: X) (l: list X),
  In x l ->
  exists l1 l2, l = l1 ++ x :: l2.
Proof.
intros X x l H. induction l as [| x' l' IHl'].
- (* l = [] *) inversion H.
- (* l = x' :: l' *) simpl in H.
  destruct H as [H | H].
  + (* x = x' *) exists [], l'. simpl. rewrite H. reflexivity.
  + (* In x l' *) apply IHl' in H. destruct H as [l1 [l2 H']].
    exists (x' :: l1), l2. simpl. rewrite H'. reflexivity.
Qed.
```

비둘기집 원리를 나타내기 위하여 리스트 내에 중복된 원소가 있음을 말하는 술어를 다음과 같이 정의한다.

```
Inductive repeats {X: Type} : list X -> Prop :=
| repeats_In : forall x l, In x l -> repeats (x :: l)
| repeats_tl : forall x l, repeats l -> repeats (x :: l).
```

이제 비둘기집 원리를 증명하자. 이 증명에는 배중률을 사용한다. (See line 21.)

```
1  From Coq Require Import Classical_Prop.
2
3  Theorem pigeonhole_principle: forall (X: Type) (l1 l2: list X),
4    (forall x, In x l1 -> In x l2) ->
5    length l2 < length l1 ->
6    repeats l1.
7  Proof.
8    intros X l1. induction l1 as [|x l1' IHl1'].
9    - (* l1 = [] *) intros l2 H1 H2. inversion H2.
10   - (* l1 = x :: l1' *) intros l2 H1 H2.
11     pose proof H1 as H1_orig.
12     specialize (H1 x). simpl in H1.
13     assert (Ha: x = x \/ In x l1').
14     { left. reflexivity. }
15     apply H1 in Ha. clear H1.
16     assert (Ha2: exists la lb, l2 = la ++ x :: lb).
17     { apply in_split. exact Ha. }
18     destruct Ha2 as [la [lb Ha2]].
19     remember (la ++ lb) as l2' eqn:Eq2.
20     specialize (IHl1' l2').
21     destruct (classic (In x l1')) as [H3 | H3].
22     + (* In x l1' *) apply repeats_In. exact H3.
23     + (* ~ In x l1' *) apply repeats_tl. apply IHl1'.
24     { intros x' H4.
25       specialize (H1_orig x'). simpl in H1_orig. unfold not in H3.
26       assert (Haxx' : x = x' -> False).
27       { intros Heq. rewrite Heq in H3. apply H3. exact H4. }
28       assert (Hax'l2 : In x' l2 ).
29       { apply H1_orig. right. exact H4. }
30       rewrite Ha2 in Hax'l2.
31       apply In_app_iff in Hax'l2. simpl in Hax'l2.
32       assert (H_goal1: In x' la \/ In x' lb).
33       { destruct Hax'l2 as [Hax1 | [ Hax2 | Hax3]].
34         - left. exact Hax1.
35         - exfalso. apply Haxx'. exact Hax2.
36         - right. exact Hax3. }
37       apply In_app_iff in H_goal1. rewrite <- Eq2 in H_goal1.
38       exact H_goal1.
39     }
40     { rewrite Eq2.
41       assert (Hlen2 : length l2 = length la + S (length lb) ).
42       { rewrite Ha2. rewrite app_length. simpl. reflexivity. }
43       rewrite <- plus_n_Sm in Hlen2.
44       rewrite <- app_length in Hlen2.
45       assert (Hlen3 : length (x :: l1') = S (length l1')).
46       { simpl. reflexivity. }
47       rewrite Hlen3 in H2. rewrite Hlen2 in H2.
```

```
48        unfold lt in *. apply le_S_n in H2. exact H2.
49      }
50  Qed.
```

9

The Curry-Howard Correspondence

다음의 세 장은 원래 SF에서는 recommended chapter for a semester course에 포함되어 있지 않지만 나는 이 장들을 이 책의 핵심적인 부분이라고 생각한다. 또한 SF에서는 이전의 Chapter 8. Inductively Defined Propositions의 바로 뒤에 Total and Partial Maps가 나오지만 나는 이것을 뒤로 미루고 이 3개 장들을 먼저 다루고자 한다.

9.1 Proof Scripts and Proof Terms

Evidence for Atomic Propositions

Proposition을 흔히 명제라고 번역하는데 이 우리는 이것을 프랍prop이라고 부른다. 그리고 술어predicate는 1개 이상의 인수와 결합하여 프랍이 되는 표현을 뜻하는 것으로 정의하자. SF에서는 이것을 프로퍼티property라고 하는데 이 책에서는 이 용어를 사용하지 않을 것이다.

술어에는 기본 술어atomic predicate와 복합 술어compound predicate가 있다. 후자는 전자(들)를 이들을 논리 결합자와 한정사로 엮어서 만든 것이다. 기본 술어의 예를 하나 보자. 이 술어의 이름은 ev이고 의도된 의미는 '짝수임'이다. 이 술어는 앞 장의 p143에서 정의하여 심도있게 다루었다.

```
1 Inductive ev : nat -> Prop :=
2   | ev_0                       : ev 0
3   | ev_SS (n : nat) (H : ev n) : ev (S (S n)).
```

Coq의 모든 기본 술어는 이렇게 Inductive를 사용하여 정의된다. 심지어 등호도 Coq의 표준 라이브러리에 다음과 같이 정의되어 있다.

```
4 Inductive eq {A: Type} (x : A) : A -> Prop :=
5   eq_refl : eq x x.
```

우리는 eq x y를 등식 x = y로 표기한다.

eq는 다형 2항 술어인데, 인수 중 하나는 라인 4의 3번째 콜론의 왼쪽에 x : A로 드러나explicit 있고, 다른 하나는 콜론의 오른쪽에 A에 숨겨져implicit 있다. Inductive 정의에서 인수는 첫 라인(라인 1, 4 등)의 콜론의 왼쪽/오른쪽, 그리고 생성자 라인(라인 2, 3, 5 등)의 콜론의 왼쪽/오른쪽 어디에 나타나는지에 따라 미묘한 차이가 있음에 유의하라.

Inductive라는 키워드는 생성자를 이용하여 타입을 정의하는데 사용된다. 생성자는 타입의 원소inhabitant를 만들어 내는 함수(혹은 상수)이다. 그러므로 기본 프랍은 타입이다. 생성자가 만들어 내는 '타입의 원소'를 에비던스evidence라고 부른다.

$x : A$는 통상 "x has type A"로 읽고, $x \in A$로 해석한다. 그런데 이것을 x는 A의 증거라고 읽는 것이 A가 프랍일 때는 더 적절하다.

`ev_0`는 프랍 `ev 0`의 증거이다. 그리고 `ev_SS n H`는 `ev (S (S n))`의 증거이다. 단, 이때 `n: nat`이고, `H: ev n`이어야 한다. 0이 짝수라는 것은 나타내는 명제(즉, 프랍) `ev 0`는 전통적 수학에서는 공리로 본다.[1] 타입이론에서는 `ev 0`가 성립함을 믿고 이것의 증거를 `ev_0`라는 이름을 주어 사용한다. 0 아닌 일반적인 자연수에 대한 '짝수성'에 대한 귀납적 공리는 $\forall n,\ \mathsf{ev}\, n \to \mathsf{ev}\,(n+2)$이다. 이 공리의 증거의 이름으로 `ev_SS`를 사용한다.

'같음성'의 공리는 $\forall x,\ x = x$이며 이 공리의 이름은 `eq_refl`이다.

에비던스, 혹은 증거항은 기본 프랍을 증명함을 보았다. 복합 프랍의 증명은 에비던스들을 다루는 프로그램으로 볼 수 있다. 혹은, 이러한 증명은 에비던스들로 이루어진 표현항, 혹은 에비던스들을 잎사귀로 가지는 트리라고 생각해도 된다. 이렇게 증명을 구문론적 표현항으로 본 것을 증거항proof term, or proof object이라고 부른다. 한편, 증명을 프로그램으로 본 것을 증명 스크립트(*proof script*)라고 부른다. 복합 프랍의 증거항을 만드는 방법은 앞으로 결합자와 한정사 각각에 대해서 하나씩 설명할 것이다.

프랍은 타입이며 프랍의 증명이 이 타입의 원소라는 것이 Curry-Howard Correspondence의 가장 기본적 레벨에서의 의미이다. 조금 더 넓은 의미로, 그리고 추상적으로 말하자면 커리-하워드 대응은 증명이론과 타입이론의 다양한 대응을 의미한다. 예를 들어 minimal propositional logic은[2] simply typed lambda calculus와 대응되고, 1계논리는 dependent type과 대응되며, 2계논리는 polymorphic type과 대응된다.

Evidence Constructors and Proof Scripts

앞서 `ev_SS n H`에 대해서 말했는데, 그렇다면 `ev_SS`는 무엇인가? 먼저 `ev_SS`의 타입을 알아보자.

```
Check ev_SS
  : forall n, ev n -> ev (S (S n)).
```

의존타입을 가짐을 알 수 있다.[3] 이 점은 `ev`와 같다. 다만 `ev_SS`의 타입은 조금 더 복잡한 형태이다. 이제 `ev 4`에 대한 증명을 세밀하게 들여다 보자.

```
1 Theorem ev_4 : ev 4.
2 Proof.
3   apply ev_SS. apply ev_SS. apply ev_0.
4 Qed.
5
6 Theorem ev_4' : ev 4.
```

[1] 혹은 '짝수성'의 정의로 볼 수도 있다.

[2] Minimal propositional logic은 배중률을 받아들이지 않는다는 점에서 직관주의 논리와 같다. Minimal logic은 여기에 더하여 ex falso(혹은 falsum, 즉 모순은 임의의 명제를 함의한다는 규칙)도 받아들이지 않는다는 점에서 이를 허용하는 직관주의 논리와 구별된다.

[3] 의존타입의 정의는 p209에 나온다.

```
7  Proof.
8    apply (ev_SS 2 (ev_SS 0 ev_0)).
9  Qed.
10
11 Print ev_4'. (* = ev_SS 2 (ev_SS 0 ev_0) : ev 4  *)
12 Print ev_4. (* same as above *)
13
14 Check (ev_SS 2 (ev_SS 0 ev_0)). (** ev 4 *)
```

정리의 증거항을 보려면 증명스크립트 내에서 Show Proof 명령을 사용하면 되는데, 라인 11, 12에서와 같이 증명 스크립트 외부에서 Print my_theorem 명령을 사용해도 된다. 라인 11은 당연한 결과인데 라인 12는 기대하지 않았던 (좋은) 결과이다. 정리의 프랍을 보려면 Check my_theorem 명령을 사용하면 됨은 잘 알고 있을 것이다.

정리의 Proof부터 Qed 사이의 부분이 증명 스크립트$_{\text{proof script}}$이다. 위에 보인 코드에서 라인 3이 증명 스크립트의 예이다.

증거항은 증명 스크립트가 만들어 내는 구문론적 존재이며 라인 8에서 apply의 인수로 사용된 표현항 ev_SS 2 (ev_SS 0 ev_0)이 하나의 예이다. 라인 12를 보면 우리가 작성한 증명 스크립트로부터 Coq이 증거항을 만들어 내었음을 알 수 있다.

증명 스크립트가 증거항을 만들어 내는 과정은 Show Proof 명령을 통하여 생생히 볼 수 있다. 만일 과정은 관심없고 최종으로 얻어진 증거항에만 관심이 있다면 앞서 설명했던 것처럼 Print my_theorem을 사용하면 된다.

```
Theorem ev_4''  ev 4.
Proof.
  Show Proof. (* ?Goal *)
  apply ev_SS.
  Show Proof. (* ev_SS 2 ?Goal *)
  apply ev_SS.
  Show Proof. (* ev_SS 2 (ev_SS 0 ?Goal) *)
  apply ev_0.
  Show Proof. (* ev_SS 2 (ev_SS 0 ev_0) *)
Qed.
```

지금까지 프랍 ev 4의 증명 3개를 보았는데 하나 더 추가해 보자.

```
Definition ev_4''' : ev 4 :=
  ev_SS 2 (ev_SS 0 ev_0).
```

이러한 Definition에서 증거항에 오류가 있는 경우에는 Coq이 받아들이지 않는다.

다시 강조하겠는데, 이 4개의 증명에 대해서 Print ev_4, Print ev_4' 등의 명령을 내리면 결과는 모두 같다.

Quantifiers, Implications, Functions

타입에 화살표가 등장하는 것에는 두 경우가 있다. 하나는 함수이고 다른 하나는 Inductive 정의에서 나타나는 생성자이다.

함수 $n \mapsto 2n + 1$을 다음과 같이 3개의 서로 다른 구문으로 정의할 수 있다.

```
Definition my_f (n: nat) : nat :=
  n * 2 + 1.
Check my_f. (* my_f : nat -> nat *)
Compute my_f 2. (* = 5 : nat*)

Definition my_f' : nat -> nat :=
  fun n => n * 2 + 1.
Check my_f'. (* my_f' : nat -> nat *)
Compute my_f' 2. (* = 5 : nat *)

Definition my_f'' : forall (_: nat), nat :=
  fun n => n * 2 + 1.
Check my_f''. (* my_f'' L nat -> nat *)
Compute my_f'' 2.  (* = 5 : nat *)
```

이 세 함수들이 모두 동일한 함수임은 다음과 같이 증명할 수 있다. (3개의 fact 중 하나만 증명하겠다. 다른 둘은 똑 같은 방법으로 증명된다.)

```
Fact same_fn : my_f = my_f'.
Proof.
  apply functional_extensionality.
  intros x.
  unfold my_f. unfold my_f'.
  reflexivity.
Qed.
```

이 증명에서 라인 17 ~ 19는 생략해도 된다. 라인 20의 `reflexivity`가 다른 작업들을 다 포함해서 해 주기 때문이다.

라인 11에서 정의하고자 하는 대상의 타입을 `forall (_: nat), nat`로 나타낸 것이 흥미롭다. 화살표 대신 전칭한정사를 사용하였다. 그리고 한정사의 묶인 변수를 _로 표시하였다. 이것은 묶인 변수로, n, m, x 등 어느 것을 사용하는 것도 허용되지만, _를 사용함으로써 `my_f''`의 리턴 값의 타입이 인수의 값에 상관이 없음을 강조한 것이다. 이런 의미에서 화살표는 전칭한정의 축퇴degenerate 경우라고 볼 수 있다. 라인 8과 라인 11을 비교해 보라.

이번에는 `Inductive` 정의에서 화살표가 등장하는 경우를 살펴보자.

```
Inductive my_type : Type :=
  | my_constr (_ : nat): my_type.
Check my_constr. (* nat -> my_type *)
Check my_constr 2. (* my_constr 2 : my_type *)
Compute my_constr 2. (* = my_constr 2 : my_type *)

Inductive my_type' : Type :=
  | my_constr' : forall (_ : nat), my_type'.
Check my_constr'. (* nat -> my_type' *)
Check my_constr' 2. (* my_constr' 2 : my_type' *)
Compute my_constr' 2. (* = my_constr' 2 : my_type' *)
```

화살표는 전칭한정의 축퇴임을 여기서도 볼 수 있다. 라인 8과 라인 9를 비교해 보라.

축퇴가 아닌 전칭한정의 예는 `ev_SS`에서 본 바 있다.

```
Check ev_SS : forall n, ev n -> ev (S (S n)).
```

n에 따라 타입이 달라지는 것을 볼 수 있다. 이런 것을 의존 타입(*dependent type*)이라고 한다. 의존 타입은 다형 타입과 비슷한 부분이 있다. 타입의 정의에 인수를 주어 다양한 타입을 얻을 수 있다는 점에서 같다. 다른 점은 다형 타입은 인수로 타입을 받아들이는 반면에 의존 타입은 인수로 어떤 타입의 값value을 받아들인다는 것이다.

이 장에서 다루었던 프랍, 예를 들어 ev 4에는 화살표가 나타나지 않았다. 이제 화살표가 등장하는 프랍의 예로 forall (n: nat), ev n -> ev (4 + n)에 대한 에비던스를 알아보자. 이러한 복합 프랍에 대한 에비던스를 나타낼 때는 함수를 사용해야 함을 보게 될 것이다.

```
1  Theorem ev_plus4 : forall n, ev n -> ev (4 + n).
2  Proof.
3    intros n H.
4    apply ev_SS.
5    apply ev_SS.
6    apply H.
7  Qed.
8
9  Print ev_plus4.  (* (fun (n : nat) (H : ev n) => ev_SS (S (S n)) (ev_SS n H)) *)
```

이 프랍의 에비던스는 라인 9에서 보듯이 함수이다. 정리 ev_plus4에서 알아낸 증거항을 이용하여 이 정리를 Definition을 써서 다음과 같이 나타낼 수 있다.

```
10 Definition ev_plus4' : forall n, ev n -> ev (4 + n) :=
11   fun (n : nat) (H : ev n) => ev_SS (S (S n)) (ev_SS n H).
```

에비던스가 함수이므로 함수 구문을 사용하여 다음과 같이 정리를 나타낼 수도 있다.

```
12 Definition ev_plus4'' (n : nat) (H : ev n) : ev (4 + n) :=
13   ev_SS (S (S n)) (ev_SS n H).
14
15 Check ev_plus4'. (* : forall n : nat, ev n -> ev (4 + n) *)
16 Check ev_plus4''. (* : forall n : nat, ev n -> ev (4 + n) *)
```

이 정리를 나타내는 방법으로 다음과 같이 두 가지가 더 있다.

```
17 Definition ev_plus4''' : forall (n: nat) (_: ev n), ev (4 + n) :=
18   fun (n : nat) (H : ev n) => ev_SS (S (S n)) (ev_SS n H).
19
20 Definition ev_plus4'''' (n: nat) : ev n -> ev (4 + n) :=
21   fun (H : ev n) => ev_SS (S (S n)) (ev_SS n H).
22
23 Check ev_plus4'''. (* : forall n : nat, ev n -> ev (4 + n) *)
24 Check ev_plus4''''. (* : forall n : nat, ev n -> ev (4 + n) *)
```

Theorem 구문을 사용해도 된다. 이때의 증명 스크립트는 apply 책략의 인수로 증거항을 사용해서 얻는다.

```
25 Theorem ev_plus4''''' : forall (n: nat) (_: ev n), ev (4 + n).
26 Proof.
27   apply (fun (n : nat) (H : ev n) => ev_SS (S (S n)) (ev_SS n H)).
28 Qed.
29
30 Check ev_plus4'''''. (* : forall n : nat, ev n -> ev (4 + n) *)
```

ev_plus4'''의 정의에서 지정한 타입과 Check ev_plus4'''가 보여주는 타입을 비교하면 다음과 같다.

```
31 Definition ev_plus4''' : forall (n: nat), forall (_: ev n), ev (4 + n)
32 Check ev_plus4'''. (* forall (n: nat), ev n -> ev (4 + n) *)
```

forall (_: ev n),과 ev n ->은 동일한 기능을 하고 있음을 알 수 있다. 일반적으로 P -> Q는 forall (_: P), Q의 설탕구문(*syntactic sugar*)이다.

① 직관주의 논리에서 전칭문 $\forall x\, P(x)$의 증명은 x를 인수로 받아 $P(x)$의 증명을 구성하여 리턴하는 함수이다.[4] 이때 '증명'은 증명 스크립트가 아니라 '증거항'을 의미한다. 예를 들어 forall n, ev n -> ev (4 + n)의 증거항은 n을 받아서 ev n -> ev (4 + n)의 증거항을 리턴하는 함수이다.

② 직관주의 논리에서 조건문 $P \to Q$의 증명은 P의 증명을 인수로 받아 Q의 증명을 구성하여 리턴하는 함수(알고리듬)이다.[5] 예를 들어 n이 특정되었을 때 ev n -> ev (4 + n)의 원소는 ev n의 증거항 H를 받아서 ev 4 + n의 증거항인 ev_SS (S (S n)) (ev_SS n H)를 리턴하는 함수가 되어야 한다. 여기서 H는 그 값을 알 수 없는 변수이다.

①과 ②를 결합하면 forall n, ev n -> ev (4 + n)의 증거항은 다음과 같음을 알 수 있다.

```
fun (n: nat) => fun (H: ev n) => ev_SS (S (S n)) (ev_SS n H).
```

이 함수는 (Currying에 의하여) 라인 9, 혹은 라인 18에 나타낸 함수와 동일하다. 이 사실은 여러 방법으로, 예를 들어 다음과 같이 확인할 수 있다.

```
Theorem ev_plus4final : forall n, ev n -> ev (4 + n).
Proof.
  apply (fun (n: nat) => fun (H: ev n) => ev_SS (S (S n)) (ev_SS n H)).
Qed.
```

고전논리에서 한정사는 대부분 $\forall x{\in}A$ 또는 $\exists x{\in}A$와 같은 형태로 상대화 되어$_{\text{relativized}}$ 사용된다. 상대화 되지 않은 상태로 사용하는 순수 한정사는 모든 모델에서 참인$_{\text{logically valid}}$ 논리식에 대해서만 증명이 가능하다. 또한 모델에 결정된 후에도(예를 들어 $\mathbb{N}$) 대부분의 한정사는 상대화 형태로 사용된다. 이런 점에서 실제 수학의 증명에는 ZF가 아니라 타입이론이 사용되고 있다고 볼 수도 있다.

고전논리에서 $\forall$과 $\to$의 관계는, $\forall x \in A, P(x)$가 $\forall x(x \in A \to P(x))$를 의미함을 생각해 보는 것이 이해에 도움이 될 수 있을 것이다. 참고로 $\exists x \in A, P(x)$는 $\exists x(x \in A \wedge P(x))$를 의미한다.

Programming with Tactics

다음의 스크립트는 흥미롭다.

[4]See p227.

[5]See p227.

```
1  Definition add1 : nat -> nat.
2  Proof.
3    intro n.
4    Show Proof.
5    apply S.
6    Show Proof.
7    apply n.
8  Defined.
9
10 Print add1. (* add1 = fun n : nat => S n : nat -> nat *)
11 Compute add1 2. (* =3 : nat *)
```

라인 1은 통상적인 함수 정의에서 사용하는 := 대시 .으로 마무리하였는데, 이것은 증명 스크립트 모드로 들어가도록 하는 것이다. 그리고 증명 모드의 종료를 위하여 Qed.가 아니라 Defined.를 사용하였다.

실은 이 스크립트에서 라인 1의 Definition을 Theorem으로, 라인 8의 Defined를 Qed로 바꾸어도 된다—결과로 얻어지는 add1은 동일하다.

함수를 정의할 때, 이런 식으로 증명 스크립트를 사용하는 것은 때때로, 의존타입을 사용하는 함수를 만들 때 유용하다. 우리는 함수 작성에 앞으로 이 기법을 사용할 것은 아니지만 Coq의 핵심 아이디어를 이해하는 데 이러한 기법을 알아두는 것이 도움이 될 것이라고 생각하여 여기서 소개하였다.

9.2 Logical Connectives as Inductive Types

논리 기호들, 즉 결합자와 한정사들 중에 Coq에 붙박이로 구현된 것은 전칭한정사universal quantifier와 함의implication밖에 없다. (후자는 전자의 축퇴 버전임은 앞서 살펴보았다.) 나머지 기호들은 모두 귀납을 써서 정의된다.

이 장에서 배워야 할 스킬은 주어진 프랍에 대한 증거항을, 증명 리스트를 거치지 않고 직접 만드는 방법이다. 기본 프랍에 대한 에비던스는 사용된 술어의 생성자를 사용하면 되는 것이고, 기본프랍(들)에 전칭한정사와 함의를 사용하여 만든 복합 프랍에 대한 에비던스를 만드는 것은 p210에서 보았다. 이제 나머지 결합자와 한정사에 대한 증거항을 만드는 방법을 알아보기로 한다.

Conjunction

and는 Inductive 정의에서 생성자 conj를 사용하여 다음과 같이 정의할 수 있다. (실제로 Coq의 표준 라이브러리에 이렇게 정의되어 있다.) and는 두 프랍을 받아 하나의 프랍을 리턴하는 함수이고, conj는 두 프랍 각각의 증거항을 받아 두 프랍의 논리곱의 증거항을 리턴하는 함수, 즉 증거항이다.

```
1  Inductive and (P Q : Prop) : Prop :=
2    | conj : P -> Q -> and P Q.
3
4  Arguments conj [P] [Q].
5
```

```
6  Notation "P /\ Q" := (and P Q) : type_scope.
7
8  Check and. (* and : Prop -> Prop -> Prop *)
9  Print and.
10 (* Inductive and (A B : Prop) : Prop := conj : A -> B -> A /\ B. *)
11 Check conj. (* conj : forall A B : Prop, A -> B -> A /\ B *)
12 Print conj.
13 (* Inductive and (A B : Prop) : Prop := conj : A -> B -> A /\ B. *)
```

and와 conj의 관계는 카테시안곱Cartesian product에서 prod와 pair의 관계와 유사하다.

```
Inductive prod (X Y : Type) : Type :=
  | pair : X -> Y -> X * Y.
```

conj가 이렇게 정의되어 있으므로 논리곱 가설에 destruct를 적용할 수 있다. 다음 정리의 증거항은 지금까지 본 것보다 다소 복잡하지만 커리-하워드 대응을 제대로 이해하려면 반드시 이해하고 넘어가야 한다. 다음 스크립트의 각 라인에서 코멘트는 그때 보이는 Coq Goals화면을 간단히 나타낸 것이다.

```
1  Theorem proj1' : forall P Q,
2    P /\ Q -> P.
3  Proof.
4    intros P Q HPQ. (* P, Q : Prop \\ HPQ : P /\ Q \\ Goal: P *)
5    destruct HPQ as [HP HQ]. (* HP : P \\ HQ : Q \\ Goal: P *)
6    apply HP. (* There are no more subgoals *)
7  Qed.
8
9  Print proj1'. (* proj1' = fun (P Q : Prop) (HPQ : P /\ Q) => *)
10    match HPQ with
11      | conj x x0 => (fun (HP : P) (_ : Q) => HP) x x0
12    end
```

라인 7 ∼ 9에 보인 증거항에 대한 설명은 다음과 같다.

> 직관주의 논리에서 조건문 $\forall P, Q,\ P \wedge Q \to P$의 증명은 P와 Q를 인수로 받아 $P \wedge Q \to P$의 증명을 구성하여 리턴하는 함수이다. 리턴값 $P \wedge Q \to P$의 증명은 $P \wedge Q$의 증명을 인수로 받아 P의 증명을 리턴하는 함수이다.
>
> 그러므로 이 증거항은 함수의 합성인데, Currying에 의하여 이 증거항이 3개의 인수 (P Q : Prop) (HPQ : P /\ Q)를 받는 것으로 둘 수 있다. HPQ는 P /\ Q의 유일한 생성자인 conj에 의하여 생성되었을 것이다. 이때 conj의 인수로 사용된 것을 x: P와 x0: Q라 하자. 라인 11의 우변은 리턴값인 x: P가 되어야 한다. 이것은 사영함수 $(x, x_0) \mapsto x$를 인수 x x0에 적용apply하여 얻을 수 있으므로, 표현 (fun (HP : P) (_ : Q) => HP) x x0를 라인 11의 우변에 둔 것이다.

이제 증명 리스트에 대한 설명을 하겠다. x와 x0는 Coq가 내부적으로 사용하는 변수이며, 사용자는 이 디폴트 이름 대신 HP와 HQ라는 이름을 주어 destruct한다. 그러면 컨텍스트에 HP: P와 HQ: Q가 생긴다. HP는 고울 프랍의 증거항이므로 이를 apply하면 된다.

이상으로써 가설 프랍이 논리곱문일 때 어떻게 이를 이용한 증거항을 만드는지를 알아보았다. 이제 고울 프랍이 논리곱문인 경우를 보자. 이 경우 증명 스크립트에는 split 책략을 사용한다. 다음의 예를 보자. split가 세 번 사용되었다.

```
Lemma and_comm : forall P Q : Prop, P /\ Q <-> Q /\ P.
Proof.
  intros P Q. split.
  - intros [HP HQ]. split.
    + apply HQ. + apply HP.
  - intros [HQ HP]. split.
    + apply HP. + apply HQ.
Qed.
```

이 정리의 증거항은 Show Proof를 통해서 알아보면 다음과 같다.

```
(fun P Q : Prop =>
  Logic.conj
  (fun H : P /\ Q => match H with
    | conj x x0 => (fun (HP : P) (HQ : Q) => conj HQ HP) x x0
  end)
  (fun H : Q /\ P => match H with
    | conj x x0 => (fun (HQ : Q) (HP : P) => conj HP HQ) x x0
  end)
)
```

상당히 복잡하다. 이를 다음과 같이 간단하게 만들 수 있다. 우선 라인 2의 Logic.conj는 conj로 써도 된다. conj가 Logic 라이브러리에 속한다는 것은 Coq이 잘 알고 있으므로 Logic을 생략하는 것이다.

증거항 함수는 인수로 2개의 프랍 P Q: Prop을 받아들인다. 인수에 증거항은 없다. 그리고 이 함수의 리턴값은 P /\ Q -> Q /\ P의 증거항과 Q /\ P -> P /\ Q의 증거항의 Logic.conj이다.

Logic.conj의 첫 인수는 라인 3 ∼ 5에 나와 있다. 그것은 P /\ Q -> Q /\ P의 증거항이며, 이는 함수이다. (함수의 리턴값이 아니다.) 이 함수는 P /\ Q의 증거항 H를 인수로 받아 어떤 함수의 리턴값을 리턴하며, 그 함수는 x1 x2를 받아서 conj x2 x1을 리턴한다. 라인 4는 다음과 같이 간단하게 나타낼 수 있다.

```
  | conj x x0 => conj x0 x
```

Logic.conj의 첫 인수인 함수는 다음과 같이 이름을 주어 나타낼 수 있다.[6]

```
Definition and_comm'_aux P Q (H : P /\ Q) : Q /\ P :=
  match H with
  | conj HP HQ => conj HQ HP
  end.
```

Logic.conj의 두 번째 인수는 and_comm'_aux와 대동소이하다. 따라서 라인 1 ∼ 9는 다음과 같이 이름을 주어 나타낼 수 있다.

```
Definition and_comm' P Q : P /\ Q <-> Q /\ P :=
  conj (and_comm'_aux P Q) (and_comm'_aux Q P).
```

and_comm'은 논리곱의 교환법칙, 즉 보조정리 and_comm의 증거항이다. 이 사실은 다음과 같이 두 가지 방법으로 확인된다.

[6]사실은 이 두 함수는 약간 다르다. 전자는 P, Q를 인수로 받지 않고 컨텍스트에 있는 것을 사용하는데 반해 후자는 이들을 인수로 받아들인다는 차이가 있다.

```
1  Theorem and_comm_thrm1 : forall P Q : Prop, P /\ Q <-> Q /\ P.
2  Proof.
3    apply and_comm'.
4  Qed.
5
6  Definition and_comm_thrm2 : forall P Q : Prop, P /\ Q <-> Q /\ P :=
7    and_comm'.
```

Coq이 증명 스크립트로부터 자동으로 구해 준 증거항은 필요이상으로 길어서 읽기 힘든 경우가 많다. 증거항을 잘 이해하고 있으면 이를 상당히 간략하게 만들 수 있으며 이러한 기술은 연습을 통하여 익혀 두길 바란다.

연습문제 9.1 다음 프랍에 대한 증명 스크립트를 작성하고 Show Proof를 이용하여 증거항을 만들어 보라.

```
forall P Q R: Prop, P /\ Q -> Q /\ R -> P /\ R
```

다음의 함수를 사용하여 증명 스크립트를 작성하여라.

```
1  Definition conj_fact_evidence : forall P Q R, P /\ Q -> Q /\ R -> P /\ R :=
2  (fun (P Q R : Prop) (HPQ : P /\ Q) (HQR : Q /\ R) =>
3    conj
4    (match HPQ with | conj x y => x end)
5    (match HQR with | conj x y => y end)).
```

증명 스크립트를 통하여 얻은 증거항이 conj_fact와 동일한 함수임을 설명하여라. Coq로 증명해 보는 것도 좋다. ⊣

Disjunction

이번엔 논리합disjunction이다. 아래의 코드를 상세하게 설명해 보라.

```
1  Inductive or (P Q : Prop) : Prop :=
2    | or_introl : P -> or P Q
3    | or_intror : Q -> or P Q.
4
5  Arguments or_introl [P] [Q].
6  Arguments or_intror [P] [Q].
7
8  Notation "P \/ Q" := (or P Q) : type_scope.
9
10 Check or_introl. (* or_introl : forall P Q : Prop, P -> P \/ Q *)
11 Check or_intror. (* or_intror : forall P Q : Prop, Q -> P \/ Q *)
```

or의 정의에서 라인 1–3은 다음과 같이 바꿔도 된다.

```
Inductive or (P Q : Prop) : Prop :=
  | or_introl (_: P): or P Q
  | or_intror (_: Q): or P Q.
```

다음에 보인 논리합 도입규칙을 설명해 보라. 그리고 inj_r도 만들고 설명해 보라.

```
1  Definition inj_l : forall (P Q : Prop), P -> P \/ Q :=
2    fun P Q HP => or_introl HP.
3
4  Theorem inj_l' : forall (P Q : Prop), P -> P \/ Q.
5  Proof.
6    intros P Q HP. left. apply HP.
7    Show Proof. (* (fun (P Q : Prop) (HP : P) => or_introl HP) *)
8  Qed.
9
10 Check inj_l. (* inj_l : forall (P Q : Prop), P -> P \/ Q *)
```

다음은 논리합 소거 규칙이다. 설명해 보라.

```
11 Definition or_elim : forall (P Q R : Prop),
12   (P \/ Q) -> (P -> R) -> (Q -> R) -> R :=
13     fun P Q R HPQ HPR HQR =>
14       match HPQ with
15       | or_introl HP => HPR HP
16       | or_intror HQ => HQR HQ
17       end.
18
19 Theorem or_elim' : forall (P Q R : Prop),
20   (P \/ Q) -> (P -> R) -> (Q -> R) -> R.
21 Proof.
22   intros P Q R HPQ HPR HQR.
23   destruct HPQ as [HP | HQ].
24   - apply HPR. apply HP.
25   - apply HQR. apply HQ.
26 Qed.
27
28 Theorem or_elim'_thrm1 : forall (P Q R : Prop),
29   (P \/ Q) -> (P -> R) -> (Q -> R) -> R.
30 Proof.
31   apply or_elim'.
32 Qed.
33
34 Definition or_elim'_thrm2 : forall (P Q R : Prop),
35   (P \/ Q) -> (P -> R) -> (Q -> R) -> R :=
36     or_elim'.
```

다음은 논리합의 교환법칙이다.

```
1  Definition or_commut' : forall P Q, P \/ Q -> Q \/ P :=
2  (fun (P Q : Prop) (HPQ : P \/ Q) => match HPQ with
3    | or_introl x => (fun HP : P => or_intror HP) x
4    | or_intror x => (fun HQ : Q => or_introl HQ) x
5  end).
6
7  Check or_commut'. (* or_commut' : forall P Q : Prop, P \/ Q -> Q \/ P *)
```

Existential Quantification

Coq에서 존재한정사는 전칭한정사universal quantifier와 함의implication를 사용하여 다음과 같이 귀납적으로 정의한다.

P : A -> Prop가 1항술어일 때 ex P는 P를 만족하는 A 의 원소가 존재한다는 것을 나타내는 프랍이다. ex P는 $\{x \in \mathsf{A} \mid \mathsf{P}\ x\} \neq \varnothing$와 같은 의미이다. 이것을 exists x, P x로 쓸 수 있다—exists는 ex의 설탕구문이다.

정의를 잘 보기 바란다. $\forall x \in \mathsf{A}, (\mathsf{P}\ x \to \mathsf{ex}\ \mathsf{P})$이다. $(\forall x \in \mathsf{A}, \mathsf{P}\ x) \to \mathsf{ex}\ \mathsf{P}$가 아니다.

```
1 Inductive ex {A : Type} (P : A -> Prop) : Prop :=
2   | ex_intro : forall x : A, P x -> ex P.
3
4 Notation "'exists' x , p" :=
5   (ex (fun x => p))
6     (at level 200, right associativity) : type_scope.
7
8 Check ex_intro.
9 (* ex_intro : forall (A : Type) (P : A -> Prop) (x : A), P x -> exists y, P y *)
```

직관주의 논리에서 $\exists x, P(x)$의 증명은 $P(x)$를 만족하는 원소 x와 $P(x)$의 증명 H의 순서쌍 (x, H)이다. (See p227.) Coq에서는 $\exists x, P(x)$를 ex P로 두고 이것의 원소, 즉 증거항을 생성자 ex_intro를 써서 만들어 낸다. 라인 2에 있는 ex_intro의 정의는 어쩌면 다음과 같이 쓰는 것이 더 이해하기에 도움될 수도 있다.

```
  | ex_intro (x : A) (_ : P x):  ex P.
```

ex_intro는 증거항을 리턴하는 함수로 볼 수 있으며, 타입은 라인 9에 나타나 있다. 4개의 인수를 취하는 함수인 것으로 보이는데 첫 번째 인수 (A: Type)은 암묵적인 것이므로 생략하면 된다. 두 번째 인수 (P: A -> Prop)는 라인 2에는 없고 라인 1에 나타나 있다. 세 번째 인수 (x: A)는 Px가 성립하는 자연수이고, 네 번째 인수 (H: P x)는 P x의 증거항이다. 이 x와 H의 순서쌍을 ex P의 증명으로 볼 수 있다.

예를 들어 P := ev일 때 순서쌍 (0, ev_0)를 exists x, P x의 증명으로 볼 수 있는데, Coq에서는 이 증명을 다음과 같은 증거항으로 나타낸다. 여기서 ex_intro의 첫 번째 인수는 (암묵적 타입 인수 이므로) 생략되었고, 두 번째 인수 (P: A -> Prop)의 값으로 fun n => ev n이 사용되었음에 주목하라.

```
ex_intro (fun n => ev n) 0 ev_0
```

그러므로 다음과 같은 증명이 가능하다.

```
Fact ex_intro: exists n, ev n.
Proof.
  apply (ex_intro (fun n : nat => ev n) 0 ev_0).
Qed.
```

그런데 위의 증명을 다음과 같이 변형해도 된다.

```
Fact ex_intro': exists n, ev n.
Proof.
  apply (ex_intro ev 0 ev_0).
Qed.
```

이래도 되는 이유를 단순히 (fun n : nat => ev n) 0 = ev 0가 성립하기 때문이라고 하면 안 된다. 함수 어플리케이션은 왼쪽 결합을 따르며 일반적으로 결합법칙이 성립하지 않기 때문이다. 하지만 이 경우에는 된다. ev의 타입이 nat -> Prop으로서 ex_intro의 첫 인수의 타입 조건을 만족하고 또한 결합법칙이 성립하기 때문이다.

세 번째 인수로 0 대신 2를 사용하면 다음과 같다.

```
Fact ex_intro2: exists n, ev n.
Proof.
  apply (ex_intro (fun n : nat => ev n) 2 (ev_SS 0 ev_0)).
Qed.
```

혹은 다음과 같이 써도 된다.

```
Fact ex_intro2': exists n, ev n.
Proof.
  apply (ex_intro ev 2 (ev_SS 0 ev_0)).
Qed.
```

세 번째 인수로 4를 써도 물론 된다.

```
Definition some_nat_is_even : exists n, ev n :=
  ex_intro ev 4 (ev_SS 2 (ev_SS 0 ev_0)).
```

ev를 _로 바꿔도 된다. 왜냐하면 Coq은 인수를 추론할 수 있기 때문이다.

```
Definition some_nat_is_even' : exists n, ev n :=
  ex_intro _ 4 (ev_SS 2 (ev_SS 0 ev_0)).
```

exists n, ev n: Prop과 같은 존재문의 경우에는 증거항이 여러 개 있을 수 있음을 보았다.

<u>**연습문제**</u> **9.2** 증거항이 두 개 이상인 프랍을 존재한정사를 사용하지 않고 만들어 보라.

(힌트). 논리합 ⊣

<u>**예제**</u> **9.3** 다음의 코드는 무엇을 하고 있는 것인지 살펴보자.

```
1 Definition ex_ev_Sn : ex (fun n => ev (S n)) :=
2   (ex_intro (fun n : nat => ev (S n)) 1 (ev_SS 0 ev_0)).
```

ex (fun n => ev (S n))를 타입으로 가지는 inhabitant를 정의하고 있다. 타입추론에 의해서 n: nat임을 알 수 있다. ex의 인수 P : A -> Prop으로 (fun n => ev (S n))를 사용했으므로 P n은 ev (S n)이다. 우리가 찾는 것은 ex P, 즉 exists x, ev (S x)의 증거항이다. 그 다음 수가 짝수인 것이 존재함을 증명해야 하므로 증인으로 1을 취하면 되겠다.

ex (fun n => ev (S n))의 원소를 만들어 내는 생성자 ex_intro는 첫 번째 인수로 ex의 인수인 <u>술어</u> P를 취한다. 그 다음 인수로는 <u>증인</u>witness이 되는 원소를 쓰고 마지막 인수로 P x, 즉 ev (S x)의 <u>증거항</u>을 쓴다. 지금 x := 1을 사용하고 있으므로 ev (S 1), 즉 ev 2의 증거항을 마지막 인수로 써야 한다. 이것은 ev_SS 0 ev_0이다. ⊣

True and False

True와 False는 상수 술어nullary predicate, 즉 프랍이다. 전자에는 증거항이 존재하고 후자에는 존재하지 않는다.

True의 귀납적 정의는 아주 간단하다.

```
Inductive True : Prop :=
  | I : True.

Definition p_implies_true : forall P, P -> True :=
  fun (P: Prop) (H: P) => I.

Check p_implies_true. (* p_implies_true : forall P : Prop, P -> True *)
```

False의 정의는 더 간단하다. 이것은 증거항이 존재하지 않는 공타입이므로 생성자가 없다.

```
Inductive False : Prop :=.
```

가설에 False가 있으면 그것의 에비던스를 destruct하면 증명이 완료된다.

```
1 Fact contra0 : False -> 0 = 1.
2 Proof.
3   intros contra.
4   destruct contra.
5 Qed.
```

contra는 False의 증거항이라고 했으므로 그것을 destruct하려는데, 애초에 construct된 적이 없으므로 destruct할 수도 없다. 지금 우리가 찾아야 하는 것은 False의 증명들로 이루어진 집합을 정의역으로 하고 0 = 1의 증명들로 이루어진 집합을 공역으로 하는 함수를 만들어 내는 것이다. 그런데 지금 정의역이 공집합이다. (공역도 공집합인것으로 보이는데 이것은 현재 우리가 찾으려는 증거항의 존재 여부와 상관이 없다.) 공집합을 정의역으로 하는 함수는 (공역이 공집합이든 아니든 항상) 존재한다. 공함수empty function가 그것이다. 이것으로 증명이 완료된 것이다.[7] 다시 말하면 공함수는 모든 프랍 P에 대해서 False -> P의 증거항이 된다.

다음 예에서 사용한 표현 ex_falso_quodlibet는 라틴어로 'from falsehood, (you get) anything'이라는 뜻이라고 한다. 라인 2는 Coq에서 공함수를 나타내는 방법이다.

```
1  Definition false_implies_zero_eq_one : False -> 0 = 1 :=
2    fun contra => match contra with end.
3
4  Check false_implies_zero_eq_one.
5  (* false_implies_zero_eq_one : False -> 0 = 1 *)
6
7  Fact ex_falso_quodlibet0 : forall (P: Prop), False -> P.
8  Proof.
9    intros P contra.
10   destruct contra.
11   Show Proof.
12 Qed.
```

[7] 이러한 추론규칙을 *ex falso*라고 하며 이는 라틴어로 '거짓으로부터'라는 뜻이다. Ex falso는 간단히 *falsum*이라고도 한다. Coq은 이 추론규칙을 destruct 책략으로 구현해 놓았다.

```
13
14 Definition ex_falso_quodlibet' : forall P, False -> P :=
15 (fun (P : Prop) (contra : False) => match contra with
16 end).
17
18 Check ex_falso_quodlibet'. (* ex_falso_quodlibet' : forall P : Prop, False -> P *)
```

False의 증명은 존재하지 않는다. 그러나 컨텍스트에 모순이 있으면 False의 증명을 만들어 낼 수 있다. 예를 들어 False -> False : Prop은 ex_falso_quodlibet0에서 P := False인 경우에 해당하므로 증명이 존재한다. 이 경우 라인 9는 intros contra가 될 것이며 이때의 고울 프랍은 False이다. 이 고울 프랍은 컨텍스트에 모순이 있으므로 destruct 책략을 써서 증명할 수 있다.

Negation

부정 결합자는 Coq에서 ~ 기호를 사용하여 나타낸다. ~ P는 P -> False를 줄여쓰기abbreviate 한 것으로 본다. 부정 결합자를 사용하는 프랍의 예로 ~ (P /\ ~ P)를 들어 이것의 증거항을 찾아보자. 이 프랍은 일종의 배중률LEM이다.

```
Definition LEM' : forall P, ~ (P /\ ~ P) :=
  fun (P : Prop) (H : P / (P -> False)) =>
    match H with
    | conj HP NHP => NHP HP
    end.
```

HP : P이고 NHP : P -> False를 가정하면 BHK 해석에 의하여 NHP HP : False이다.

9.3 Equality

Coq는 등호도 귀납적으로 정의하여 사용한다. Notation ==은 지금 잠시 사용하는 로컬 스코프에서만 존재한다.

```
Module EqualityPlayground.

Inductive eq {X: Type} : X -> X -> Prop :=
  | eq_refl : forall x, eq x x.

Notation "x == y" := (eq x y)
                     (at level 70, no associativity)
                     : type_scope.

Lemma four : 2 + 2 == 1 + 3.
Proof.
  apply eq_refl. (* [simpl] can be omitted *)
Qed.

Definition four' : 2 + 2 == 1 + 3 :=
  eq_refl 4.
```

우리가 많이 사용했던 reflexivity 책략은 apply eq_refl의 축약이다. 그런데 reflexivity는 원래 =에 대해서 정의된 것이므로 여기서의 ==에 대해서는 작동하지 않는다. rewrite에 대해서도 마찬가지로 말할 수 있다. 다음 스크립트의 라인 15와 라인 17에서 이를 볼 수 있다.

```
Definition singleton : forall (X: Type) (x: X), []++[x] == x::[] :=
  fun (X:Type) (x:X) => eq_refl [x].

Definition eq_add :
  forall (n1 n2 : nat), n1 == n2 -> (S n1) == (S n2) :=
    fun n1 n2 Heq =>
      match Heq with
        | eq_refl n => eq_refl (S n)
      end.

Theorem eq_add' :
  forall (n1 n2 : nat), n1 == n2 -> (S n1) == (S n2).
Proof.
  intros n1 n2 Heq.
  Fail rewrite Heq.
  destruct Heq.
  Fail reflexivity.
  apply eq_refl.
Qed.
```

다음은 두 리스트는 헤드와 테일이 각각 일치하면 동일하다는 사실의 증명이다.

```
Definition eq_cons : forall (X : Type) (h1 h2 : X) (t1 t2 : list X),
    h1 == h2 -> t1 == t2 -> h1 :: t1 == h2 :: t2 :=
  fun (X: Type) (h1 h2: X) (t1 t2: list X)
      (Hh: h1 == h2) (Ht: t1 == t2) =>
  match Hh with
  | eq_refl h => match Ht with
    | eq_refl t => eq_refl (hoho)
    end
  end.

Check eq_cons.

Fact eq_cons0 : forall (X : Type) (h1 h2 : X) (t1 t2 : list X),
    h1 == h2 -> t1 == t2 -> h1 :: t1 == h2 :: t2.
Proof.
  intros X h1 h2 t1 t2 Hh Ht.
  destruct Hh as [h] eqn:Eqh. (* can omit [as ..] *)
  destruct Ht as [t] eqn:Eqt. (* can omit [as ..] *)
  apply eq_refl.
  Show Proof. (* longer than the hand-written version *)
Qed.
```

라인 17, 18에서 as 이하를 생략하고 화면 변화를 관찰하여라.

'같음'의 정의 중에 가장 중요한 것 2개를 꼽으면 Leibniz equality와 Definitional equality를 들 수 있다. 우리가 정의한 등호가 전자와 동등함을 증명해 보자.

먼저 순방향이다.

```
1 Lemma equality__leibniz_equality : forall (X : Type) (x y: X),
2   x == y -> forall (P : X -> Prop), P x -> P y.
3 Proof.
4   intros X x y Hxy P Hx.
5   destruct Hxy as [z] eqn:Eq.
6   apply Hx.
7   Show Proof. (* longer than necessary, and ugly *)
8 Qed.
```

라인 4를 실행한 후의 고울 화면은 다음과 같다.

```
.. snip ..
Hxy : x == y
P : X -> Prop
Hx : P x
(1 / 1) ----------------
P y
```

여기서 라인 5를 실행하면 다음과 같은 화면이 나타난다.

```
.. snip ..
Hxy : z == z
P : X -> Prop
Hx : P z
Eq : Hxy = eq_refl y
(1 / 1) ----------------
P z
```

왜 이렇게 되는지 살펴보자.

원래의 `Hxy : x == y`는 생성자 `eq_refl : forall x, x = x`에 의하여 만들어졌을 것이다. 이 생성자의 묶인변수를 `x := z`로 놓으면 `z = z`가 `x = y`에 매치되어야 하므로 `x`와 `y`는 모두 `z`로 바뀌게 된다. `Hx`와 고울 프랍도 각각 `P z`로 바뀐다. 라인 5의 `as [z]`는 `as [x]`, `as [y]`, `as [zz1]` 등 어떤 변수로 바꾸어도 상관없다.

이번에는 증거항을 직접 만들어서 증명해 보자.

```
1 Definition equality__leibniz_equality_term :
2   forall (X : Type) (x y: X),
3     x == y -> forall P : (X -> Prop), P x -> P y :=
4   fun (X : Type) (x y: X) (Hxy : x == y) (P : X -> Prop) (Hx : P x) =>
5     match Hxy with
6     | eq_refl x0 => fun (Hx : P x0) => Hx
7     end Hx.
```

`Hxy : x == y`가 `eq_refl x0 : eq x0 x0`에 매치되어야 하므로 모든 `x`, `y`는 `x0`로 바뀌게 된다. 그러므로 `Hx : P x`는 `Hx : P x0`로 바뀌고, 우리가 증거항을 찾기를 원하는 프랍 `P y`도 역시 `P x0`로 바뀐다. 이것의 증거항으로는 `Hx`를 그대로 사용하면 될 것이다. 그렇다고 라인 5 ~ 7을 딸랑 `Hx`로 쓰는 것은 안 된다. 왜냐하면 `Hx`가 무엇인지는 `match Hxy with .. end`를 통해서 알 수 있기 때문이다. 따라서 `match .. end`의 결과로 얻은 함수를 `Hx`에 적용한 것을 리턴하는 함수를 증거항으로 삼으면 된다.

다음은 역방향이다.

```
1  Lemma leibniz_equality__equality : forall (X : Type) (x y: X),
2    (forall P: X -> Prop, P x -> P y) -> x == y.
3  Proof.
4    intros X x y H.
5    apply H.
6    apply eq_refl.
7    Show Proof. (* (fun (X : Type) (x y : X) (H : forall P : X -> Prop, P x -> P y) =>
8                     H (fun y0 : X => x == y0) (eq_refl x)) *)
9  Qed.
10
11 End EqualityPlayground.
```

라인 7 ~ 8에 나타난 증거항은 특별히 더 줄일 것이 없으니 그대로 사용하면 될 것이다. `H`의 인수인 술어 `P : X -> Prop`로 무엇을 취할 것인지가 관건이다.

우리가 원하는 것은 `x == y`의 증거항이다. 이것은 `eq_refl`로 만드는 수밖에 없다. 그리고 이 증거항의 타입 `x == y`는 `H`의 타입(즉 프랍)의 후건 `P y`가 되어야 할 것이다.

`P`를 도입할 당시 `x`는 이미 도입되어 있는 자유변수이다. `P`를 `fun y0: X => x == y0`로 두면 `P x`는 `x == x`이므로 증거항 `eq_refl x`를 가질 것이며, `P y`는 `x == y`인데, 이것의 증거항으로는 바로 `H (fun y0 : X => x == y0) (eq_refl x)`가 존재한다.

Inversion, Again

`inversion` 책략은 강력하지만 하는 일이 상당히 많은 관계로 파악하기가 좀 어렵다. SF에서는 다음과 같이 설명한다.

We've seen `inversion` used with both equality hypotheses and hypotheses about inductively defined propositions. Now that we've seen that these are actually the same thing, we're in a position to take a closer look at how `inversion` behaves.

In general, the `inversion` tactic...

- takes a hypothesis `H` whose type `P` is inductively defined, and
- for each constructor `C` in `P`'s definition,
  - generates a subgoal in which we assume `H` was built with `C`,
  - adds the arguments (premises) of `C` to the context of the subgoal as extra hypotheses,
  - matches the conclusion (result type) of `C` against the current goal and calculates a set of equalities that must hold in order for `C` to be applicable,
  - adds these equalities to the context (and, for convenience, rewrites them in the goal), and
  - if the equalities are not satisfiable (e.g., they involve things like `S n = O`], immediately solves the subgoal.

9.4 More Stuff

Coq's Trusted Computing Base

왜 우리는 Coq을 믿어야 하는가? 혹시 버그가 있다면 어쩔려구?

이 질문에 대하여 완벽히 만족스러운 답을 내놓기는 어렵겠지만, 커리-하워드 대응에 의하여 Coq의 신뢰성은 타입 확인 알고리듬type-checking algorithm으로 환원된다. 타입체커는 비교적 간단한 프로그램이라서 오류가 개입될 여지가 극히 적다.

More Exercies

다음 7개의 연습문제는 모두 증거항을 직접 찾아내는 것이다.

```
Definition and_assoc : forall P Q R : Prop,
    P /\ (Q /\ R) -> (P /\ Q) /\ R :=
  fun (P Q R : Prop) (H : P /\ (Q /\ R)) =>
    match H with
    | conj HP (conj HQ HR) => conj (conj HP HQ) HR
    end.

Check and_assoc.
(* forall P Q R : Prop, P /\ Q /\ R -> (P /\ Q) /\ R *)

Definition or_distributes_over_and : forall P Q R : Prop,
    P \/ (Q /\ R) <-> (P \/ Q) /\ (P \/ R) :=
  fun (P Q R : Prop) =>
    conj
    (fun (H : P \/ (Q /\ R)) => match H with
      | or_introl HP => conj (or_introl HP) (or_introl HP)
      | or_intror (conj HQ HR) => conj (or_intror HQ) (or_intror HR)
    end)
    (fun (H : (P \/ Q) /\ (P \/ R)) => match H with
      | conj (or_introl HP) (or_introl HP') => or_introl HP
      | conj (or_introl HP) (or_intror HR) => or_introl HP
      | conj (or_intror HQ) (or_introl HP) => or_introl HP
      | conj (or_intror HQ) (or_intror HR) => or_intror (conj HQ HR)
    end).

Check or_distributes_over_and.
(* forall P Q R : Prop, P \/ Q /\ R <-> (P \/ Q) /\ (P \/ R) *)

Definition double_neg : forall P : Prop,
    P -> ~~P :=
  fun (P : Prop) (H : P) => fun (H' : P -> False) => H' H.

Check double_neg.
(* forall P : Prop, P -> ~ ~ P*)

Definition contradiction_implies_anything : forall P Q : Prop,
    (P /\ ~P) -> Q :=
  (fun (P Q : Prop) (H : P /\ ~ P) =>
```

```
    match H with
          | conj x x0 => (fun (HP : P) (HNP : ~ P) =>
                 match (HNP HP) with end) x x0
    end).

Check contradiction_implies_anything.
(* forall P Q : Prop, P /\ ~ P -> Q *)

Definition de_morgan_not_or : forall P Q : Prop,
    ~ (P \/ Q) -> ~P /\ ~Q :=
  fun (P Q : Prop) (H : ~ (P \/ Q)) =>
    conj (fun (HP : P) => H (or_introl HP))
         (fun (HQ : Q) => H (or_intror HQ)).

Check de_morgan_not_or.
(* forall P Q : Prop, ~ (P \/ Q) -> ~ P /\ ~ Q *)

Definition curry : forall P Q R : Prop,
    ((P /\ Q) -> R) -> (P -> (Q -> R)) :=
  fun (P Q R : Prop) (H : (P /\ Q) -> R) (HP : P) (HQ : Q) =>
    H (conj HP HQ).

Check curry.
(* forall P Q R : Prop, (P /\ Q -> R) -> P -> Q -> R *)

Definition uncurry : forall P Q R : Prop,
    (P -> (Q -> R)) -> ((P /\ Q) -> R) :=
  fun (P Q R : Prop) (H : P -> (Q -> R)) (HPQ : P /\ Q) =>
    match HPQ with
    | conj HP HQ => (fun (HP : P) (HQ : Q) => H HP HQ) HP HQ
    end.

Check uncurry.
(* forall P Q R : Prop, (P -> Q -> R) -> P /\ Q -> R *)
```

Proof Irrelevance (Advanced)

Functional extensionality와 비슷한 propositional extensionality는 다음과 같이 정의된다.

```
Definition propositional_extensionality : Prop :=
  forall (P Q : Prop), (P <-> Q) -> P = Q.
```

연습문제 2개.

```
1 Theorem pe_implies_or_eq :
2   propositional_extensionality ->
3   forall (P Q : Prop), (P \/ Q) = (Q \/ P).
4 Proof.
5   intros PE P Q.
6   apply PE.
7   split.
8   - intros [HP | HQ].
9     + right. apply HP.
```

```
10      + left. apply HQ.
11    - intros [HQ | HP].
12      + right. apply HQ.
13      + left. apply HP.
14  Qed.
15
16  Lemma pe_implies_true_eq :
17    propositional_extensionality ->
18    forall (P : Prop), P -> True = P.
19  Proof.
20    intros PE P HP.
21    apply PE.
22    split.
23    - intros _. apply HP.
24    - intros _. apply I.
25  Qed.
```

Proof_irrelevance는 동일한 프랍에 대한 증거항은 모두 같다는 주장이다. 이것은 다음과 같이 정의할 수 있다.

```
Definition proof_irrelevance : Prop :=
  forall (P : Prop) (pf1 pf2 : P), pf1 = pf2.
```

다음의 연습문제는 아직 배우지 않은 기법을 사용한다.

```
1  Theorem pe_implies_pi :
2    propositional_extensionality -> proof_irrelevance.
3  Proof.
4    intros PE P pf1 pf2.
5    assert (H : P <-> True).
6    { split.
7      - intros. apply I.
8      - intros _. apply pf1.
9    }
10   apply PE in H. (* Hypothesis [H : P = True] appears in the context. *)
11   revert pf1. revert pf2.
12   rewrite H.
13   intros pf1 pf2.
14   destruct pf1. destruct pf2.
15   reflexivity.
16 Qed.
```

라인 10을 실행한 후에 Coq Goals 화면은 다음과 같다.

```
17  PE : propositional_extensionality
18  P : Prop
19  pf1, pf2 : P
20  H : P = True
21  (1 / 1) ------------------------
22  pf1 = pf2
```

라인 11의 revert는 고울이 전칭문 forall x: T, ...인 경우의 intros의 역작업이다.[8] 따라서 라인 11 실행 후에 Coq Goals 화면은 다음과 같다.

[8] intros는 특정화particularization, revert는 일반화generalization로 보면 될 것이다.

```
23 PE : propositional_extensionality
24 P : Prop
25 H : P = True
26 (1 / 1) ------------------------
27 forall pf2 pf1 : P, pf1 = pf2
```

라인 13 실행 후에 Coq Goals 화면은 다음과 같다.

```
28 PE : propositional_extensionality
29 P : Prop
30 H : P = True
31 pf1, pf2 : True
32 (1 / 1) ------------------------
33 pf2 = pf1
```

결국 코드에서 라인 11–13이 한 일은 라인 19 `pf1, pf2 : P`를 라인 31 `pf1, pf2 : True`로 바꾼 것이다. 이제 라인 14의 `destruct`가 제 역할을 한다.

주의: 프랍이 술어를 포함하고 있는 경우에는 그 프랍의 모든 증거항이 동일할 필요가 없다.

Intutitionistic Logic

수학에서의 구성주의constructivism는 어떤 객체가 존재함을 증명하려면 그것을 구성해야만 한다는 믿음, 또는 사상이다. 이것의 가장 두드러진 특징은 배중률law of excluded middle, 또는 이와 동등하다고 볼 수 있는 2중부정소거double negation elimination를 받아들이지 않는다는 것이다.

직관주의intuitionism는 20세기 초에 L.E.J. Brower에 의해서 주창된 것으로 수학적인 객체와 진실은 인간의 마음과 독립적으로 존재할 수 없는 정신적 구성이라는 믿음이다. 이것은 구성주의의 매운맛 버전, 혹은 부분집합으로 볼 수 있다.

많은 사람들이 구성주의와 직관주의를 구별하지 않고 혼용해서 사용한다. 이 책도 그러하다.

19세기말에서 20세기 초에 수학 기초론에는 3가지 학파가 있었다.

- 형식주의formalism: 수학은 정해진 규칙에 따른 기호들의 조작이다. David Hilbert (1862-1943), Paul Bernays (1888-1977), Wilhelm Ackermann (1896-1962)
- 구성주의constructivism: 수학은 구성적인 행위이다. Henri Poincaré (1854-1912), Hermann Weyl (1885-1955), L.E.J. Brower (1881-1966)
- 공리주의logicism: 수학은 논리학의 일부이다. Gottlob Frege (1848-1925), Bertrand Russell (1872-1970), Alfred North Whitehead (1861-1947)

이 학파의 주장은 대략 이러한데, 이것이 정확히 무엇을 의미하는지는, 그 주장들이 수학의 언어로 기술되지 않았으므로, 알 수 없다.

브라우어의 주장은 그의 제자 Arend Heyting (1898-1980)에 의하여 어느 정도 수학화 되었다.[9] 그가 만든 Heyting Algebra는 다음과 같이 정의되며 부울리언 앨지브라의 일반화로 볼 수 있다.

[9]참고로 이들은 모두 네덜란드인이다. Heyting의 제자 중에는 Anne Troelstra (1939-2019)과 Dirk van Dalen (1932 ~)이 있다.

<u>정의</u> **9.4** A *Heyting algebra* H is a bounded lattice such that for all a and b in H there is a greatest element x of H such that $a \wedge x \leqslant b$.

This element is the *relative pseudo-complement* of a with respect to b, and is denoted $a \to b$. We write 1 and 0 for the largest and the smallest element of H, respectively.

In any Heyting algebra, one defines the *pseudo-complement* $\neg a$ of any element a by setting $\neg a = (a \to 0)$. By definition, $a \wedge \neg a = 0$, and $\neg a$ is the largest element having this property. However, it is not in general true that $a \vee \neg a = 1$, thus $\neg$ is only a pseudo-complement, not a true complement, as would be the case in a Boolean algebra. ⊣

하이팅 앨지브라와 직관주의 논리와의 관계는 부울리언 앨지브라와 전통적 논리와의 관계와 같다고 보면 된다.

직관주의 논리의 시맨틱스는 집합론을 이용하지 않는다. 증명이 존재하는 명제를 참이라고 보며 증명의 의미는 BHK interpretation (Brower-Heyting-Kolmogorov interpretation)에 의하여 정의된다.[10]

- A proof of $P \wedge Q$ is a pair $\langle a, b \rangle$ where a is a proof of P and b is a proof of Q.
- A proof of $P \vee Q$ is either $\langle 0, a \rangle$ where a is a proof of P or $\langle 1, b \rangle$ where b is a proof of Q.
- A proof of $P \to Q$ is a function f that converts a proof of P into a proof of Q.
- A proof of $\exists x \in S,\ P(x)$ is a pair $\langle x, a \rangle$ where x is an element of S and a is a proof of $P\,x$.
- A proof of $\forall x \in S,\ P(x)$ is a function f that converts an element x of S into a proof of $P\,x$.
- The formula $\neg P$ is defined as $P \to \bot$, so a proof of it is a function f that converts a proof of P into a proof of $\bot$.
- There is no proof of $\bot$.

BHK는 여러 결합자와 한정기호 각각에 대해서 그것을 주 연산자로 사용하여 만든 명제에 대한 증명의 정의를 말하고 있다. 예를 들어 $P \to Q$의 증명에 대한 정의를 따르면 $\neg(P \wedge \neg P) :\equiv (P \wedge (P \to \bot)) \to \bot$의 증명은 다음과 같음을 알 수 있다: $(a, b) \mapsto b(a)$

방금 증명한 $\neg(P \wedge \neg P)$는 전통적 논리에서는 $P \vee \neg P$와 동등하다. 하지만 BHK에 따르면 직관주의 논리에서는 $P \vee \neg P$를 증명하려면 P를 증명하거나 혹은 $\neg P$를 증명해야 한다. 따라서 P에 대한 아무런 정보가 없는 경우에는, 예를 들어 P가 명제를 나타내는 변수인 경우에는 $P \vee \neg P$는 증명할 수 없다.

직관주의 논리에 대한 책에서는 정작 중요한 아톰 프랍atomic proposition에 대한 증명은 설명하지 않는 경우가 많다. Coq에서는 기본 술어들에 대한 증명은 그 술어의 귀납적 정의에 따라서 귀납적으로 증명한다.

[10]Brower는 여기에 이름은 들어있지만 하이팅이 그냥 넣어준 것이며, 정작 브라우어 본인은 이 해석을 별로 탐탁치 않게 여겼다고 한다. 러시아의 천재 수학자 콜모고로프는 하이팅과 독립적으로 이걸 발견했다.

직관주의 논리의 시맨틱스를 더 깊이 이해하기 위하여 Kripke semantics와 Kripke model을 알아두는 것이 좋다. 이에 대한 설명은 생략한다.[11]

이제까지의 내용은 철학적, 사변적인 것이며 실제 수학의 증명에는 1도 응용되지 않는다. 뜻밖에도 직관주의 논리의 수학적 증명에의 사용은 컴퓨터 과학에 의해서 시작되었다.

Alonzo Church, Haskell Curry는 Lambda Calculus의 초기 연구자들이다. Henk P. Barendregt는 Lambda Calculus의 타입 이론을 크게 발전시켰다. 특히 Lambda Cube가 중요하다. Dana Scott, Tony Hoare 등은 programming language semantics에 기여했다. Thierry Coquand, Benjamin C. Pierce는 1960년대에 출생하여 아직까지 활발히 활동중인 연구자들이다.

Per Martin-Löf (1942, Swedish) is indeed a pivotal figure in the intersection of logic and computer science, particularly known for his contributions to type theory and constructive mathematics.

Per Martin-Löf's work bridges foundational aspects of mathematical logic with practical applications in computer science, especially in the domains of programming language design and automated proof verification.

Intuitionistic Type Theory

마틴뢰프Per Martin-Löf가 1970년대에 창시한 `Intuitionistic Type Theory` [1]는 직관주의 논리의 시맨틱스를 타입이론으로 해석한 것이다.

직관주의 논리에 대한 수학적 이론인 하이팅 앨지브라Heyting Algebra, 크립케 시맨틱스Kripke Semantics 등은 직관주의 논리의 구체화에 충분한 도구를 제공하지 못하고 있었는데 마틴뢰프는 직관주의 타입이론으로써 그 돌파구를 찾았다고 본다.

이 이론의 핵심을 이루는 개념은 proposition과 judgement이다. 이들은 수학적으로 엄격하게 정의되어 있지 않아서 (인문학적으로 정의되었다는 느낌) 수학자들이 이해하기 힘들 수 있다. 대략 프랍은 closed formula(sentence)이고 저지먼트는 $\vdash A$ 형태의 문장이라고 생각하면 될 것 같다. 중요한 것은 증명나무가 프랍이 아니라 저지먼트로 레이블된다는 사실이다. 다음은 [1]에서 발췌한 것이다.

> Here the distinction between proposition and assertion or judgement is essential.
> ... snip ...
>
> Contrary to formulas, propositions are not defined inductively. So to speak, they form an open concept. In standard textbook presentations of first order logic, ... snip ...

이와 관련된 내용을 p10에서 찾아볼 수 있다.

[11]참고로 Saul Kripke (1940-2022)는 최종학력이 Harvard University 수학과 B.S.이다. 양상논리의 시맨틱스에 비상한 업적을 남겼으며 CUNY의 석좌교수를 역임했고 프린스턴의 명예교수였다. 제자는 많지 않아 2명의 Ph.D.를 길러냈다.

10

Induction Principles

우리가 새로운 Inductive 데이터 타입 T를 정의할 때마다 Coq은 자동으로 그 타입에 대한 귀납 원리(*induction principle*)를 만들어 낸다. 이 원리는 정리이며, 우리는 그 타입에 대한 프랍을 귀납적으로 증명할 때 통상적인 정리와 동일한 방법으로 이 원리를 사용할 수 있다.

귀납적 타입 foo에 대응하는 귀납 원리의 이름은 foo_ind이다.

Basics

nat: Type에 대한 귀납원리는 다음과 같다.

```
Check nat_ind :
  forall P : nat -> Prop,
    P 0 ->
    (forall n : nat, P n -> P (S n)) ->
    forall n : nat, P n.
```

(induction 책략 대신에) 귀납 원리를 사용하는 증명의 예를 아래 보였다.

```
1  Theorem plus_one_r' : forall n: nat,
2    n + 1 = S n.
3  Proof.
4  apply nat_ind.
5  - simpl. reflexivity.
6  - intros n IHn.
7    simpl. rewrite -> IHn. reflexivity.
8  Qed.
```

nat_ind의 후건 forall n: nat, P n은 forall n: nat, n + 1 = S n에 매치된다. 따라서 라인 4의 apply nat_ind를 실행하면 2개의 서브고울 P 0와 forall n: nat, P n -> P (S n)이 생성된다.

```
9  Goal 1
10 (1 / 2) ------------------------
11 0 + 1 = 1
12
13 Goal 2
14 (2 / 2) ------------------------
15 forall n : nat, n + 1 = S n -> S n + 1 = S (S n)
```

라인 6을 실행한 후에 Coq Goals 화면은 다음과 같다.

```
16 n : nat
17 IHn : n + 1 = S n
18 (1 / 1) ------------------------
19 S n + 1 = S (S n)
```

그 다음은 설명할 필요가 없을 것이다.

plus_one_r'을 induction 책략을 사용하여 증명하면, 즉 라인 4를 induction n으로 바꾸어 실행하면 Goal 1은 라인 9–11과 동일하고, Goal 2에는 컨텍스트에 라인 16–17의 내용이 추가된다. 따라서 더 간편하게 증명을 진행할 수 있다. 라인 17은 귀납 가설induction hypothesis라고 부르며 induction 책략을 사용하면 통상 이름이 IH로 시작한다.

따라서 induction 책략은 foo_ind를 정리로 취급하여 apply foo_ind를 수행하는 wrapper라고 보면 된다.

지금까지 foo가 nat인 아주 간단한 경우를 보았는데, 이번에는 아주 조금 더 복잡한 경우를 살펴보자.

```
1 Inductive natlist : Type :=
2   | nnil
3   | ncons (n : nat) (l : natlist).
4
5 Check natlist_ind :
6   forall P : natlist -> Prop,
7     P nnil  ->
8     (forall (n : nat) (l : natlist), P l -> P (ncons n l)) ->
9     forall l : natlist, P l.
```

일반적으로, 귀납적으로 정의된 타입 foo: Type의 생성자가 c1, .., cn일 때 foo_ind는 다음과 같은 형태를 가진다.

```
foo_ind :
  forall (P : foo -> Prop),
    case for c1 ->
    case for .. ->
    case for cn ->
    forall (x : foo), P x
```

여기서 case for ci는 생성자 ci에 대한 귀납가설이며 다음과 같은 형태를 가진다. ci의 인수들을 x1: a1, .., xk: ak라고 했을 때,

```
forall (x1:a1) ... (xk:ak),
P xj1 -> ... -> P xjm -> P (c x1 ... xk)
```

가 ci에 대한 귀납가설이다. (정확히 말하면 귀납가설은 P xj1 /\ ... /\ P xjm이고 이 가설을 이용하여 서브고울 P (c x1 ... xk)을 증명하면 된다.) 여기서 xj1, .., xjm은 ci의 인수들 중에서 타입이 foo인 것들이다. 즉 aj1 = foo, ..., ajm = foo이다.

위와 같은 기술은 실은 oversimplification이며 SF는 다음의 natlist': Type을 이 부분을 이해하기 위한 하나의 예로 제시한다. 이때의 귀납가설들은 natlist의 경우에서와 논리적으로 동등하지만 형식적으로는 다르다는 점을 주목하자.

```
10 Inductive natlist' : Type :=
11   | nnil'
12   | nsnoc (l : natlist') (n : nat).
13
14 Check natlist'_ind :
15   forall P : natlist' -> Prop,
16     P nnil' ->
17     (forall l : natlist', P l -> forall n : nat, P (nsnoc l n)) ->
18     forall n : natlist', P n.
```

라인 3과 라인 12의 차이로 인하여 귀납가설 라인 8과 라인 17의 차이가 발생하였다.

<u>연습문제</u> 2개.

```
1  Inductive booltree : Type :=
2    | bt_empty
3    | bt_leaf (b : bool)
4    | bt_branch (b : bool) (t1 t2 : booltree).
5
6  Definition booltree_property_type : Type := booltree -> Prop.
7
8  Definition base_case (P : booltree_property_type) : Prop :=
9    P bt_empty.
10
11 Definition leaf_case (P : booltree_property_type) : Prop :=
12   forall b: bool, P (bt_leaf b).
13
14 Definition branch_case (P : booltree_property_type) : Prop :=
15   forall (b: bool) (t1: booltree), P t1 ->
16     forall t2: booltree, P t2 ->
17     P (bt_branch b t1 t2).
18
19 Definition booltree_ind_type :=
20   forall (P : booltree_property_type),
21     base_case P ->
22     leaf_case P ->
23     branch_case P ->
24     forall (b : booltree), P b.
25
26 Theorem booltree_ind_type_correct : booltree_ind_type.
27 Proof. exact booltree_ind. Qed.
28
29 Check booltree_ind.
```

여기서 주의할 것은 라인 15–17을 다음과 같이 쓰면 안 된다는 것이다.

```
forall (b: bool) (t1 t2: booltree),
  P t1 -> P t2 -> P (bt_branch b t1 t2).
```

다음의 연습문제는 귀납 원리가 주어졌을 때 그것에 대응하는 귀납적 정의를 찾아내는 것이다. 귀납 원리가 다음과 같이 주어졌다고 하자.

```
forall P : Toy -> Prop,
  (forall b : bool, P (con1 b)) ->
```

```
    (forall (n : nat) (t : Toy), P t -> P (con2 n t)) ->
    forall t : Toy, P t
```

Toy의 귀납적 정의는 다음과 같이 하면 된다.

```
Inductive Toy : Type :=
| con1 (b : bool)
| con2 (n : nat) (t : Toy).
```

그리고 다음의 정리를 증명한다.

```
Theorem Toy_correct : exists (f: bool -> Toy) (g: nat -> Toy -> Toy),
forall P : Toy -> Prop,
  (forall b : bool, P (f b)) ->
  (forall (n : nat) (t : Toy), P t -> P (g n t)) ->
  forall t : Toy, P t.
Proof.
  exists con1. exists con2. (* exists con1 con2. doesn't work. *)
  exact Toy_ind.
Qed.
```

Polymorphism

귀납 원리는 `forall P: foo -> Prop, ...`의 형태를 가진다. 하지만 `foo`가 `bar X`와 같은 다형 타입일 때는 다음과 같은 형태를 가져야 할 것이다.

```
forall (X: Type) (P: bar X -> Prop), ...
```

<u>연습문제</u> 4개.

```
1  Inductive tree (X: Type) : Type :=
2    | leaf (x: X)
3    | node (t1 t2: tree X).
4
5  Fact tree_ind_my :
6    forall (X: Type) (P: tree X -> Prop),
7      (forall x: X, P (leaf X x)) ->
8      (forall (t1: tree X), P t1 ->
9        forall (t2: tree X), P t2 -> P (node X t1 t2)) ->
10     forall t: tree X, P t.
11 Proof.
12   exact tree_ind.
13 Qed.
```

이번에는 주어진 귀납 원리에 대응하는 다형 데이터 타입을 찾는 문제이다.

```
1  mytype_ind :
2    forall (X : Type) (P : mytype X -> Prop),
3        (forall x : X, P (constr1 X x)) ->
4        (forall n : nat, P (constr2 X n)) ->
5        (forall m : mytype X, P m ->
6           forall n : nat, P (constr3 X m n)) ->
7        forall m : mytype X, P m
```

다음과 같이 하면 된다.

```
Inductive mytype (X: Type) : Type :=
  | constr1 (x : X)
  | constr2 (n : nat)
  | constr3 (m : mytype X) (n : nat).

Check mytype_ind.
```

3번째 문제는 2번째와 비슷하다. 조금 더 복잡할 뿐이다.

```
1  foo_ind :
2    forall (X Y : Type) (P : foo X Y -> Prop),
3         (forall x : X, P (bar X Y x)) ->
4         (forall y : Y, P (baz X Y y)) ->
5         (forall f1 : nat -> foo X Y,
6           (forall n : nat, P (f1 n)) -> P (quux X Y f1)) ->
7         forall f2 : foo X Y, P f2
8
9  Inductive foo (X Y: Type) : Type :=
10 | bar (x : X)
11 | baz (y : Y)
12 | quux (f1 : nat -> foo X Y).
13
14 Check foo_ind.
```

마지막 문제.

```
1  Inductive foo' (X:Type) : Type :=
2    | C1 (l : list X) (f : foo' X)
3    | C2.
4
5  foo'_ind :
6    forall (X : Type) (P : foo' X -> Prop),
7         (forall (l : list X) (f : foo' X),
8               P f ->
9               P (C1 l f)) ->
10        P (C2 X) ->
11        forall f : foo' X, P f.
```

생성자 뒤에 타입을 두는 것을 잊지 말 것. 그리고 `P`는 1항 술어이므로 인수에 괄호가 필요한 경우가 많음을 잊지 말 것.

Induction Hypotheses

귀납적 증명에서 귀납 가설(*induction hypothesis*)이란 정확히 무엇인가?

그 증명에서 얻고자 하는 고울은 `forall x: foo, P x` 형태이다. 타입 `foo`의 생성자 `c`의 인수들 중에 타입이 `foo`인 것들을 `xj1, .., xjm`이라고 했을 때

```
P xj1 /\ ... /\ P xjm
```

가 생성자 `c`에 대한 귀납가설이 된다. 실제 증명세션에서는 귀납가설이 이러한 논리곱문 하나로 주어지지 않고 `m` 개의 논리곱 인자들이 가설로서 컨텍스트에 들어앉게 된다. 그리고 이것을 가설

로 삼아서 생성자 c에 대한 서브고울 P (c x1 ... xn)을 증명하면 된다. 이를 귀납 고울(*induction goal*)이라고 부르기로 하자.

xj1, .., xjm은 c x1 .. xn보다 이전에 생성된 것이다. 그러므로 P는 xj1, .., xjm 각각에 대해서 성립한다고 가정하는 것이다.

More on the induction Tactic

induction n을 사용할 때 n은 intros 되어 있어도 되고, 안 되어 있어도 된다. (전칭한정사에) 묶인 변수가 여러 개인 경우에는 이와 관련하여 유의할 부분이 있다.

intros 후에 induction을 하는 경우에는 모든 변수가 특정화 되었다가 귀납 변수만 다시 regeneralize 된다.

intros 없이 직접 induction n을 하는 경우에는 다시 두 경우로 나누어 생각해야 한다. n이 맨 처음 변수인 경우에는 나머지 변수들은 아직 묶여 있다. 그러나 n의 앞에 다른 묶인 변수들이 있는 경우에는 이 변수들은 자동으로 특정된다. 그리고 n 이후에 나타나는 묶인 변수들은induction n 후에도 묶인 상태로 남아 있게 된다.

Induction Principles for Propositions

타입을 Inductive를 써서 정의하면 자동으로 이에 대응하는 귀납 원리가 만들어진다고 하였다. 프랍을 Inductive를 써서 정의하는 경우에도 귀납 원리(*induction principle*)가 만들어 진다.

P: X -> Prop가 술어일 때 귀납에 의한 증명의 고울은 forall x: X, P x가 아니라, 귀납에 의하여 정의된 술어를 Q: X -> Prop이라고 했을 때, forall x: X, Q x -> P x이다.

그리고 생성자에 대한 귀납가설은 일단 간단한 경우에 대해서만 말하자면, 생성자가 c : forall x: X, Q x -> Q (f x)일 때 다음과 같다.

```
forall x: X, Q x -> P x -> P (f x)
```

왜 이렇게 되는지 숙고해 보라. (그리고 수학적으로 증명해 보라.)

ev: nat -> Prop을 예로 들어 보자.

```
1  Print ev.
2  (* Inductive ev : nat -> Prop :=
3    | ev_0 : ev 0
4    | ev_SS : forall n : nat, ev n -> ev (S (S n))) *)
5
6  Check ev_ind :
7    forall P : nat -> Prop,
8      P 0 ->
9      (forall n : nat, ev n -> P n -> P (S (S n))) ->
10     forall n : nat, ev n -> P n.
```

참고로 ev의 정의에서는 라인 4가 ev_SS (n: nat) (H: ev n): ev (S (S n))이었다. Print의 출력과 약간 다르다.

에비던스 ev n에 대한 induction은 apply ev_ind로 대체할 수 있다. 짝수임을 좀 어색한 방법으로 나타내는 술어 ev'을 다음과 같이 정의했을 때 이것이 ev와 동등함을 induction을 써서 증명한 적이 있는데, 여기서는 apply ev_ind를 써서 한 방향만 증명해 보자.

```
1  Inductive ev' : nat -> Prop :=
2    | ev'_0 : ev' 0
3    | ev'_2 : ev' 2
4    | ev'_sum n m (Hn: ev' n) (Hm: ev' m) : ev' (n + m).
5
6  Theorem ev_ev' : forall n : nat, ev n -> ev' n.
7  Proof.
8    apply ev_ind.
9    - (* ev_0 *)
10     apply ev'_0.
11   - (* ev_SS *)
12     intros m Hm IH.
13     apply (ev'_sum 2 m).
14     + apply ev'_2.
15     + apply IH.
16 Qed.
```

라인 8을 실행한 후에 Coq Goals 화면은 다음과 같다.

```
Goal 1
(1 / 2) ------------------------
ev' 0

Goal 2
(2 / 2) ------------------------
forall n : nat, ev n -> ev' n -> ev' (S (S n))
```

Goal 1에 대해서는 설명이 필요 없을 것이다. Goal 2를 증명하기 위하여 라인 12를 실행한 후에 Coq Goals 화면은 다음과 같다.

```
m : nat
Hm : ev m
IH : ev' m
(1 / 1) ------------------------
ev' (S (S m))
```

여기서 라인 13을 실행하면 고울 ev' (S (S m))은 ev'의 생성자들과 2가지 방법으로 매치될 수 있다. 하나는 라인 3의 생성자 ev'_2의 정의에 따라 S (S m)이 S (S 0)와 매치되는 것이고, 다른 하나는 라인 4의 생성자 ev'_sum의 정의에 따라 n + m과 매치되는 것이다. 후자의 경우 n := 2로 매치될 것이다. 따라서 라인 4에서 보듯이 2개의 서브고울 Hn과 Hm이 필요한데, 전자는 ev' 2이므로 생성자 ev'_2에 의하여 자동으로 해결되고[1], 후자는 ev' m이다.

따라서 Coq Goals 화면은 다음과 같게 된다.

```
Goal 1
m : nat
Hm : ev m
IH : ev' m
(1 / 2) ------------------------
ev' 2
```

[1] Coq의 이런 거동이 맘에 들지 않는다.

```
Goal 2
m : nat
Hm : ev m
IH : ev' m
(2 / 2) ------------------------
ev' m
```

그 다음은 생략한다.

술어의 귀납적 정의는 내용적으로 동등해도 형식적으로 다르면 대응하는 귀납 원리도 달라진다. 이것의 설명을 위하여 SF에서 제시하는 예는 2항 술어, 즉 관계이다. 그래서 이 내용은 여기가 아니라 다음 장, Properties of Relations에 옮겨 두었다.

Explicit Proof Objects for Induction (Optional)

귀납 원리는 정리라고 하였다. 따라서 그것의 증거항은 `Print foo_ind.` 명령으로 알 수 있다.

```
Print nat_ind. (* nat_ind =
fun (P : nat -> Prop) (f : P 0)
  (f0 : forall n : nat, P n -> P (S n)) =>
  fix F (n : nat) : P n := match n as n0 return (P n0) with
  | 0 => f
  | S n0 => f0 n0 (F n0)
end
: forall P : nat -> Prop,
  P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
*)
```

증명하고자 하는 명제의 증거항을 내가 직접 찾는 것은, 그리 쉽지 않지만, 어떤 종류의 귀납적 증명에서는 유용하게 쓰일 수 있다. 다음의 예를 보라.

```
Lemma even_ev : forall n: nat, even n = true -> ev n.
Proof.
  induction n.
  - intros. apply ev_0.
  - intros. destruct n.
    + simpl in H. discriminate H.
    + simpl in H.
      apply ev_SS.
Abort.
```

의외로 증명이 잘 되지 않는다. 왜냐하면 `induction n`은 `P n`과 `P (S n)`을 이어주는 것인데 우리가 원하는 것은 `ev n`과 `ev (S (S n))`이기 때문이다.[2]

even_ev의 증명에는 다음과 같은 방법이 있기는 하다.

```
Lemma even_ev' : forall n: nat,
  (even n = true -> ev n) /\ (even (S n) = true -> ev (S n)).
Proof.
  induction n.
  - split.
```

[2]참고로 `ev_even : forall n: nat, ev n -> even n = true`는 에비던스에 대한 귀납을 써서 쉽게 증명된다.

```
    + intros. apply ev_0.
    + intros. discriminate H.
  - destruct IHn as [IHn0 IHn1].
    split.
    + intros. apply IHn1. exact H.
    + intros. simpl in H.  apply IHn0 in H.
      apply ev_SS. exact H.
Qed.
```

여기까지 한 다음에는, even_ev는 apply even_ev'를 써서 간단히 증명된다.

더 좋은 방법이 있다.

```
1  Definition nat_ind2 :
2    forall (P : nat -> Prop),
3    P 0 ->
4    P 1 ->
5    (forall n : nat, P n -> P (S(S n))) ->
6    forall n : nat , P n :=
7      fun P => fun P0 => fun P1 => fun PSS =>
8        fix f (n: nat) := match n with
9          0 => P0
10       | 1 => P1
11       | S (S n') => PSS n' (f n')
12       end.
13
14 Lemma even_ev : forall n, even n = true -> ev n.
15 Proof.
16   intros.
17   induction n as [ | |n'] using nat_ind2.
18   - apply ev_0.
19   - simpl in H.
20     discriminate H.
21   - simpl in H.
22     apply ev_SS.
23     apply IHn'.
24     apply H.
25 Qed.
```

라인 17의 induction ... using은 이러한 비표준 귀납에 사용하는 구문이다. nat_ind2의 증거항을 찾는 것은 처음에는 상당히 어렵지만 조금 익숙해지면 그럭저럭 가능하다. 오히려 nat_ind2의 증명 스크립트를 찾는 것은 더 어려운 것 같다.[3]

통상적인 자연수에 대한 귀납 원리의 증거항은 다음과 같이 찾으면 된다.

```
1  Fixpoint build_proof
2           (P : nat -> Prop)
3           (evPO : P 0)
4           (evPS : forall n : nat, P n -> P (S n))
5           (n : nat) : P n :=
6    match n with
7    | 0 => evPO
```

[3] 못 찾겠다. nat_ind2뿐만 아니라 nat_ind의 증명 스크립트도 찾지 못했다.

```
8    | S k => evPS k (build_proof P evP0 evPS k)
9    end.
10
11 Definition nat_ind_tidy := build_proof.
12
13 Check build_proof.
14 Check nat_ind_tidy. (* forall P : nat -> Prop,
15   P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n *)
16 Print build_proof.
17 Print nat_ind_tidy.
```

11

Properties of Relations

Relations

술어predicate가 1개의 인수를 받아 프랍을 리턴하는 함수라고 한다면, 관계(*relation*)은 2개의 인수를 받아 프랍을 리턴하는 함수라고 할 수 있다. 더 일반적인 시각에서는, 관계는 $n \geqslant 0$개의 인수를 받아 프랍을 리턴하는 함수이며, $n = 0$일 때는 프랍, $n = 1$일 때는 술어, $n = 2$일 때는 관계, 그 이상은 3항관계, 4항관계 등으로 부른다.

가장 일반적인 의미에서는 관계는 각 인수들이 서로 다른 타입을 가질 수 있으나, Coq에서는 관계의 인수들은 고정된 타입을 가지는 것으로 가정한다.

```
Definition relation (X: Type) := X -> X -> Prop.
```

예를 들어 nat 위의 관계 le는 다음과 같다.

```
1  Print le.
2  (*  Inductive le (n : nat) : nat -> Prop :=
3          le_n : n <= n
4        | le_S : forall m : nat, n <= m -> n <= S m *)
5  Check le : nat -> nat -> Prop.
6  Check le : relation nat.
```

라인 5와 라인 6은 흥미롭다. 라인 4에서 콜론의 전후에 인수를 각각 하나씩 둔 이유는 귀납을 편리하게 사용하기 위함이다. 조금 후에 이와 관련한 설명이 나올 것이다.

Partial Functions

관계의 성질중 하나로 부분함수partial function가 있다.

```
Definition partial_function {X: Type} (R: relation X) : Prop :=
  forall x y1 y2 : X, R x y1 -> R x y2 -> y1 = y2.
```

관계 $(n, n+1)$는 부분함수이며(실은 전함수total function임.) 다음과 같이 정의할 수 있다.

```
Inductive next_nat : nat -> nat -> Prop :=
  | nn n : next_nat n (S n).

Check next_nat. (* nat -> nat -> Prop *)
Check next_nat : relation nat.
```

이제 next_nat가 부분함수임을 증명해 보자.

```
1  Theorem next_nat_partial_function :
2    partial_function next_nat.
3  Proof.
4    unfold partial_function.
5    intros x y1 y2 H1 H2.
6    inversion H1. inversion H2.
7    reflexivity.  Qed.
```

라인 5를 실행한 후의 Coq Goals 화면은 다음과 같다.

```
x, y1, y2 : nat
H1 : next_nat x y1
H2 : next_nat x y2
(1 / 1) ------------------------
y1 = y2
```

라인 6의 inversion H1을 실행한 후의 화면은 다음과 같다. 더 이상의 설명은 생략한다.

```
...
n : nat
H : n = x
H0 : S x = y1
(1 / 1)
S x = y2
```

le는 부분함수가 아니다. 반례로는 0 <= 0, 0 <= 1을 들 수 있다. 이 반례를 사용하여 le가 부분함수가 아님을 보이는 Coq 증명을 만들어 보자.

```
1  Theorem le_not_a_partial_function :
2    ~ (partial_function le).
3  Proof.
4    unfold not. unfold partial_function. intros Hc.
5    assert (0 = 1) as Nonsense. {
6    (* assert (Nonsense: 0 = 1). { *)
7      apply Hc with (x := 0).
8      - apply le_n.
9      - apply le_S. apply le_n. }
10   discriminate Nonsense.   Qed.
```

라인 5는 라인 6과 동등하다. 라인 5를 실행한 후의 Coq Goals 화면은 다음과 같다.

```
Goal 1
Hc : forall x y1 y2 : nat, x <= y1 -> x <= y2 -> y1 = y2
(1 / 2) ------------------------
0 = 1

Goal 2
Hc : forall x y1 y2 : nat, x <= y1 -> x <= y2 -> y1 = y2
Nonsense : 0 = 1
(2 / 2) ------------------------
False
```

Goal 2는 간단하게 `discriminate Nonsense`로 증명된다.

Goal 1은 라인 7에 의하여 다시 2개의 subgoal로 나뉜다.

```
Goal 1
Hc : forall x y1 y2 : nat, x <= y1 -> x <= y2 -> y1 = y2
(1 / 2) ------------------------
0 <= 0

Goal 2
Hc : forall x y1 y2 : nat, x <= y1 -> x <= y2 -> y1 = y2
(2 / 2) ------------------------
0 <= 1
```

다음은 자연수 위의 total relation $\mathbb{N} \times \mathbb{N}$을 정의하고 이것이 부분함수가 아님을 증명하는 문제이다.

```
1  Inductive total_relation : nat -> nat -> Prop :=
2    tot (n m: nat) : total_relation n m.
3
4  Theorem total_relation_not_partial_function :
5    ~ (partial_function total_relation).
6  Proof.
7    unfold not. unfold partial_function. intros Hc.
8    assert (Nonsense: 0 = 1). {
9      apply Hc with (x := 0).
10     - apply tot.
11     - apply tot. }
12   discriminate Nonsense.   Qed.
```

이번에는 자연수 위의 공관계 $\varnothing$를 정의하고 이것이 부분함수임을 증명하는 문제이다.

```
1  Inductive empty_relation : nat -> nat -> Prop :=
2     empty (n m: nat) (H: n <> n) : empty_relation n m.
3
4  Theorem empty_relation_partial_function :
5    partial_function empty_relation.
6  Proof.
7    unfold partial_function. intros x y1 y2 H1 H2.
8    inversion H1.
9    unfold not in H. exfalso. apply H. reflexivity.
10   Qed.
```

라인 8을 실행한 후의 `Coq Goals` 화면은 다음과 같다.

```
x, y1, y2 : nat
H1 : empty_relation x y1
H2 : empty_relation x y2
n, m : nat
H : x <> x
H0 : n = x
H3 : m = y1
(1 / 1) ------------------------
y1 = y2
```

`H : x <> x`를 보면 라인 9 이후의 코드를 쉽게 이해할 수 있을 것이다.

Reflexive Relations, Transitive Relations

```
Definition reflexive {X: Type} (R: relation X) : Prop :=
  forall a : X, R a a.

Theorem le_reflexive :
  reflexive le.
Proof.
  unfold reflexive. intros n. apply le_n.  Qed.
```

다음 정리의 증명은 중요하다. 라인 8의 induction이 정확히 어떤 역할을 하는지 잘 생각해 보아야 한다.

```
1  Definition transitive {X: Type} (R: relation X) : Prop :=
2    forall a b c : X, (R a b) -> (R b c) -> (R a c).
3
4  Theorem le_trans :
5    transitive le.
6  Proof.
7    unfold transitive. intros n m o Hnm Hmo.
8    induction Hmo as [|m' Hmo'].
9    - (* le_n, o = m *)
10     exact Hnm.
11   - (* le_S, m = m', o = S m' *)
12     apply le_S. exact IHHmo'.  Qed.
```

induction 직전의 화면이다.

```
n, m, o : nat
Hnm : n <= m
Hmo : m <= o
(1 / 1) ------------------------
n <= o
```

여기서 induction Hmo를 실행하면 Hmo: m <= o의 에비던스는 다음 둘 중 어느 하나이다.

① le_n : n <= n이 m <= o에 매치된다. 따라서 o = m이어야 하고 고울에 나타난 o는 m으로 바꿔져서 고울 n <= o가 n <= m으로 바뀐다.

② le_S : forall (m: nat), n <= m -> n <= S m. 어떤 m'에 대해서 n <= S m'이 m <= o에 매치된다. 고로 o = S m'이므로 고울에 나타난 o는 S m'으로 바꿔져서 고울 n <= o가 n <= S m'으로 바뀐다.

induction 후의 화면이다.

```
Goal 1
n, m : nat
Hnm : n <= m
(1 / 2) ------------------------
n <= m

Goal 2
n, m : nat
```

```
Hnm : n <= m
m' : nat
Hmo' : m <= m'
IHHmo' : n <= m'
(2 / 2) ________________________
n <= S m'
```

`Hml'`와 `IHHmo`'을 설명하지 못하겠다. ㅠㅠ

`induction` 책략이 아니라 귀납 원리를 사용하는 증명을 보이겠다. 이건 더 명료하게 설명할 수 있다.

```
1  Check le_ind :
2    forall (n : nat) (P : nat -> Prop),
3      P n ->
4      (forall m : nat, n <= m -> P m -> P (S m)) ->
5      forall n0 : nat, n <= n0 -> P n0.
6
7  Fact le_trans_le_ind :
8      transitive le.
9    Proof.
10     unfold transitive. intros a b c Hab.
11     generalize dependent c.
12     apply le_ind with (n := b) (P := fun x => a <= x).
13     - exact Hab.
14     - intros c Hbc Hac.
15       apply le_S. exact Hac.
16   Qed.
```

라인 11을 실행한 후의 `Coq Goals` 화면은 다음과 같다.

```
a, b : nat
Hab : a <= b
(1 / 1) ________________________
forall c : nat, b <= c -> a <= c
```

이렇게 만들어 놓은 이유는 `apply le_ind`를 사용하기 위해서이다. 라인 12을 실행한 후의 `Coq Goals` 화면은 다음과 같다.

```
Goal 1
a, b : nat
Hab : a <= b
(1 / 2) ________________________
a <= b

Goal 2
a, b : nat
Hab : a <= b
(2 / 2) ________________________
forall m : nat, b <= m -> a <= m -> a <= S m
```

Goal 1은 라인 3이며 아주 쉽게 해결된다.

Goal 2는 라인 4이며 라인 14의 `intros c Hbc Hac`를 실행한 후의 `Coq Goals` 화면은 다음과 같다. 그 다음은 쉬우므로 생략한다.

```
a, b : nat
Hab : a <= b
c : nat
Hbc : b <= c
Hac : a <= c
(1 / 1) ________________________
a <= S c
```

여기서 잠시, 관계의 귀납적 정의는 내용적으로 동등해도 형식적으로 다르면 대응하는 귀납 원리도 달라진다는 점을 짚고 넘어가겠다. 이를 보여주는 예로, 자연수에서의 작거나 같음을 나타내는 술어에 대한 귀납적 정의를 다음과 같이 두 방법으로 만들었다고 하자.

```
Inductive le1 : nat -> nat -> Prop :=
  | le1_n : forall n, le1 n n
  | le1_S : forall n m, (le1 n m) -> (le1 n (S m)).

Notation "m <=1 n" := (le1 m n) (at level 70).

Inductive le2 (n:nat) : nat -> Prop :=
  | le2_n : le2 n n
  | le2_S m (H : le2 n m) : le2 n (S m).

Notation "m <=2 n" := (le2 m n) (at level 70).
```

각 술어에 대한 귀납 원리는 다음과 같다.

```
Check le1_ind :
  forall P : nat -> nat -> Prop,
    (forall n : nat, P n n) ->
    (forall n m : nat, n <=1 m -> P n m -> P n (S m)) ->
    forall n n0 : nat, n <=1 n0 -> P n n0.

Check le2_ind :
  forall (n : nat) (P : nat -> Prop),
    P n ->
    (forall m : nat, n <=2 m -> P m -> P (S m)) ->
    forall n0 : nat, n <=2 n0 -> P n0.
```

두 번째 정의 le2는 지금 우리가 사용하는 le와 조금 다르지만 귀납 원리는 두 정의에서 동일하다. 즉, 라인 10과 라인 11은 거기에 나타나는 <=2를 <=로 바꾸어도 성립한다.

첫 번째 정의보다 이것이 더 낫다. 그 이유는 귀납 원리의 결론이 1개의 묶인 변수만을 가지는, 더 간단한 형태이기 때문이다. 귀납으로 증명하고자 하는 프랍은 1개의 묶인 변수만을 가지는 것이 대부분이고, 또한 이것이 바람직하기 때문에 귀납 원리의 결론도 여기에 맞춰 주는 것이 좋다.

lt의 추이성은 정리 le_trans를 사용하여 쉽게 증명된다.

```
Theorem lt_trans:
  transitive lt.
Proof.
  unfold transitive. unfold lt.
  intros n m o Hnm Hmo.
```

```
6    apply le_S in Hnm.
7    apply le_trans with (a := (S n)) (b := (S m)) (c := o).
8    apply Hnm.
9    apply Hmo. Qed.
```

lt의 추이성을 증명할 때 le_trans를 사용하지 않고 m < o의 에비던스에 대한 귀납을 사용하여 증명할 수 있다.

```
1  Theorem lt_trans_hard_way :
2    transitive lt.
3  Proof.
4    (* Prove this by induction on evidence that m is less than o. *)
5    unfold lt. unfold transitive.
6    intros n m o Hnm Hmo.
7    induction Hmo as [| m' Hmo'].
8    - (* le_n, S m = o *)
9      apply le_S. exact Hnm.
10   - (* le_S, *)
11     apply le_S. exact IHHmo'.
12 Qed.
```

o: nat에 대한 귀납을 사용해서 해결할 수도 있다.

```
Theorem lt_trans'' :
  transitive lt.
Proof.
  unfold lt. unfold transitive.
  intros n m o Hnm Hmo.
  induction o.
  - (* o = 0 *) inversion Hmo.
  - (* o = S o' *) inversion Hmo.
    + (* m = o *) rewrite <- H0. apply le_S. exact Hnm.
    + (* S m <= o *) apply le_S. apply IHo. exact H0.
Qed.
```

간단하고 유용한 정리다.

```
Theorem le_Sn_le : forall n m, S n <= m -> n <= m.
Proof.
  intros n m H. apply le_trans with (b := S n).
  - apply le_S. apply le_n.
  - exact H.
Qed.
```

연습문제.

```
Theorem le_S_n : forall n m,
  (S n <= S m) -> (n <= m).
Proof.
  intros. inversion H.
  - (* le_n *) apply le_n.
  - (* le_S *) apply le_Sn_le. exact H1.
Qed.
```

연습문제.

```
Theorem le_Sn_n : forall n,
  ~ (S n <= n).
Proof.
  intros n. unfold not. intros H.
  induction n.
  - (* n = 0 *)
    inversion H.
  - (* n = S n' *)
    apply IHn. apply le_S_n. exact H.
Qed.
```

Symmetric and Antisymmetric Relations

```
Definition symmetric {X: Type} (R: relation X) :=
  forall a b : X, (R a b) -> (R b a).
```

연습문제

```
1  Theorem le_not_symmetric :
2    ~ (symmetric le).
3  Proof.
4    unfold not. unfold symmetric. intros H.
5    assert (1 <= 0) as Nonsense. {
6      apply H with (a := 0) (b := 1).
7      apply le_S. apply le_n.
8    }
9    inversion Nonsense.
10 Qed.
```

```
Definition antisymmetric {X: Type} (R: relation X) :=
  forall a b : X, (R a b) -> (R b a) -> a = b.
```

연습문제

```
1  Theorem le_antisymmetric :
2    antisymmetric le.
3  Proof.
4    unfold antisymmetric. induction a as [| a'].
5    - (* a = 0 *)
6      intros. inversion H0. reflexivity.
7    - (* a = S a' *)
8      intros. destruct b.
9      + (* b = 0 *)
10       inversion H.
11     + (* b = S b' *)
12       apply f_equal. apply IHa'.
13        apply le_S_n. exact H.
14        apply le_S_n. exact H0.
15 Qed.
16
17 Theorem le_step : forall n m p,
```

```
18    n < m ->
19    m <= S p ->
20    n <= p.
21  Proof.
22    intros. unfold lt in H.
23    apply le_S_n.
24    apply le_trans with (b := m).
25    - exact H.
26    - exact H0.
27  Qed.
```

Equivalence relation, partial order, preorder.

```
Definition equivalence {X:Type} (R: relation X) :=
  (reflexive R) /\ (symmetric R) /\ (transitive R).

Definition order {X:Type} (R: relation X) :=
  (reflexive R) /\ (antisymmetric R) /\ (transitive R).

Definition preorder {X:Type} (R: relation X) :=
  (reflexive R) /\ (transitive R).
```

Reflexive Transitive Closure

```
Inductive clos_refl_trans {A: Type} (R: relation A) : relation A :=
  | rt_step x y (H : R x y) : clos_refl_trans R x y
  | rt_refl x : clos_refl_trans R x x
  | rt_trans x y z
        (Hxy : clos_refl_trans R x y)
        (Hyz : clos_refl_trans R y z) :
        clos_refl_trans R x z.
```

next_nat 관계의 반사추이 폐포가 le임을 증명한다.

```
1   Theorem next_nat_closure_is_le : forall n m,
2     (n <= m) <-> ((clos_refl_trans next_nat) n m).
3   Proof.
4     intros n m. split.
5     - (* -> *)
6       intro H. induction H as [| m' H'].
7       + (* H = le_n *) apply rt_refl with (x:= n).
8       + (* H = le_S *)
9         apply rt_trans with m'. apply IHH'. apply rt_step.
10        apply nn.
11    - (* <- *)
12      intro H. induction H.
13      + (* rt_step *) inversion H. apply le_S. apply le_n.
14      + (* rt_refl *) apply le_n.
15      + (* rt_trans *)
16        apply le_trans with y.
17        apply IHclos_refl_trans1.
18        apply IHclos_refl_trans2. Qed.
```

이 증명을 통하여 le의 에비던스에 대한 귀납에 대해서 철저히 설명해 보겠다. 라인 6의 induction H 직전의 화면은 다음과 같다.

```
n, m : nat
H : n <= m
(1 / 1) ------------------------
clos_refl_trans next_nat n m
```

induction H를 실행하면 다음과 같이 된다.

```
Goal 1
n : nat
(1 / 2) ---------------------
clos_refl_trans next_nat n n

Goal 2
n, m : nat
H' : n <= m'
IHH' : clos_refl_trans next_nat n m'
(2 / 2) ---------------------
clos_refl_trans next_nat n (S m')
```

Goal 1은 H: n <= m이 le_n : n <= n에 의하여 얻어진 경우이다. 따라서 n = m이어야 한다. 이에 따라 Goal 프랍이 clos_refl .. n n로 바뀐다. 여기서 주의할 것은 고울 프랍이 G(n,m)이라면 이것이 조건 n = m에 의하여 G(n,n)으로 바뀌는 것인지 아니면 G(m,m)으로 바뀌는 것인지를 잘 판단해야 한다는 것이다.

> 가설이 H(n,m)이면 고울은 G(n,n)으로 바뀌고, 가설이 H(m,n)이면 고울은 G(m,n)으로 바뀐다.

바뀌는 기호 m은 고울 프랍뿐만 아니라 H를 제외한 모든 가설들에 나타나는 경우에도 n으로 바꿔 주어야 한다. 이 경우에는 H 외에는 가설이 없으므로 H만 제거해 주면 된다.

여기서 주의할 것이 있다. 생성자에서 사용하는 변수 기호와 가설 및 고울에서 사용하는 변수 기호를 혼동하지 않아야 한다. 두 그룹에 나타나는 변수 기호에 공통되는 것이 있는 경우 혼동이 있을 수 있다. 다음의 원칙 하나만 알면 혼동을 피할 수 있다.

> 생성자에서 사용하는 변수는 묶인 변수이다. 그러므로 임의로 다른 기호로 바꾸어 생각해도 된다. 람다 캘큘러스에서의 α-변환과 같다고 보면 될 것이다.

Goal 2는 H: n <= m이 le_S : forall x, x <= y -> x <= S y에 의하여 얻어진 경우이다. x <= S y가 n <= m에 매치되므로 x := n, y := m', S m' = m이 되어야 한다. 자유변수 y를 특정할 때 사용할 변수 기호를 m'로 지정하여 induction H as [| m' H']를 실행한 것에 주목하라. 이제 고울 프랍이 clos_ .. n (S m')이 된 것이 설명되었다.

조건문의 전건 x <= y는 n <= m'이 되고 이것이 컨텍스트에 가설로 들어가야 한다. 이 가설의 이름을 H'으로 준다. (사실은 이 가설의 에비던스 H'을 컨텍스트에 넣는 것이다.)

마지막으로 귀납 가설이다. H(n,m)일 때 G(n,m)을 증명하는 것이 우리의 목표인데, 이것의 전 단계인 H'(n,m')일 때는 이에 대응하는 G(n,m')이 성립함을 가정한 것이 바로 귀납 가설이다. 귀납 가설의 이름은 통상 IH로 시작하며 H'가 성립할 때의 고울 프랍이므로 IHH'라는 이름을 Coq이 자동으로 주게 된다. 실은 귀납 가설의 이름도 다음과 같이 임의로 줄 수도 있었다.

```
induction H as [| m' H' IHmy].
```

이하의 증명은 생략한다.

정의 clos_refl_trans는 수학적으로는 옳지만 증명에 사용하기에는 때때로 불편하다. 이것보다 더 나은, 수학적으로 동등한 정의를 다음과 같이 만들어 보자. 이 정의는 관계를 입력받아서 관계를 만드는 것이 아니라 관계 인수에 1개의 인수를 추가로 받아서 1항 술어를 만든다. (결국 관계 인수에 더하여 2개의 인수를 받은 셈이 된다.)

```
Inductive clos_refl_trans_1n {A : Type}
                             (R : relation A) (x : A)
                             : A -> Prop :=
  | rt1n_refl : clos_refl_trans_1n R x x
  | rt1n_trans (y z : A)
      (Hxy : R x y) (Hrest : clos_refl_trans_1n R y z) :
      clos_refl_trans_1n R x z.
```

이 정의는 생성자를 3개 사용했던 이전의 정의와는 달리 생성자를 2개만 사용한다.

두 정의가 동등함을 증명하기에 앞서 후자가 만드는 2항 술어는 원래의 관계를 포함하고 (확장성extensionality) 또한 추이적이라는 것을 각각 보조정리를 통하여 증명한다.

```
1 Lemma rsc_R : forall (X:Type) (R:relation X) (x y : X),
2   R x y -> clos_refl_trans_1n R x y.
3 Proof.
4   intros X R x y H.
5   apply rt1n_trans with (y := y).
6   - apply H.
7   - apply rt1n_refl.   Qed.
```

라인 4를 실행한 후에 Coq Goals 화면은 다음과 같다.

```
X : Type
R : relation X
x, y : X
H : R x y
(1 / 1) ----------------
clos_refl_trans_1n R x y
```

그 다음 apply rt1n_trans를 실행하는데, 노파심에서 다음을 확인하고 넘어가기로 한다.

```
Check @rt1n_trans.
(** forall (A : Type) (R : relation A) (x y z : A),
    R x y -> clos_refl_trans_1n R y z -> clos_refl_trans_1n R x z *)
```

고울 프랍과 맞추어 보면 x := x, z := y가 되어야 함을 알 수 있다. 또 하나의 묶인 변수 y는 그대로 y := y로 두면 2개의 서브고울

```
1 R x y
2 clos_refl_trans_1n R y y
```

가 생성된다. 컨텍스트는 바뀔 이유가 없어 그대로다. 라인 1은 apply H로 해결되고 라인 2는 apply rt1n_refl로 해결된다. ✓

이상으로 확장성을 해결했고, 이제 추이성이다.

```
3 Lemma rsc_trans :
4   forall (X: Type) (R: relation X) (x y z : X),
5       clos_refl_trans_1n R x y  ->
6       clos_refl_trans_1n R y z ->
7       clos_refl_trans_1n R x z.
8 Proof.
9   intros X R x y z H1 H2.
10  induction H1.
11  - (* rt1n_refl *)
12    exact H2.
13  - (* rt1n_trans *)
14    apply rt1n_trans with (y := y).
15    + exact Hxy.
16    + apply IHclos_refl_trans_1n. exact H2.
17 Qed.
```

두 정의가 동등함을 다음 정리에서 증명한다.

```
1 Theorem rtc_rsc_coincide :
2   forall (X:Type) (R: relation X) (x y : X),
3     clos_refl_trans R x y <-> clos_refl_trans_1n R x y.
4 Proof.
5   intros X R x y. split.
6   - (* -> *)
7     intro H. induction H.
8     + (* rt_step *) apply rsc_R. exact H.
9     + (* rt_refl *) apply rt1n_refl.
10    + (* rt_trans *) apply rsc_trans with y.
11      { exact IHclos_refl_trans1. }
12      { exact IHclos_refl_trans2. }
13  - (* <- *)
14    intros H. induction H.
15    + (* rt1n_refl *) apply rt_refl.
16    + (* rt1n_trans *) apply rt_trans with y.
17      { apply rt_step. exact Hxy. }
18      { exact IHclos_refl_trans_1n. }
19 Qed.
```

12

Total and Partial Maps

전에 다음과 같이 다형성 함수 map을 정의했었다.

```
Fixpoint map {X Y: Type} (f: X -> Y) (l: list X) : list Y :=
  match l with
  | [] => []
  | h :: t => (f h) :: (map f t)
  end.
```

이 장에서 말하는 *map*은 이렇게 정의된 map 함수가 아니라, 타 언어에서 흔히 dictionary, associative array, hash 등으로 불리는 자료구조로, 키$_{\text{key}}$와 값$_{\text{value}}$의 쌍을 다루는 자료구조이다. 이 토픽에 대해서 §4.3에서 간단히 다룬 적이 있다.

The Coq Standard Library

지금까지 우리는 다른 곳에서 정의해 놓은 정의와 정리들을 사용하기 위하여 이전 장을 불러와서 사용해 왔다. 이제부터는 이렇게 하지 않고 Coq의 표준 라이브러리에서 불러올 것이다.

```
From Coq Require Import Arith.Arith.
From Coq Require Import Bool.Bool.
Require Export Coq.Strings.String.
From Coq Require Import Logic.FunctionalExtensionality.
From Coq Require Import Lists.List.
Import ListNotations.
Set Default Goal Selector "!".
```

다음 사이트를 참고할 것: https://coq.inria.fr/doc/V8.18.0/stdlib/.

맵을 사용하기 위하여는 우선 키$_{\text{key}}$로 사용할 타입을 정해야 한다. 우리는 키로 항상 문자열을 사용하기로 한다.

이 장에서 다음 정리가 유용하게 사용된다. 접두 String.은 생략해도 된다.

```
Check String.eqb_spec :
  forall x y : string, reflect (x = y) (String.eqb x y).
```

여기서 reflect: Prop -> bool -> Prop는 reflection에 의한 증명에 유용하게 사용되는 2항 술어이며 §8.5에서 공부하였다. [팩트 8.1]을 보라.

String.eqb는 문자열이 같은지를 비교하는 부울값 함수이다. 이 함수는 Coq.Strings 라이브러리에 정의되어 있다. =?는 nat 위의 등식 관계를 나타내는 부울값 함수인 것으로 배웠는데 이 기호는 string 위의 등식 관계를 나타내기도 한다.

```
Locate "_ =? _".
(** Notation "x =? y" := (Nat.eqb x y) : nat_scope (default interpretation)
    Notation "x =? y" := (eqb x y) : string_scope *)
```

이 표기법에 의하여 eqb_spec은

```
forall x y : string, reflect (x = y) (x =? y).
```

로 볼 수 있다.

String.eqb를 사용하는 기본적인 정리 3개를 아래 보였다. 이 정리들의 Nat 버전은 이전에 증명한 바 있다.

```
Check String.eqb_refl :
  forall x : string, (x =? x)%string = true.
Check String.eqb_eq :
  forall x y : string, (x =? y)%string = true <-> x = y.
Check String.eqb_neq :
  forall x y : string, (x =? y)%string = false <-> x <> y.
```

Total Maps

전에 §4.3에서 partial map을 조금 공부했었는데 그때는 키로 자연수를 사용했고, 유한한 개수의 키에 대해서만 값이 정의되어 있는 부분맵만을 다루었다. 그리고 맵을 key-value pair의 리스트로 구현했었다. 이번에는 먼저 total map을 공부하고 그 다음에 partial map을 다룰 것이다. 그리고 맵은 문자열의 집합에서 특정한 타입으로 가는 함수로 구현할 것이다. 즉, 키는 문자열이고 값은 어떤 타입이든 될 수 있다.

먼저 total_map이라는 타입을 정의한다. 이 타입의 원소는 전맵(*total map*)이라고 부르기로 한다.

```
Definition total_map (A: Type) := string -> A.
```

이 타입의 원소는, 즉 전맵은 값을 주지 않은 키를 받으면 디폴트 값을 리턴하도록 구현될 것이다. 이는 전함수total function로 만들기 위함이다.

제일 먼저, 가장 간단한 전맵인 *empty map*을 정의한다. 모든 입력에 대하여 디폴트 값을 리턴하는 함수이다. 디폴트 값은 인수로 받는다.

```
Definition t_empty {A: Type} (v: A) : total_map A :=
  (fun _ => v).
```

예를 들어 t_empty 3는 모든 문자열을 받아 3을 리턴하는 함수이다. 그런데 다음과 같이 하면 에러가 난다.

```
Compute t_empty 3 "foo". (* Error *)
```

왜냐하면 Coq은 "foo"를 아직 문자열로 인식하지 못하기 때문이다. 다음과 같이 하여 해결한다.

```
Fail Check "foo".
Open Scope string_scope.
Check "foo". (* = "foo": string *)
Compute t_empty 3 "foo". (* = 3: nat *)
```

맵을 업뎃하는 함수를 다음과 같이 정의한다. 이것은 기존의 맵과 거의 동일하며, 단 하나의 입력값 x에 대해서만 값이 v로 재설정 된 함수이다.

```
Definition t_update {A: Type} (m: total_map A) (x: string) (v: A) :=
  fun x' => if String.eqb x x' then v else m x'.
```

이 간단한 두 함수들을 사용하여 전맵을 하나 만들어 보자. 키가 "foo"이거나 "bar"인 경우에는 true를 리턴하고 그 밖의 모든 키(즉, 문자열)에 대해서는 false를 리턴하는 맵이다. 참고로 여기서는 Open Scope string_scope를 필요로 하지 않는다. 그 이유는 t_update의 두 번째 인수는 string 타입인 것을 Coq이 알고 있기 때문이다.

```
Definition examplemap :=
  t_update (t_update (t_empty false) "foo" true) "bar" true.
```

이 맵을 사용하여 다음과 같은 계산을 할 수 있다.

```
Compute (examplemap "foo", examplemap "bar", examplemap "baz").
(** = (true, true, false): bool * bool * bool *)
```

맵을 편리하게 사용하기 위한 표기법 2개를 다음과 같이 정의한다.[1]

```
1  Notation "'_' '!->' v" := (t_empty v)
2    (at level 100, right associativity).
3
4  Example example_empty := (_ !-> false).
5
6  Notation "x '!->' v ';' m" :=
7      (t_update m x v)
8      (at level 100, v at next level, right
9      associativity).
10
11 Definition examplemap' :=
12   ( "bar" !-> true;
13     "foo" !-> true;
14     _     !-> false
15   ).
```

이렇게 맵을 (순서쌍의 리스트가 아니라) 함수로 구현하면, §4.3에서는 별도의 find 함수를 만들어야 했지만 이제는 그럴 필요가 없다는 장점이 있다.

첫 예로 다음의 간단한 3 개의 문제를 보자.

[1]라인 4의 Example은 Definition으로 써도 된다. 라인 4의 구문은 Example로 정리를 정의할 때의 구문과 어떤 차이가 있는지 알겠는가?

```
Lemma t_apply_empty : forall (A : Type) (x : string) (v : A),
  (_ !-> v) x = v.
Proof.
  intros. reflexivity. Qed.
```

다음 예에서는 unfold를 사용해야만 계산이 진행됨에 유의하라.

```
1  Lemma t_update_eq : forall (A : Type) (m : total_map A) (x: string) (v: A),
2    (x !-> v ; m) x = v.
3  Proof.
4    intros. unfold t_update.
5    rewrite eqb_refl. reflexivity.
6  Qed.
7
8  Theorem t_update_neq :
9    forall (A : Type) (m : total_map A) (x1 x2: string) (v: A),
10     x1 <> x2 ->
11     (x1 !-> v ; m) x2 = m x2.
12 Proof.
13   intros. unfold t_update.
14   rewrite <- eqb_neq in H. rewrite H.
15   reflexivity. Qed.
```

다음 문제는 두 함수가 동일함을 증명하는 것이므로 함수 외연 책략을 사용한다.

```
1  Check @functional_extensionality.
2  (** forall (A B : Type) (f g : A -> B),
3      (forall x : A, f x = g x) -> f = g *)
4
5  Lemma t_update_shadow :
6    forall (A : Type) (m : total_map A) (x: string) (v1 v2: A),
7      (x !-> v2 ; x !-> v1 ; m) = (x !-> v2 ; m).
8  Proof.
9    intros. unfold t_update.
10   apply functional_extensionality.
11   intros.
12   destruct (x =? x0).
13   - reflexivity.
14   - reflexivity.
15 Qed.
```

라인 11을 실행한 후의 화면은 다음과 같다.

```
(if (x =? x0) then v2 else if (x =? x0) then v1 else m x0) =
(if (x =? x0) then v2 else m x0)
```

조금 복잡해 보이는데 이러한 if-then-else 식은 거의 항상 destruct를 사용하면 간단한 표현으로 환원된다.

다음 예에서는 라인 8 대신 라인 7을 사용하면 어떻게 되는지는 증명 뒤에 설명하였다. 그 다음 예에서도 비슷한 부분이 있다.

```
1  Theorem t_update_same : forall (A : Type) (m : total_map A) x,
2    (x !-> m x ; m) = m.
```

```
3  Proof.
4    intros. unfold t_update.
5    apply functional_extensionality.
6    intros.
7    (* destruct (x =? x0) eqn:E. Doesn't work this time. *)
8    destruct (eqb_spec x x0).
9    - rewrite e. reflexivity.
10   - reflexivity.
11 Qed.
```

라인 6을 실행한 후에 고울 프랍은 다음과 같다. 컨텍스트는 타입 선언만으로 이루어져 있고 가설은 없으므로 아래에 보여주지 않았다.

```
(if x =? x0 then m x else m x0) = m x0
```

라인 8을 실행하면 다음과 같이 된다. 지면을 아끼기 위하여 컨텍스트에서 타입 선언은 생략하고 가설들만 보였다.

```
Goal 1
e : x = x0
(1 / 2) --------------------
m x = m x0

Goal 2
n : x <> x0
(2 / 2) --------------------
m x0 = m x0
```

그 다음 라인 9 이하는 설명할 필요가 없을 것이다.

그럼 이제 왜 라인 8, `destruct (eqb_spec x x0)`를 실행하면 이렇게 되는지를 알아보자. 이것은 `eqb_spec`의 정의에 의하여 `destruct (reflect (x = x0) (x =? x0))`와 같다. 이것은 에비던스의 생성자별 분석이다. `reflect`의 정의에 의하여, ① `(x =? x0) = true`인 경우라면 생성자가 `reflectT`이었을 것이며 `x = x0`의 에비던스가 있어야 할 것이고, ② `(x =? x0) = false`인 경우라면 생성자가 `reflectF`이었을 것이며 `x <> x0`의 에비던스가 있어야 할 것이다. ✓

만일 라인 8 대신 라인 7을 실행했다면 고울 화면은 다음과 같을 것이다.

```
Goal 1
E : (x =? x0) = true
(1 / 2) --------------------
m x = m x0

Goal 2
E : (x =? x0) = false
(2 / 2) --------------------
m x0 = m x0
```

Goal 2는 설명할 것이 없고, Goal 1에서는 `eqb_eq in E`를 실행하여 `E: x = x0`로 바꾼 다음 진행하면 된다. ✓

```
1  Theorem t_update_permute : forall (A : Type) (m : total_map A)
2                                    v1 v2 x1 x2,
```

```
3     x2 <> x1 ->
4     (x1 !-> v1 ; x2 !-> v2 ; m) = (x2 !-> v2 ; x1 !-> v1 ; m).
5   Proof.
6     intros. unfold t_update.
7     apply functional_extensionality.
8     intros. destruct (eqb_spec x1 x).
9     - rewrite e in H. rewrite <- eqb_neq in H.
10      rewrite H. reflexivity.
11    - reflexivity.
12  Qed.
```

Partial Maps

부분맵(*partial map*)은 전맵에서 디폴트 값을 지정해 주지 않고 그냥 None으로 사용하는 것이다. 아주 간단한 개념이다.

```
1   Definition partial_map (A : Type) := total_map (option A).
2
3   Definition empty {A : Type} : partial_map A :=
4     t_empty None.
5
6   Definition update {A : Type} (m : partial_map A)
7               (x : string) (v : A) :=
8               (x !-> Some v ; m).
9
10  Notation "x '|->' v ';' m" := (update m x v)
11    (at level 100, v at next level, right associativity).
12
13  Notation "x '|->' v" := (update empty x v)
14    (at level 100).
15
16  Definition examplepmap :=
17    ("Church" |-> true ; "Turing" |-> false).
18
19  Compute (examplepmap "Church", examplepmap "Turing", examplepmap "vonNeumann").
20  (** = (Some true, Some false, None)
21    : option bool * option bool * option bool *)
```

앞서 전맵에 대해서 증명했던 모든 정리들의 부분맵 버전을 다음과 같이 쉽게 얻을 수 있다. 정리들의 이름은 전맵 버전의 이름에서 접두 t_를 빼서 사용한다.

```
1   Lemma apply_empty : forall (A : Type) (x : string),
2     @empty A x = None.
3   Proof.
4     intros. unfold empty.
5     rewrite t_apply_empty.
6     reflexivity.
7   Qed.
8
9   Lemma update_eq : forall (A : Type) (m : partial_map A) x v,
10    (x |-> v ; m) x = Some v.
11  Proof.
```

```
  intros. unfold update.
  rewrite t_update_eq.
  reflexivity.
Qed.
```

update_eq는 증명에서 자주 사용되므로 이를 다음과 같이 힌트 데이터베이스에 넣어 둔다. 이렇게 함으로써 앞으로 공부할 auto 책략 등에서 이것을 쉽게 찾을 수 있게 된다.

```
#[global] Hint Resolve update_eq : core.
```

```
Theorem update_neq : forall (A : Type) (m : partial_map A) x1 x2 v,
  x2 <> x1 ->
  (x2 |-> v ; m) x1 = m x1.
Proof.
  intros A m x1 x2 v H.
  unfold update. rewrite t_update_neq.
  - reflexivity.
  - apply H.
Qed.

Lemma update_shadow : forall (A : Type) (m : partial_map A) x v1 v2,
  (x |-> v2 ; x |-> v1 ; m) = (x |-> v2 ; m).
Proof.
  intros A m x v1 v2. unfold update.
  rewrite t_update_shadow.
  reflexivity.
Qed.

Theorem update_same : forall (A : Type) (m : partial_map A) x v,
  m x = Some v ->
  (x |-> v ; m) = m.
Proof.
  intros A m x v H. unfold update.
  rewrite <- H.
  apply t_update_same.
Qed.

Theorem update_permute : forall (A : Type) (m : partial_map A)
                                x1 x2 v1 v2,
  x2 <> x1 ->
  (x1 |-> v1 ; x2 |-> v2 ; m) = (x2 |-> v2 ; x1 |-> v1 ; m).
Proof.
  intros A m x1 x2 v1 v2. unfold update.
  apply t_update_permute.
Qed.
```

맵은 정의역이 문자열들의 집합인 함수이다. 공역이 동일한 두 맵에 대하여 포함관계inclusion relation을 정의해서 사용하는 것이 편리하다.

```
Definition includedin A : Type (m m' : partial_map A) :=
  forall x v, m x = Some v -> m' x = Some v.
```

Update는 포함관계를 유지한다는 사실을 다음과 같이 증명할 수 있다. 이 증명은 한 줄씩 따라가면서 음미하는 것이 좋겠다.

```
Lemma includedin_update : forall (A : Type) (m m' : partial_map A)
                                  (x : string) (vx : A),
  includedin m m' ->
  includedin (x |-> vx ; m) (x |-> vx ; m').
Proof.
  unfold includedin.
  intros A m m' x vx H.
  intros y vy.
  destruct (eqb_spec x y) as [Hxy | Hxy].
  - rewrite Hxy.
    rewrite update_eq. rewrite update_eq.
    intro H1. apply H1.
  - rewrite update_neq.
    + rewrite update_neq.
      { apply H. }
      { apply Hxy. }
    + apply Hxy.
Qed.
```

13

IMP, Simple Imperative Programs

이 장에서 우리는 Imp라는 간단한 명령형 프로그래밍 언어를 정의하고 이 언어로 작성된 프로그램을 Coq을 사용하여 분석할 것이다.

13.1 Arithemetic and Boolean Expressions

Inductive를 사용하여 산술 표현 aexp와 부울 표현 bexp을 정의한다.

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp).

Inductive bexp : Type :=
  | BTrue
  | BFalse
  | BEq (a1 a2 : aexp)
  | BNeq (a1 a2 : aexp)
  | BLe (a1 a2 : aexp)
  | BGt (a1 a2 : aexp)
  | BNot (b : bexp)
  | BAnd (b1 b2 : bexp).
```

1 + 2 * 3과 같은 산술식은 다음과 같은 aexp로 나타낼 수 있다.

```
APlus (ANum 1) (AMult (ANum 2) (ANum 3))
```

aexp의 값계산$_{\text{evaluation}}$ 함수 aeval을 다음과 같이 정의한다.

```
Fixpoint aeval (a : aexp) : nat :=
  match a with
  | ANum n => n
  | APlus  a1 a2 => (aeval a1) + (aeval a2)
  | AMinus a1 a2 => (aeval a1) - (aeval a2)
  | AMult  a1 a2 => (aeval a1) * (aeval a2)
  end.
```

```
Example test_aeval1:
  aeval (APlus (ANum 2) (ANum 2)) = 4.
Proof. simpl. reflexivity. Qed.
```

bexp의 값계산은 다음과 같이 정의한다.

```
Fixpoint beval (b : bexp) : bool :=
  match b with
  | BTrue       => true
  | BFalse      => false
  | BEq a1 a2   => (aeval a1) =? (aeval a2)
  | BNeq a1 a2  => negb ((aeval a1) =? (aeval a2))
  | BLe a1 a2   => (aeval a1) <=? (aeval a2)
  | BGt a1 a2   => negb ((aeval a1) <=? (aeval a2))
  | BNot b1     => negb (beval b1)
  | BAnd b1 b2  => andb (beval b1) (beval b2)
  end.
```

산술식과 부울식을 이렇게 정의해서 얻는 이점이 무엇인지 궁금할 것이다. 이점의 간단한 예를 들어 보면,

산술식에 나타나는 모든 부분 표현 `APlus (ANum 0) e`를 `e`로 대체할 수 있음 (13.1)

을 증명할 수 있다.

```
Fixpoint optimize_0plus (a:aexp) : aexp :=
  match a with
  | ANum n => ANum n
  | APlus (ANum 0) e2 => optimize_0plus e2
  | APlus  e1 e2 => APlus  (optimize_0plus e1) (optimize_0plus e2)
  | AMinus e1 e2 => AMinus (optimize_0plus e1) (optimize_0plus e2)
  | AMult  e1 e2 => AMult  (optimize_0plus e1) (optimize_0plus e2)
  end.

Example test_optimize_0plus :
  optimize_0plus (APlus (ANum 2)
                        (APlus (ANum 0)
                               (APlus (ANum 0) (ANum 1))))
  = APlus (ANum 2) (ANum 1).
Proof. simpl. reflexivity. Qed.
```

물론 위의 코드는 대략적인 아이디어를 보여주는 것이고, 엄격하게 (13.1)을 보이려면 다음과 같은 정리를 증명해야 한다.

```
Theorem optimize_0plus_sound: forall (a: aexp),
  aeval (optimize_0plus a) = aeval a.
```

이 정리의 증명은 3 부분으로 나누어 설명하겠다.

```
1 Proof.
2   intros a. induction a.
3   - (* a = ANum n *) (* aeval (optimize_0plus (ANum n)) = aeval (ANum n) *)
```

```
4       simpl. reflexivity.
5     - (* a = APlus a1 a2 *)
6       (* IHa1 : aeval (optimize_0plus a1) = aeval a1
7          IHa2 : aeval (optimize_0plus a2) = aeval a2
8          (1 / 1) --------
9          aeval (optimize_0plus (APlus a1 a2)) = aeval (APlus a1 a2) *)
10      destruct a1 eqn:Ea1.
11      + (* a1 = ANum n *) destruct n eqn:En.
12        { (* n = 0 *)
13          (* Ea1 : a1 = Anum 0
14             IHa1 : aeval (optimize_0plus (ANum 0)) = aeval (ANum 0)
15             IHa2 : aeval (optimize_0plus a2) = aeval a2
16             (1 / 1) --------
17             aeval (optimize_0plus (APlus (ANum 0) a2)) = aeval (APlus (ANum 0) a2) *)
18          simpl. exact IHa2. }
19        { (* n = S n0 *)
20          (* Ea1 : a1 = Anum (S n0)
21             IHa1 : aeval (optimize_0plus (ANum (S n0))) = aeval (ANum (S n0))
22             IHa2 : aeval (optimize_0plus a2) = aeval a2
23             (1 / 1) --------
24             aeval (optimize_0plus (APlus (ANum (S n0)) a2)) =
25             aeval (APlus (ANum (S n0)) a2) *)
26          simpl.
27          (* S (n0 + aeval (optimize_0plus a2)) = S (n0 + aeval a2) *)
28          rewrite IHa2. reflexivity. }
```

a: aexp에 대한 귀납을 하므로 a = ANum n인 경우, a = APlus a1 a2인 경우, ... 이런 식으로 4가지 경우로 나누어 진행해야 한다. 첫 번째 경우는 라인 3–4를 읽으면 쉽게 이해될 것이다.

두 번째 경우는 라인 11에서와 같이 a1을 destruct하여 다시 4가지 경우로 나누어 진행한다. a1 = Anum n 경우에는 다시 n = 0인 경우와 n = S n0인 경우로 나누어 진행한다.

n = 0인 경우에는 라인 6–9는 라인 13–17로 바뀌게 된다. 이때 라인 18의 simpl을 실행하면 고울 프랍이 IHa2의 프랍과 일치하게 된다.

n = S n0인 경우에는 조금 더 복잡하다. 우선 라인 6–9는 라인 20–25로 바뀌게 된다. 이때 라인 26의 simpl에 의하여 고울 프랍이 라인 27로 바뀌게 되는데 이 과정의 디테일을 이해하는 것이 중요하다. 고울 프랍의 좌변, 즉 라인 24는 다음과 같은 단계를 거쳐 라인 27의 좌변이 된다.

```
aeval (optimize_0plus (APlus (ANum (S n0)) a2)) =>
aeval (APlus (optimize_0plus (ANum (S n0))) (optimize_0plus a2)) =>
aeval (optimize_0plus (ANum (S n0))) + aeval (optimize_0plus a2) =>
aeval (ANum (S n0)) + aeval (optimize_0plus a2) =>
S n0 + aeval (optimize_0plus a2) =>
S (n0 + aeval (optimize_0plus a2))
```

고울 프랍의 우변, 즉 라인 25가 라인 27의 우변이 되는 것에 대해서는 설명을 생략하겠다.

그 다음 라인 28의 rewrite IHa2. reflexivity.는 쉽게 이해할 수 있을 것이다.

이것으로써 a = APlus a1 a2인 경우의 4가지 하위 경우 중 첫 번째 것을 완료하였다. 남은 3가지 하위 경우의 증명은 다음과 같다. 이들의 증명은 모두 유사하므로 첫 번째 하위 경우인 a1 = APlus a3 a4에 대해서만 설명하겠다.

```
29      + (* a1 = APlus a3 a4 *)
30        (* Ea1 : a1 = APlus a3 a4
31           IHa1 : aeval (optimize_0plus (APlus a3 a4)) = aeval (APlus a3 a4)
32           IHa2 : aeval (optimize_0plus a2) = aeval a2
33           (1 / 1) --------
34           aeval (optimize_0plus (APlus (APlus a3 a4) a2)) =
35           aeval (APlus (APlus a3 a4) a2) *)
36        simpl. simpl in IHa1.
37        (* IHa1 : aeval match a3 with
38           | ANum 0 => optimize_0plus a4
39           | _ => APlus (optimize_0plus a3) (optimize_0plus a4)
40           end = aeval a3 + aeval a4
41           IHa2 : aeval (optimize_0plus a2) = aeval a2
42           (1 / 1) --------
43           aeval match a3 with
44           | ANum 0 => optimize_0plus a4
45           | _ => APlus (optimize_0plus a3) (optimize_0plus a4)
46           end + aeval (optimize_0plus a2) = aeval a3 + aeval a4 + aeval a2 *)
47        rewrite IHa1.
48        rewrite IHa2. reflexivity.
49      + (* a1 = AMinus a3 a4 *)
50        simpl. simpl in IHa1. rewrite IHa1.
51        rewrite IHa2. reflexivity.
52      + (* a1 = AMult a3 a4 *)
53        simpl. simpl in IHa1. rewrite IHa1.
54        rewrite IHa2. reflexivity.
```

라인 36의 첫 번째 simpl은 고울 프랍을 단순화 한다. 우변의 단순화는 아주 쉬우므로 설명하지 않겠다. 좌변은 다음과 같은 단계를 거쳐 계산된다.

```
aeval (optimize_0plus (APlus (APlus a3 a4) a2)) =>
aeval (APlus (optimize_0plus (APlus a3 a4)) (optimize_0plus a2)) =>
aeval (optimize_0plus (APlus a3 a4)) + aeval (optimize_0plus a2)
```

여기서 멈춘다면 rewrite IHa1을 사용할 수 있을 것이니 좋다. 그런데 simpl은 멈추지 않고 aeval (optimize_0plus (APlus a3 a4))를 더 '단순화' 한다.

```
aeval (optimize_0plus (APlus a3 a4)) =>
aeval (APlus (optimize_0plus a3) (optimize_0plus a4)) =>
aeval (optimize_0plus a3) + aeval (optimize_0plus a4)
```

여기서 a3 = ANum 0인 경우와 그렇지 않은 경우로 나누어 결국 다음과 같이 된다.

```
aeval match a3 with
| ANum 0 => optimize_0plus a4
| _ => APlus (optimize_0plus a3) (optimize_0plus a4)
end
```

이것은 당연히 simpl IHa1의 결과의 좌변과 일치하게 된다. 그리고 simpl IHa1의 결과의 우변은 aeval a3 + aeval a4이므로, 이제 rewrite IHa1을 실행하면 고울 프랍이 다음과 같이 된다.

```
aeval a3 + aeval a4 + aeval (optimize_0plus a2) = aeval a3 + aeval a4 + aeval a2
```

여기서 라인 38의 `rewrite IHa2`를 실행하여 이 하위 경우의 증명을 마치게 된다.

남은 두 경우는 비교적 쉬우므로 설명을 생략하겠다. 이것으로써 증명 전체가 완료되었다.

```
55    - (* AMinus *)
56      simpl. rewrite IHa1. rewrite IHa2. reflexivity.
57    - (* AMult *)
58      simpl. rewrite IHa1. rewrite IHa2. reflexivity.  Qed.
```

(13.1)은 직관적으로 너무나 당연한 것이지만 엄격한 증명은 이토록 복잡하다. 다행히도 Coq은 자동 증명 기능이 있어 우리의 노력과 수고를 상당 부분 덜어줄 수 있다.

13.2 Coq Automation

Tacticals

택티컬(*Tactical*)은 책략을 인수로 사용하는 책략, 즉 higher-order tactic이다. 간단한 택티컬로 try가 있다. 이것은 그것의 인수 책략을 실행하여 잘 되면 그대로 진행하고 안 되면 아무것도 하지 않는다.

```
Theorem silly1 : forall (P : Prop), P -> P.
Proof.
  intros P HP.
  try reflexivity. (* Plain [reflexivity] would have failed. *)
  apply HP. (* We can still finish the proof in some other way. *)
Qed.

Theorem silly2 : forall ae, aeval ae = aeval ae.
Proof.
  try reflexivity. (* This just does [reflexivity]. *)
Qed.
```

try는 단독으로 사용되어서는 실제 증명에서 효용이 크지 않다. 하지만 택티컬 ;와 함께 사용하면 매우 유용할 수 있다. 복합 책략compound tactic `T ; T'`은 `T`를 실행하고, 결과로 나온 모든 하위 고울에 `T'`을 실행한다.

예를 들어 다음의 plus_1_neq_0의 증명에서 destruct를 사용하여 2개의 하위 고울을 만드는 것이 필요하다. 그런데 각 하위 고울에서의 증명은 동일하다. 이 중복을 피할 수 있다. ;를 사용하여 plus_1_neq_0'과 같이 하면 된다.

```
Theorem plus_1_neq_0 : forall n : nat,
  (n + 1) =? 0 = false.
Proof.
  intros n.
  destruct n as [| n'] eqn:E.
  - simpl. reflexivity.
  - simpl. reflexivity.   Qed.

Theorem plus_1_neq_0' : forall n : nat,
  (n + 1) =? 0 = false.
Proof.
```

```
  intros n.
  destruct n as [| n'] eqn:E; simpl; reflexivity.
  Qed.
```

p260에 있는 optimize_0plus_sound의 증명에서 a a1: aexp에 대한 생성자별 분석을 사용할 때 동일한 증명의 반복이 있었는데, 이를 try와 ;를 사용하여 피할 수 있다. 이를 두고 [cases; try (..;..)] 관용구idiom라고 부른다. 여기서 cases는 induction/destruct/inversion 중 어느 하나이다.

```
Theorem optimize_0plus_sound': forall a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a.
  induction a;
    try (simpl; rewrite IHa1; rewrite IHa2; reflexivity).
    (* without [try] this will get stuck *)
  - (* ANum *) reflexivity.
  - (* APlus *) destruct a1;
    try (simpl; simpl in IHa1; rewrite IHa1;
         rewrite IHa2; reflexivity).
    + (* a1 = ANum n *) destruct n;
      simpl; rewrite IHa2; reflexivity.   Qed.
```

다음과 같이 불릿을 제거할 수도 있다.

```
Theorem optimize_0plus_sound'': forall a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a.
  induction a;
    try (simpl; rewrite IHa1; rewrite IHa2; reflexivity);
    try reflexivity;
    (* APlus *) destruct a1;
    try (simpl; simpl in IHa1; rewrite IHa1;
         rewrite IHa2; reflexivity);
    (* a1 = ANum n *) destruct n;
    simpl; rewrite IHa2; reflexivity. Qed.
```

생성자별 분석에서 경우에 따라 증명이 다를 때도 ;를 사용하여 불릿이 없는 증명을 얻을 수 있다. 다음과 같은 구문을 사용한다.

```
T; [T1 | T2 | ... | Tn]
```

위의 구문은 n개의 하위 고울 각각에 대하여 책략 Ti를 실행하라는 뜻이다. 그러므로 T; T'는 다음의 구문과 동일하다.

```
T; [T' | T' | ... | T']
```

경우별 분석이 아닌 경우에도 책략을 반복해야 하는 경우가 있다. 예를 들어

```
In10 : In 10 [1;2;3;4].
```

는 다음과 같이 증명할 수 있다.

```
unfold In. right. right. right. left. reflexivity.
(* [unfold In] may by omitted. *)
```

이런 경우에 repeat 택티컬이 유용하다. 그런데 단순히 repeat right를 사용하면 안 된다. 만일 이렇게 한다면 right가 성공하는 한 계속 진행하기 때문에 다음의 결과를 얻게 될 것이다.

```
In 4 []
```

그러므로 right를 실행한 후에는 반드시 혹시 left; reflexivity가 성공하는지를 확인해 보아야 한다. 아니 일반적으로 In a li를 증명하고자 할 때는 제일 첫 원소가 a인 경우도 있으므로 left; reflexivity를 right에 앞서 실행하는 것이 더 좋다.

```
Theorem In4' : In 10 [1;2;3;4].
Proof.
  repeat (try (left; reflexivity); right).
Qed.
```

repeat T는 T가 fail하기 전에 멈추게 되므로 절대로 fail하지 않는다. 그래서 때로는 repeat T가 무한 루프에 빠질 수 있으므로 이런 일이 발생하지 않도록 주의해야 한다.

```
Theorem repeat_loop : forall (m n : nat),
  m + n = n + m.
Proof.
  intros m n.
  (* repeat rewrite Nat.add_comm. *)
Admitted.
```

연습문제 13.1 optimize_0plus는 주어진 bexp내에 있는 aexp의 값들을 그대로 유지하므로 그 bexp의 값도 유지할 것이다. 이 사실을 말하는 정리 optimize_0plus_b_sound를 증명하라. 먼저 함수 optimize_0plus_b: bexp -> bexp를 정의해야 할 것이다.

```
Fixpoint optimize_0plus_b (b : bexp) : bexp :=
  match b with
  | BEq l r => BEq (optimize_0plus l) (optimize_0plus r)
  | BLe l r => BLe (optimize_0plus l) (optimize_0plus r)
  | BNot b' => BNot (optimize_0plus_b b')
  | BAnd l r => BAnd (optimize_0plus_b l) (optimize_0plus_b r)
  | _ => b
  end.

Theorem optimize_0plus_b_sound : forall b,
  beval (optimize_0plus_b b) = beval b.
Proof.
  intros.
  induction b;
    try (simpl; reflexivity);
    try (simpl; repeat rewrite optimize_0plus_sound; reflexivity).
    - simpl. rewrite IHb. reflexivity.
    - simpl. rewrite IHb1, IHb2. reflexivity.
Qed.
```

Defining New Tactics

Ltac idiom을 써서 몇 개의 책략을 묶어 하나의 새로운 책략을 정의할 수 있다. 아래에 간단한 예를 보였다.

```
Ltac invert H :=
  inversion H; subst; clear H.

Lemma invert_example1 : forall a b c: nat,
  [a ;b] = [a;c] -> b = c.
Proof.
  intros.
  invert H.
  reflexivity.
Qed.
```

Ltac 사용법을 공부하기 위한 좋은 자료는 Certified Programming with Dependent Types (CPDT)이다.

The lia Tactic

lia는 Linear Integer Arithmetic의 약자로, '정수에 대한 선형 산술식'을 다루는 데 유용한 책략이다.[1]

```
Example silly_presburger_example : forall m n o p,
  m + n <= n + o /\ o + 3 = p + 3 ->
  m <= p.
Proof. lia. Qed.

Example add_comm__lia : forall m n,
  m + n = n + m.
Proof. lia. Qed.

Example add_assoc__lia : forall m n p,
  m + (n + p) = m + n + p.
Proof. lia. Qed.
```

lia 책략을 사용하기 위하여는 이전에 다음을 실행해 두어야 한다.

```
From Coq Require Import Lia.
```

A Few More Handy Tactics

간단하고 유용한 몇 개의 책략을 소개한다.

- `clear H`: Delete hypothesis `H` from the context. (컨텍스트에 더 이상 남아있을 필요가 없어진 가설을 제거한다.)
- `subst x`: Given a variable `x`, find an assumption `x = e` or `e = x` in the context, replace `x` with `e` throughout the context and current goal, and clear the assumption.

[1] 정수에 대한 선형 산술식이 무엇인지는 SF에 나와 있다.

- `subst`: Substitute away *all assumptions* of the form `x = e` or `e = x`(where `x` is a variable). (유용하다.)
- `rename .. into ..`: Change the name of a hypothesis in the proof context. For example, if the context includes a variable named `x`, then `rename x into y` will change all occurrences of `x` to `y`. (무슨 소린지?)
- `assumption`: Try to find a hypothesis `H` in the context that exactly matches the goal; if one is found, solve the goal. (별로다.)
- `contradiction`: Try to find a hypothesis `H` in the context that is logically equivalent to `False`; if one is found, solve the goal. (별로다.)
- `constructor`: Try to find a constructor `c` (from some `Inductive` definition in the current environment) that can be applied to solve the current goal. If one is found, behave like `apply c`. (유용하다.)

이 책략들의 사용 예는 앞으로 공부하면서 보게 될 것이다.

Multiple Arguments and Targets of Tactics

이 책에서는 책략의 인수와 타겟이라는 용어를 사용한다. 이 용어들의 의미는 다음의 구문에서 쉽게 이해할 수 있다.

tactic_name [*arg1*, ..] [`in` *target1*, ..].

`rewrite` 책략을 예로 들어 설명하겠다.

```
1  From Coq Require Import Arith.
2
3  Fact silly_multiple : forall a b x y z: nat,
4    a = x + y -> b = y + z -> a + b =  (y + x) + (z + y).
5  Proof.
6    intros a b x y z H1 H2.
7    rewrite Nat.add_comm in H1, H2.
8    rewrite H1, H2.
9    reflexivity.
10 Qed.
```

라인 7은 multiple targets의 예이며 다음의 두 줄을 하나로 합친 것이다.

```
rewrite Nat.add_comm in H1.
rewrite Nat.add_comm in H2.
```

라인 8은 multiple arguments의 예이며 다음의 두 줄을 하나로 합친 것이다.

```
rewrite H1.
rewrite H2.
```

13.3 Evaluation as a Relation

aeval과 beval은 Fixpoint를 이용하여 정의된 함수이다. Evaluation은 함수가 아닌 관계relation 로도 정의할 수 있다. (관계는 2개의 인수를 가지는 프랍을 뜻한다.)

```
Reserved Notation "e '==>' n" (at level 90, left associativity).

Inductive aevalR : aexp -> nat -> Prop :=
  | E_ANum (n : nat) :
      (ANum n) ==> n
  | E_APlus (e1 e2 : aexp) (n1 n2 : nat) :
      (e1 ==> n1) ->
      (e2 ==> n2) ->
      (APlus e1 e2)  ==> (n1 + n2)
  | E_AMinus (e1 e2 : aexp) (n1 n2 : nat) :
      (e1 ==> n1) ->
      (e2 ==> n2) ->
      (AMinus e1 e2) ==> (n1 - n2)
  | E_AMult (e1 e2 : aexp) (n1 n2 : nat) :
      (e1 ==> n1) ->
      (e2 ==> n2) ->
      (AMult e1 e2)  ==> (n1 * n2)

  where "e '==>' n" := (aevalR e n) : type_scope.
```

각 생성자에 대한 ==>의 생성규칙에서 콜론 오른쪽이 조건문인 경우 그 전건(들)을 콜론 왼쪽으로 옮겨 '->'를 사라지게 할 수 있다.

```
  ..
  | E_APlus (e1 e2 : aexp) (n1 n2 : nat)
      (H1 : e1 ==> n1)
      (H2 : e2 ==> n2) :
      (APlus e1 e2)  ==> (n1 + n2)
  ..
```

물론 모든 것을 콜론의 오른쪽에 둘 수도 있다.

```
  ..
  | E_APlus : forall (e1 e2 : aexp) (n1 n2 : nat),
      (e1 ==> n1) ->
      (e2 ==> n2) ->
      (APlus e1 e2)  ==> (n1 + n2)
  ..
```

Inference Rule Notation

각 생성자에 대한 '==>' 생성규칙은 Coq 코드가 아닌 일상적인informal 문서에서는 추론규칙inference rule의 형식으로 나타낼 수 있다.

$$\frac{}{\texttt{ANum}\ n \Rightarrow n}\ (\texttt{E_ANum})$$

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{\texttt{APlus}\ e_1\ e_2 \Rightarrow n_1 + n_2} \quad (\texttt{E_APlus})$$

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{\texttt{AMinus}\ e_1\ e_2 \Rightarrow n_1 - n_2} \quad (\texttt{E_AMinus})$$

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{\texttt{AMult}\ e_1\ e_2 \Rightarrow n_1 * n_2} \quad (\texttt{E_AMult})$$

부울 표현식에 대해서도 추론규칙을 다음과 같이 나타낼 수 있다.

$$\frac{}{\texttt{Btrue} \Rightarrow \texttt{true}} \quad (\texttt{E_BTrue})$$

$$\frac{}{\texttt{BFalse} \Rightarrow \texttt{false}} \quad (\texttt{E_BFalse})$$

$$\frac{a_1 \Rightarrow n_1 \qquad a_2 \Rightarrow n_2}{\texttt{BEq}\ a_1\ a_2 \Rightarrow n_1 = n_2} \quad (\texttt{E_BEq})$$

$$\frac{a_1 \Rightarrow n_1 \qquad a_2 \Rightarrow n_2}{\texttt{BNeq}\ a_1\ a_2 \Rightarrow n_1 \neq n_2} \quad (\texttt{E_BNeq})$$

$$\frac{a_1 \Rightarrow n_1 \qquad a_2 \Rightarrow n_2}{\texttt{BLe}\ a_1\ a_2 \Rightarrow n_1 \leqslant n_2} \quad (\texttt{E_Le})$$

$$\frac{a_1 \Rightarrow n_1 \qquad a_2 \Rightarrow n_2}{\texttt{BGt}\ a_1\ a_2 \Rightarrow \neg(n_1 \leqslant n_2)} \quad (\texttt{E_Gt})$$

$$\frac{b \Rightarrow v}{\texttt{BNot}\ b \Rightarrow \neg v} \quad (\texttt{E_BNot})$$

$$\frac{b_1 \Rightarrow v_1 \qquad b_2 \Rightarrow v_2}{\texttt{BAnd}\ b_1\ b_2 \Rightarrow v_1 \wedge v_2} \quad (\texttt{E_BAnd})$$

Equivalence of the Definitions

aexp에 대한 값계산은 함수 aeval로 정의한 것과 관계 aevalR로 정의한 것의 두 가지가 있는데, 이들이 사실상 동등하다는 것을 다음의 정리를 통하여 증명하기로 한다.

```
Theorem aeval_iff_aevalR : forall (a: aexp) (n: nat),
  (a ==> n) <-> aeval a = n.
```

우선 ->를 증명하기로 한다.

```
1   intros. split.
2   - (* a ==> n -> aeval a = n *)
3     intros H.
4     induction H.
5     + (* E_ANum *)
6       simpl.
7       reflexivity.
```

```
8       + (* E_APlus *)
9         simpl.
10        rewrite IHaevalR1.  rewrite IHaevalR2.  reflexivity.
11      + (* E_AMinus *)
12        (* same as E_APlus case *)
13      + (* E_AMult *)
14        (* same as E_APlus case *)
15    - (* <- *)
16      ..
```

라인 4를 실행한 후의 고울 화면은 다음과 같다.

```
17  Goal 1
18  n : nat
19  (1 / 4) ----------------
20  aeval (ANum n) = n
21
22  Goal 2
23  e1, e2 : aexp
24  n1, n2 : nat
25  H : e1 ==> n1
26  H0 : e2 ==> n2
27  IHaevalR1 : aeval e1 = n1
28  IHaevalR2 : aeval e2 = n2
29  (2 / 4) ----------------
30  aeval (APlus e1 e2) = n1 + n2
31
32  Goal 3
33  (* same as in Goal 2 *)
34  (3 / 4) ----------------
35  aeval (AMinus e1 e2) = n1 - n2
36
37  Goal 4
38  (* same as in Goal 2 *)
39  (4 / 4) ----------------
40  aeval (AMult e1 e2) = n1 * n2
```

라인 6을 실행하면 고울 프랍이 `n = n`이 된다. ✓

라인 9를 실행하면 고울 프랍이 `aeval e1 + aeval e2 = n1 + n2`가 된다. 여기서 `rewrite`를 두 번 실행하면 양변이 동일해진다. ✓

남은 두 경우에 대해서는 더 이상의 설명이 필요 없을 것이다.

이제 택티컬을 써서 증명을 간략하게 만들어 보자.

라인 9를 실행한 뒤의 고울 프랍은 (앞서 말했듯이) 다음과 같다.

```
aeval e1 + aeval e2 = n1 + n2
```

여기서 `subst`를 실행하면 라인 27과 라인 28에 의하여 고울 프랍이 다음과 같게 된다.

```
aeval e1 + aeval e2 = aeval e1 + aeval e2
```

즉, `rewrite`가 필요 없이 `reflexivity`만 사용하면 된다.[2] 실은 `reflexivity`는 `simpl`을 포함하므로 라인 9와 라인 10은 다음과 같은 하나의 라인으로 바꿀 수 있다.

[2]등식의 양변 중 하나가 변수일 때는 `rewrite` 대신 `subst`를 사용하는 것이 더 간편하다.

```
subst. reflexivity.
```

라인 6과 라인 7도 마찬가지다. 이 경우에는 `subst. reflexivity.`에서 `subst`는 아무런 일도 하지 않지만 문제는 없다. 그러므로 복합 책략을 써서 라인 3 ~ 14는 다음과 같은 단 하나의 라인으로 대체할 수 있다.

```
intros H. induction H; subst; reflexivity.
```

그 다음은 역방향 `<-`이다. 이번에는 `induction H`가 아니라 `induction a`를 사용하는 것이 당연하다. 귀납 증명에서 귀납 변수 아닌 다른 변수들은 전칭한정 변수로 남겨두는 것이 통상 더 나으므로 라인 42에 `generalize dependent`를 사용하였다. 이를 생략하면 어디에서 문제가 발생하는지 확인해 보라.

```
41    - (* aeval a = n -> a ==> n *)
42      generalize dependent n.
43      induction a.
44      + (* ANum *)
45        simpl. intros. subst.
46        apply E_ANum.
47      + (* APlus *)
48        simpl. intros. subst.
49        apply E_APlus.
50        * apply IHa1. reflexivity.
51        * apply IHa2. reflexivity.
52      + (* AMinus *)
53        (* same as APlus case *) apply E_AMinus. (* same as APlus case *)
54      + (* AMult *)
55        (* same as APlus case *) apply E_AMult. (* same as APlus case *)
56  Qed.
```

라인 43을 실행한 후의 고울 화면은 다음과 같다.

```
57  Goal 1
58  n : nat
59  (1 / 4) ----------------
60  forall n0 : nat, aeval (ANum n) = n0 -> ANum n ==> n0
61
62  Goal 2
63  a1, a2 : aexp
64  IHa1 : forall n : nat, aeval a1 = n -> a1 ==> n
65  IHa2 : forall n : nat, aeval a2 = n -> a2 ==> n
66  (2 / 4) ----------------
67  forall n : nat, aeval (APlus a1 a2) = n -> APlus a1 a2 ==> n
68
69  Goal 3
70  (* same as in Goal 2 *)
71  (3 / 4) ----------------
72  forall n : nat, aeval (AMinus a1 a2) = n -> AMinus a1 a2 ==> n
73
74  Goal 4
75  (* same as in Goal 2 *)
76  (4 / 4) ----------------
77  forall n : nat, aeval (AMult a1 a2) = n -> AMult a1 a2 ==> n
```

라인 45를 실행하면 고울 프랍이 다음과 같이 된다.

```
ANum n0 ==> n0
```

따라서 이것은 라인 46으로 해결된다.

라인 48을 실행하면 고울 프랍이 다음과 같이 된다.

```
APlus a1 a2 ==> aeval a1 + aeval a2
```

따라서 apply E_Aplus를 실행하면 다음과 같은 2개의 하위 고울이 생성된다.

```
..
(1 / 2) ----------------
a1 ==> aeval a1
..
(2 / 2) ----------------
a2 ==> aeval a2
```

여기서 라인 50의 apply IHa1을 실행하면 고울 프랍이 다음과 같이 된다.

```
aeval a1 = aeval a1
```

APlus 경우에 대한 설명은 이것으로 충분할 것이다. ✓

나머지 경우도 거의 비슷하다. 다만 apply E_APlus 대신 각각 apply E_AMinus와 apply E_AMult를 사용해야 한다는 것만 다르다. 이 책략들은 constructor로 대체할 수 있다.

이제 try와 ;를 사용하여 반복된 코드를 제거하면 다음과 같이 간략화 된 증명을 얻을 수 있다.

```
Theorem aeval_iff_aevalR' : forall (a: aexp) (n: nat),
  (a ==> n) <-> aeval a = n.
Proof.
  intros. split.
  - (* -> *)
    intros H. induction H; subst; reflexivity.
  - (* <- *)
    generalize dependent n.
    induction a; simpl; intros; subst; constructor;
       try apply IHa1; try apply IHa2; reflexivity.
Qed.
```

이제 aevalR과 같은 요령으로 bevalR을 정의하고 beval_iff_bevalR을 증명해 보자.

```
Reserved Notation "e '==>b' b" (at level 90, left associativity).

Inductive bevalR : bexp -> bool -> Prop :=
  | E_BTrue : BTrue ==>b true
  | E_BFalse : BFalse ==>b false
  | E_BEq (a1 a2: aexp) (n1 n2: nat) :
      (a1 ==> n1) -> (a2 ==> n2) -> (BEq a1 a2) ==>b (n1 =? n2)
  | E_BNeq (a1 a2: aexp) (n1 n2: nat) :
      (a1 ==> n1) -> (a2 ==> n2) -> (BNeq a1 a2) ==>b negb (n1 =? n2)
  | E_BLe (a1 a2: aexp) (n1 n2: nat) :
      (a1 ==> n1) -> (a2 ==> n2) -> (BLe a1 a2) ==>b (n1 <=? n2)
```

```
  | E_BGt (a1 a2: aexp) (n1 n2: nat) :
      (a1 ==> n1) -> (a2 ==> n2) -> (BGt a1 a2) ==>b negb (n1 <=? n2)
  | E_BNot (b: bexp) (bv: bool) :
      (b ==>b bv) -> (BNot b) ==>b (negb bv)
  | E_BAnd (b1 b2: bexp) (bv1 bv2: bool) :
      (b1 ==>b bv1) -> (b2 ==>b bv2) -> (BAnd b1 b2) ==>b (andb bv1 bv2)

where "e '==>b' b" := (bevalR e b) : type_scope.
```

```
1  Lemma beval_iff_bevalR : forall (b: bexp) (bv: bool),
2    b ==>b bv <-> beval b = bv.
3  Proof.
4    intros. split.
5    - intros.
6      induction H; subst; simpl;
7        try reflexivity;
8        try (rewrite aeval_iff_aevalR in H, H0;
9             rewrite H, H0;
10            reflexivity).
11   - intros. generalize dependent bv.
12     induction b; simpl; intros; subst; constructor;
13       try (apply aeval_iff_aevalR; reflexivity);
14       try (apply IHb; reflexivity);
15       try (apply IHb1; reflexivity);
16       try (apply IHb2; reflexivity).
17 Qed.
```

Computational vs. Relational Definitions

표현식의 값계산을 aeval, beval 등의 함수로 정의하는 것(함수형 정의)과 aevalR, bevalR 등의 관계로 정의하는 것(관계형 정의)은 취향에 따라 결정할 문제이지만, 때로는 후자를 써야만 하는 경우가 있다. 전함수total function가 아닌 부분함수partial function인 경우에 그러하다. 예를 들어 자연수의 나눗셈을 $a \div b = c \iff (a = b \times c) \wedge (b \neq 0)$로 정의하면 많은 a, b에 대해서 $a \div b$는 존재하지 않는다. 이러한 부분함수는 다음과 같이 정의할 수 있다.

```
Inductive aexp : Type :=
  ..
  | ADiv (a1 a2 : aexp).   (* <--- NEW *)

Inductive aevalR : aexp -> nat -> Prop :=
  ..
  | E_ADiv (a1 a2 : aexp) (n1 n2 n3 : nat) :     (* <----- NEW *)
    (a1 ==> n1) -> (a2 ==> n2) -> (n2 > 0) ->
    (mult n2 n3 = n1) -> (ADiv a1 a2) ==> n3
```

이 외에도 non-deterministic function 등 total function이 아닌 어떠한 관계도 관계형 정의로만 가능하다.

함수형 정의와 관계형 정의의 장단점은 다음과 같다.

- 관계형 정의: 함수가 아닌 관계의 정의에 사용할 수 있다. inversion과 induction 책략이 자동으로 주어진다.

- 함수형 정의: simpl, reflexivity, rewrite 등의 책략을 사용할 수 있다.

Coq 프로젝트에서 함수형과 관계형 정의를 둘 다 주고, 이 둘이 동등함을 증명하는 방식을 취하는 경우가 많다.

13.4 Expressions with Variables

변수를 포함하는 산술식을 다루기 위하여 aexp를 다음과 같이 확장한다.

```
Inductive aexp : Type :=
| ANum (n : nat)
| AId (x : string)              (* <--- NEW *)
| APlus (a1 a2 : aexp)
| AMinus (a1 a2 : aexp)
| AMult (a1 a2 : aexp).
```

자주 쓰이는 변수 이름을 다음과 같이 정의해 둔다.

```
Definition W : string := "W".
Definition X : string := "X".
Definition Y : string := "Y".
Definition Z : string := "Z".
```

bexp의 정의는 이전 것을 그대로 사용한다. 그러나 bexp는 aexp의 원소를 포함하므로 aexp의 확장에 따라 bexp도 확장된다.

States

변수들의 값은 state라는 전맵total map으로 구현한다. 정확히 말하자면 state는 타입이고, state의 원소가 string에서 nat로 가는 함수이므로 이것으로써 *machine state*, 즉 변수들의 값을 나타낸다.

```
Definition state := total_map nat.
```

이 장에서 우리가 공부할 프로그래밍 언어 Imp의 간단한 프로그램을 아래 보였다.

```
Z := X;
Y := 1;
while Z <> 0 do
  Y := Y * Z;
  Z := Z - 1
end
```

이 예에서 보듯이 Imp는 변수를 포함한 산술식과 부울식을 다루는 언어이므로 Imp의 시맨틱스에는 state라고 하는 전맵이 핵심적인 요소이다. 부분맵은 여기서는 사용하지 않는다.

Notations

Imp 프로그램을 쓰고 읽기 쉽게 하기 위하여 몇 가지 표기법을 정의할 것인데, 이게 상당히 복잡하다. 일단 정의부터 보자. 설명은 뒤에 하겠다.

```
Coercion AId : string >-> aexp.
Coercion ANum : nat >-> aexp.

Declare Custom Entry com.
Declare Scope com_scope.

Notation "< e >" := e (at level 0, e custom com at level 99) : com_scope.
Notation "( x )" := x (in custom com, x at level 99) : com_scope.
Notation "x" := x (in custom com at level 0, x constr at level 0) : com_scope.
Notation "f x .. y" := (.. (f x) .. y)
                  (in custom com at level 0, only parsing,
                  f constr at level 0, x constr at level 9,
                  y constr at level 9) : com_scope.
Notation "x + y"   := (APlus x y) (in custom com at level 50, left associativity).
Notation "x - y"   := (AMinus x y) (in custom com at level 50, left associativity).
Notation "x * y"   := (AMult x y) (in custom com at level 40, left associativity).
Notation "'true'"  := true (at level 1).
Notation "'true'"  := BTrue (in custom com at level 0).
Notation "'false'" := false (at level 1).
Notation "'false'" := BFalse (in custom com at level 0).
Notation "x <= y"  := (BLe x y) (in custom com at level 70, no associativity).
Notation "x > y"   := (BGt x y) (in custom com at level 70, no associativity).
Notation "x = y"   := (BEq x y) (in custom com at level 70, no associativity).
Notation "x <> y"  := (BNeq x y) (in custom com at level 70, no associativity).
Notation "x && y"  := (BAnd x y) (in custom com at level 80, left associativity).
Notation "'~' b"   := (BNot b) (in custom com at level 75, right associativity).

Open Scope com_scope.
```

여기에 대한 설명은 SF의 원문으로 갈음한다.

> You do not need to understand exactly what these declarations do.
>
> Briefly, though:
>
> - The `Coercion` declaration stipulates that a function (or constructor) can be implicitly used by the type system to coerce a value of the input type to a value of the output type. For instance, the coercion declaration for `AId` allows us to use plain strings when an `aexp` is expected; the string will implicitly be wrapped with `AId`.
> - `Declare Custom Entry com` tells `Coq` to create a new *custom grammar* for parsing `Imp` expressions and programs. The first notation declaration after this tells `Coq` that anything between `<{` and `}>` should be parsed using the `Imp` grammar. Again, it is not necessary to understand the details, but it is important to recognize that we are defining *new* interpretations for some familiar operators like `+`, `-`, `*`, `=`, `<=`, etc., when they occur between `<{` and `}>`.

We can now write `3 + (X * 2)` instead of `APlus 3 (AMult X 2)`, and `true && ~(X <= 4)` instead of `BAnd true (BNot (BLe X 4))`.

이 표기법에 의하여, 예를 들어 다음과 같이 쓸 수 있다.

```
Definition example_aexp : aexp := <{ 3 + (X * 2) }>.
Definition example_bexp : bexp := <{ true && ~(X <= 4) }>.
```

Evaluation

확장된 aexp와 bexp의 값계산은 state을 인수로 사용해야 한다.

```
Fixpoint aeval (st : state) (* <--- NEW *)
               (a : aexp) : nat :=
  match a with
  | ANum n => n
  | AId x => st x (* <--- NEW *)
  | <{a1 + a2}> => (aeval st a1) + (aeval st a2)
  | <{a1 - a2}> => (aeval st a1) - (aeval st a2)
  | <{a1 * a2}> => (aeval st a1) * (aeval st a2)
  end.

Fixpoint beval (st : state) (* <--- NEW *)
               (b : bexp) : bool :=
  match b with
  | <{true}>      => true
  | <{false}>     => false
  | <{a1 = a2}>   => (aeval st a1) =? (aeval st a2)
  | <{a1 <> a2}>  => negb ((aeval st a1) =? (aeval st a2))
  | <{a1 <= a2}>  => (aeval st a1) <=? (aeval st a2)
  | <{a1 > a2}>   => negb ((aeval st a1) <=? (aeval st a2))
  | <{~ b1}>      => negb (beval st b1)
  | <{b1 && b2}>  => andb (beval st b1) (beval st b2)
  end.
```

전에 정의해 두었던 표기법을 다음과 같이 요긴하게 사용한다.

```
Definition empty_st := (_ !-> 0).

Notation "x '!->' v" := (x !-> v ; empty_st) (at level 100).

Example aexp1 :
    aeval (X !-> 5) <{ 3 + (X * 2) }>
  = 13.
Proof. simpl. reflexivity. Qed.

Example aexp2 :
    aeval (X !-> 5 ; Y !-> 4) <{ Z + (X * Y) }>
  = 20.
Proof. reflexivity. Qed.

Example bexp1 :
    beval (X !-> 5) <{ true && ~(X <= 4) }>
```

```
    = true.
Proof. reflexivity. Qed.
```

13.5 Commands

Syntax

프로그래밍 언어 Imp의 *command*는 다음과 같이 정의된다.

```
Inductive com : Type :=
| CSkip
| CAsgn (x : string) (a : aexp)
| CSeq (c1 c2 : com)
| CIf (b : bexp) (c1 c2 : com)
| CWhile (b : bexp) (c : com).

Notation "'skip'"  :=
         CSkip (in custom com at level 0) : com_scope.
Notation "x := y"  :=
         (CAsgn x y)
            (in custom com at level 0, x constr at level 0,
             y at level 85, no associativity) : com_scope.
Notation "x ; y" :=
         (CSeq x y)
           (in custom com at level 90,
            right associativity) : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" :=
         (CIf x y z)
           (in custom com at level 89, x at level 99,
            y at level 99, z at level 99) : com_scope.
Notation "'while' x 'do' y 'end'" :=
         (CWhile x y)
           (in custom com at level 89, x at level 99,
            y at level 99) : com_scope.
```

이 언어로 팩토리얼을 계산하는 커맨드(프로그램)을 다음과 같이 작성할 수 있다. X에 저장된 값을 x: nat라고 했을 때 이 프로그램은 $x!$을 계산하여 Y에 저장한다.

```
Definition fact_in_coq : com :=
<{ Z := X;
   Y := 1;
   while Z <> 0 do
     Y := Y * Z;
     Z := Z - 1
   end }>.
```

이 프로그램을 '돌리는' 방법은 조금 후에 설명하겠다.

Desugaring Notations

코드에 Notaion을 많이 쓰다 보면 오히려 혼란스러울 수 있다. 이럴 때 다음과 같은 명령을 써서 표기법 기능을 끄고 켤 수 있다.

- `Unset Printing Notations` (undo with `Set Printing Notations`)
- `Set Printing Coercions` (undo with `Unset Printing Coercions`)
- `Set Printing All` (undo with `Unset Printing All`)

다음의 예를 보면 이해가 될 것이다.

```
Unset Printing Notations.
Print fact_in_coq.
(* ===>
    fact_in_coq =
    CSeq (CAsgn Z X)
        (CSeq (CAsgn Y (S O))
              (CWhile (BNot (BEq Z O))
                      (CSeq (CAsgn Y (AMult Y Z))
                            (CAsgn Z (AMinus Z (S O))))))
        : com *)
Set Printing Notations.

Print example_bexp.
(* ===> example_bexp = <{(true && ~ (X <= 4))}> *)

Set Printing Coercions.
Print example_bexp.
(* ===> example_bexp = <{(true && ~ (AId X <= ANum 4))}> *)

Print fact_in_coq.
(* ===>
  fact_in_coq =
  <{ Z := (AId X);
      Y := (ANum 1);
      while ~ (AId Z) = (ANum 0) do
        Y := (AId Y) * (AId Z);
        Z := (AId Z) - (ANum 1)
      end }>
        : com *)
Unset Printing Coercions.
```

Locate Again

표기법이 많아지면 동일한, 혹은 비슷한 기호의 의미를 혼동할 수 있다. 이럴 때 Locate가 유용하다.

```
Locate aexp.
Locate "&&".
Locate ";".
Locate "while".
```

More Examples

```
Definition plus2 : com :=
  <{ X := X + 2 }>.

Definition XtimesYinZ : com :=
  <{ Z := X * Y }>.

Definition subtract_slowly_body : com :=
  <{ Z := Z - 1 ;
     X := X - 1 }>.

Definition subtract_slowly : com :=
  <{ while X <> 0 do
       subtract_slowly_body
     end }>.

Definition subtract_3_from_5_slowly : com :=
  <{ X := 3 ;
     Z := 5 ;
     subtract_slowly }>.

Definition loop : com :=
  <{ while true do
       skip
     end }>.
```

13.6 Evaluating Commands

while 루프는 terminate가 보장되어 있지 않다는 사실이 이 작업을 어렵게 한다.

Evaluation as a Function (Failed Attempt)

문제가 되는 while은 잠시 접어두고 다음과 같이 정의해 보자.

```
1  Fixpoint ceval_fun_no_while (st : state) (c : com) : state :=
2    match c with
3    | <{ skip }> =>
4        st
5    | <{ x := a }> =>
6        (x !-> (aeval st a) ; st)
7    | <{ c1 ; c2 }> =>
8        let st' := ceval_fun_no_while st c1 in
9        ceval_fun_no_while st' c2
10   | <{ if b then c1 else c2 end}> =>
11       if (beval st b)
12         then ceval_fun_no_while st c1
13         else ceval_fun_no_while st c2
14   | <{ while b do c end }> =>
15       st  (* bogus *)
16 end.
```

```
17
18 Check ceval_fun_no_while. (* state -> com -> state *)
```

라인 8,9에서 `let .. := .. in ..` 구문을 사용했다. 이것은 다음을 의미한다.

```
ceval_fun_no_while (ceval_fun_no_while c1) c2
```

라인 14,15를 다음과 같이 바꾸어 `while` 루프를 처리하려고 시도하면,

```
Fixpoint eceval_fun (st: state) (c: com) : state :=
  match c with
  ..
  | <{ while b do c end}> =>
  if (beval st b)
    then ceval_fun st <{c ; while b do c end}>
    else st
  end.
```

다음의 오류가 발생한다: `Error: Cannot guess decreasing argument of fix.`

OCaml이나 Haskell 같은 언어에서는 이러한 구문을 허용하지만 Coq은 종료한다는 것이 보장되지 않는 구문은 절대로 허락하지 않는다.

Evaluation as a Relation

`st: state`가 `c: cmd`에 의하여 `st': state`로 변할 때 이를 다음과 같이 나타낸다. (영어로는, `c` *takes* `st` *to* `st'`.)

```
st =[ c ]=> st'
```

커맨드의 시맨틱스를 추론규칙 형식으로 나타내면 다음과 같다.

$$\frac{}{\texttt{st =[skip]=> st'}}\ (\text{E_Skip})$$

$$\frac{\texttt{aeval st a = n}}{\texttt{st =[x:=a]=> (x !-> n ; st)}}\ (\text{E_Asgn})$$

$$\frac{\texttt{st =[c1]=> st'} \quad \texttt{st =[c2]=> st''}}{\texttt{st =[c1;c2]=> st''}}\ (\text{E_Seq})$$

$$\frac{\texttt{beval st b = true} \quad \texttt{st =[c1]=> st'}}{\texttt{st =[if b then c1 else c2 end]=> st'}}\ (\text{E_IfTrue})$$

$$\frac{\texttt{beval st b = false} \quad \texttt{st =[c2]=> st''}}{\texttt{st =[if b then c1 else c2 end]=> st'}}\ (\text{E_IfFalse})$$

$$\frac{\texttt{beval st b = false}}{\texttt{st =[while b do c end]=> st}}\ (\text{E_WhileFalse})$$

$$\frac{\begin{array}{c}\texttt{beval st b = true} \\ \texttt{st =[c]=> st'} \\ \texttt{st' =[whild b do c end]=> st''}\end{array}}{\texttt{st =[while b do c end]=> st''}} \quad \text{(E_WhileTrue)}$$

형식 정의는 다음과 같다. 각 생성자의 경우에, 이 정의가 추론규칙 정의와 어떻게 대응되는지 세밀하게 확인해 보기 바란다.[3]

```
Reserved Notation
         "st '=[' c ']=>' st'"
         (at level 40, c custom com at level 99,
          st constr, st' constr at next level).

Inductive ceval : com -> state -> state -> Prop :=
  | E_Skip : forall st,
      st =[ skip ]=> st
  | E_Asgn  : forall st a n x,
      aeval st a = n ->
      st =[ x := a ]=> (x !-> n ; st)
  | E_Seq : forall c1 c2 st st' st'',
      st  =[ c1 ]=> st'  ->
      st' =[ c2 ]=> st'' ->
      st  =[ c1 ; c2 ]=> st''
  | E_IfTrue : forall st st' b c1 c2,
      beval st b = true ->
      st =[ c1 ]=> st' ->
      st =[ if b then c1 else c2 end]=> st'
  | E_IfFalse : forall st st' b c1 c2,
      beval st b = false ->
      st =[ c2 ]=> st' ->
      st =[ if b then c1 else c2 end]=> st'
  | E_WhileFalse : forall b st c,
      beval st b = false ->
      st =[ while b do c end ]=> st
  | E_WhileTrue : forall st st' st'' b c,
      beval st b = true ->
      st  =[ c ]=> st' ->
      st' =[ while b do c end ]=> st'' ->
      st  =[ while b do c end ]=> st''

  where "st =[ c ]=> st'" := (ceval c st st').
```

이렇게 시맨틱스를 관계로써 정의하면 while 루프를 다룰 수 있다는 장점이 있는 반면에, 표현의 값을 계산해 주지 않기 때문에 우리가 그 값을 예상하고, 그 다음에 그것이 맞다는 것을 증명해야 하는 부담이 있다.

간단한 예를 가지고 공부해 보자.

```
1 Example ceval_example1:
2   empty_st =[
```

[3] aexp의 값계산을 위하여 함수를 사용할 때는 그 함수를 aeval라고 이름지었고, 술어를 사용할 때는 그 술어를 aevalR라고 이름지었었다. com의 값계산에는 함수를 사용할 수 없으며, 술어의 이름은 cevalR로 부르면 될 것인데 SF는 이 술어에 ceval라는 이름을 주어 사용하고 있다.

```
3       X := 2;
4       if (X <= 1)
5         then Y := 3
6         else Z := 4
7       end
8    ]=> (Z !-> 4 ; X !-> 2).
9  Proof.
10   apply E_Seq with (X !-> 2).
11   - apply E_Asgn. simpl. reflexivity.
12   - apply E_IfFalse.
13     + simpl. reflexivity.
14     + apply E_Asgn. simpl. reflexivity.
15 Qed.
```

이 예에서 커맨드의 메인 생성자는 E_Seq이므로 증명의 처음인 라인 10에서 apply E_Seq를 사용한 것이다. 그 다음엔 apply E_Asgn과 apply E_IfFalse를 사용했다. 이 과정을 명확히 이해하기 위하여 이 예에서의 커맨드를 다음과 같이 풀어 쓰는 것이 도움된다.

```
Unset Printing Notations.
Definition example1com : com :=
  <{ X := 2;
     if (X <= 1)
       then Y := 3
       else Z := 4
     end }>.
Print example1com.

example1 = CSeq (CAsgn X 2)
                (CIf (BLe X 1)
                     (CAsgn Y 3)
                     (CAsgn Z 4))
: com

Set Printing Notations.
```

라인 10의 apply E_Seq는 5개의 인수 c1 c2 st st' st''를 필요로 하는데 이들 중에 4개는 Coq이 자동으로 찾아낸다.

```
c1   := X := 2
c2   := if X <= 1 then Y := 3 else Z := 4 end
st   := empty_st
st'  := (X !-> 2)
st'' := (Z !-> 4 ; X !-> 2)
```

물론 라인 10을 다음과 같이 아주 친절하게 쓸 수도 있다.

```
apply E_Seq with
  (c1 := <{ X := 2 }>) (c2 := <{ if X <= 1 then Y := 3 else Z := 4 end }>)
  (st:= empty_st) (st':= X !-> 2) (st'':= (Z !-> 4 ; X !-> 2)).
```

라인 10에 의하여 2개의 하위 고울이 생성된다.

```
Goal 1
(1 / 2) -------------------------
empty_st =[ X := 2 ]=> (X !-> 2)

Goal 2
(2 / 2) -------------------------
(X !-> 2) =[ if X <= 1 then Y := 3 else Z := 4 end ]=> (Z !-> 4; X !-> 2)
```

라인 11의 apply E_Asgn을 실행하면 고울 화면은 다음과 같이 된다.

```
aeval empty_st 2 = 2
```

이후의 과정은 생략한다.

다음의 연습문제에서 핵심 단계는 intermediate state를 정확히 지정해 주는 것이다.

```
Example ceval_example2:
  empty_st =[
    X := 0;
    Y := 1;
    Z := 2
  ]=> (Z !-> 2 ; Y !-> 1 ; X !-> 0).
Proof.
  apply E_Seq with (X !-> 0).
  - apply E_Asgn. reflexivity.
  - apply E_Seq with (Y !-> 1; X !-> 0).
    + apply E_Asgn. reflexivity.
    + apply E_Asgn. reflexivity.
Qed.
```

<u>연습문제</u> **13.2** 1부터 X까지의 합을 계산하는 Imp 프로그램 pub_to_n을 작성하여라. 다음의 정리를 증명할 수 있어야 한다.

```
Theorem pup_to_2_ceval :
  (X !-> 2) =[ pup_to_n ]=>
  (X !-> 0 ; Y !-> 3 ; X !-> 1 ; Y !-> 2 ; Y !-> 0 ; X !-> 2).
Proof. Admitted.
```

이 정리의 프랍을 보고 reverse-engineering 하여 프로그램을 작성하여라.

```
Definition pup_to_n : com :=
  <{ Y := 0;
     while ~ (X = 0) do
       Y := Y + X;
       X := X - 1
     end }>.
```

(풀이). 이 증명에서도 역시 intermediate state를 정확히 지정해 주는 것이 중요하다.

```
1 Theorem pup_to_2_ceval :
2   (X !-> 2) =[
3     pup_to_n
4   ]=> (X !-> 0 ; Y !-> 3 ; X !-> 1 ; Y !-> 2 ; Y !-> 0 ; X !-> 2).
5 Proof.
```

```
6    apply E_Seq with (Y !-> 0; X !-> 2).
7    - apply E_Asgn. reflexivity.
8    - apply E_WhileTrue with (X !-> 1 ; Y !-> 2 ; Y !-> 0; X !-> 2).
9      + reflexivity.
10     + apply E_Seq with (Y !-> 2; Y !-> 0; X !-> 2).
11       { (* ... *) }
12       { (* ... *) }
13     + apply E_WhileTrue with (X !-> 0; Y !-> 3; X !-> 1 ;
14                               Y !-> 2 ; Y !-> 0; X !-> 2).
15       { (* ... *) }
16       { apply E_Seq with (Y !-> 3; X !-> 1 ;
17                           Y !-> 2 ; Y !-> 0; X !-> 2).
18         - (* ... *)
19         - (* ... *) }
20       { apply E_WhileFalse. reflexivity. }
21 Qed.
```

Determinism of Evaluation

어떤 상태에 대하여 어떤 커맨드를 실행하면 그 결과는 유일하다. 이것을 다음 정리로 나타낼 수 있다. (이 정리는 ceval 관계는 one-to-many가 아님을, 즉 함수라는 것을 증명한다.)

```
1  Theorem ceval_deterministic: forall c st st1 st2,
2       st =[ c ]=> st1  ->
3       st =[ c ]=> st2 ->
4       st1 = st2.
5  Proof.
6    intros c st st1 st2 E1 E2.
7    generalize dependent st2.
8    induction E1; intros st2 E2; inversion E2; subst.
9    - (* E_Skip *) reflexivity.
10   - (* E_Asgn *) reflexivity.
11   - (* E_Seq *)
12     rewrite (IHE1_1 st'0 H1) in *.
13     apply IHE1_2. assumption.
14   - (* E_IfTrue, b evaluates to true *)
15       apply IHE1. assumption.
16   - (* E_IfTrue,  b evaluates to false (contradiction) *)
17       rewrite H in H5. discriminate.
18   - (* E_IfFalse, b evaluates to true (contradiction) *)
19       rewrite H in H5. discriminate.
20   - (* E_IfFalse, b evaluates to false *)
21       apply IHE1. assumption.
22   - (* E_WhileFalse, b evaluates to false *)
23     reflexivity.
24   - (* E_WhileFalse, b evaluates to true (contradiction) *)
25     rewrite H in H2. discriminate.
26   - (* E_WhileTrue, b evaluates to false (contradiction) *)
27     rewrite H in H4. discriminate.
28   - (* E_WhileTrue, b evaluates to true *)
29     rewrite (IHE1_1 st'0 H3) in *.
30     apply IHE1_2. assumption.  Qed.
```

13.7 Reasoning About Imp Programs

이 섹션은 기본적으로 연습문제 모음이다. Imp 프로그램을 분석하는 체계적인 방법은 다음 번 볼륨인 Programming Language Foundations에서 다룰 것이며, 여기서는 현재까지 배운 지식을 활용하여 몇몇 문제들을 풀어보기로 한다.

```
Print plus2. (*  <{ X := X + 2 }>: com *)
Check t_update_eq.
(** forall (A : Type) (m : total_map A) (x : string) (v : A),
    (x !-> v; m) x = v *)

Fact plus2_3_is_5 : forall (st: state),
  (X !-> 3) =[ plus2 ]=> st -> st X = 5.
Proof.
  intros st H.
  inversion H. subst. simpl.
  apply t_update_eq. (* or use [reflexivity] *)
Qed.

Compute (ceval_fun_no_while (X !-> 3) plus2) X. (* = 5: nat *)

Fact plus2_3_is_5' :
  (X !-> 3) =[ plus2 ]=> (X !-> 5).
Proof. Abort.
```

```
Theorem plus2_spec : forall (st st': state) (n: nat),
  st X = n ->
  st =[ plus2 ]=> st' ->
  st' X = n + 2.
Proof.
  intros st n st' HX Heval.
  inversion Heval. subst. simpl.
  apply t_update_eq.  (* or use reflexivity.*)
  Qed.
```

```
Print XtimesYinZ. (* < Z := X * Y >  *)

Theorem XtimesYinZ_spec : forall st x y st',
  st X = x ->
  st Y = y ->
  st =[ XtimesYinZ ]=> st' ->
  st' Z = x * y.
Proof.
  intros st x y st' HX HY Heval.
  inversion Heval.
  subst. clear Heval. simpl. reflexivity. Qed.
```

```
Theorem loop_never_stops : forall st st',
  ~(st =[ loop ]=> st').
Proof.
  intros st st'.
  unfold not. intros contra.
  unfold loop in contra.
```

```
7    remember <{ while true do skip end }> as loopdef
8             eqn:Heqloopdef.
9    induction contra; inversion Heqloopdef.
10   - rewrite H1 in H. inversion H.
11   - apply IHcontra2. rewrite H1, H2. reflexivity.
12 Qed.
```

```
1  Fixpoint no_whiles (c : com) : bool :=
2    match c with
3    | <{ skip }> =>
4        true
5    | <{ _ := _ }> =>
6        true
7    | <{ c1 ; c2 }> =>
8        andb (no_whiles c1) (no_whiles c2)
9    | <{ if _ then ct else cf end }> =>
10       andb (no_whiles ct) (no_whiles cf)
11   | <{ while _ do _ end }>  =>
12       false
13   end.
```

```
1  Inductive no_whilesR: com -> Prop :=
2  | nw_skip: no_whilesR < skip >
3  | nw_ass: forall x ex,
4      no_whilesR (< x := ex >)
5  | nw_seq: forall c1 c2,
6      no_whilesR c1 ->
7      no_whilesR c2 ->
8      no_whilesR (< c1 ; c2 >)
9  | nw_ifb: forall b c1 c2,
10     no_whilesR c1 ->
11     no_whilesR c2 ->
12     no_whilesR (< if b then c1 else c2 end >).
```

```
1  Theorem no_whiles_eqv:
2     forall (c: com), no_whiles c = true <-> no_whilesR c.
3  Proof.
4    intros. split.
5    - (* -> *) intros.
6      induction c.
7      + apply nw_skip.
8      + apply nw_ass.
9      + simpl in H.
10       apply andb_true_iff in H.
11       destruct H.
12       apply nw_seq.
13       { apply IHc1. apply H. }
14       { apply IHc2. apply H0. }
15     + simpl in H.
16       apply andb_true_iff in H.
17       destruct H.
18       apply nw_ifb.
19       { apply IHc1. apply H. }
```

```
      { apply IHc2. apply H0. }
    + simpl in H. inversion H.
  - (* <- *) intros.
    induction c; simpl.
    + reflexivity.
    + reflexivity.
    + inversion H.
      apply andb_true_iff.
      split.
      { apply IHc1. apply H2. }
      { apply IHc2. apply H3. }
    + inversion H.
      apply andb_true_iff.
      split.
      { apply IHc1. apply H2. }
      { apply IHc2. apply H4. }
    + inversion H.
Qed.
```

```
Theorem no_whiles_terminating: forall c st,
    no_whilesR c ->
    exists st', st =[ c ]=> st'.
Proof.
  intros.
  generalize dependent st.
  induction H; intros.
  - exists st.
    apply E_Skip.
  - exists (t_update st x (aeval st ex)).
    apply E_Asgn. reflexivity.
  - destruct (IHno_whilesR1 st).
    destruct (IHno_whilesR2 x).
    exists x0.
    apply (E_Seq c1 c2 st x x0).
    + apply H1.
    + apply H2.
  - destruct (beval st b) eqn:bval.
    + destruct (IHno_whilesR1 st).
      exists x.
      apply E_IfTrue.
      { exact bval. }
      { exact H1. }
    + destruct (IHno_whilesR2 st).
      exists x.
      apply E_IfFalse.
      { exact bval. }
      { exact H1. }
Qed.
```

```
Inductive sinstr : Type :=
| SPush (n : nat)
| SLoad (x : string)
| SPlus
```

```
| SMinus
| SMult.

Fixpoint s_execute (st : state) (stack : list nat)
  (prog : list sinstr) : list nat :=
  match prog with
  | [] => stack
  | (SPush n) :: prog' => s_execute st (n :: stack) prog'
  | (SLoad k) :: prog' => s_execute st ((st k) :: stack) prog'
  | SPlus :: prog' => s_execute st (((hd 0 (tl stack)) + (hd 0 stack)) :: (tl (tl stack)))
                              prog'
  | SMinus :: prog' => s_execute st (((hd 0 (tl stack)) - (hd 0 stack)) :: (tl (tl stack)))
                              prog'
  | SMult :: prog' => s_execute st (((hd 0 (tl stack)) * (hd 0 stack)) :: (tl (tl stack)))
                              prog'
  end.

Example s_execute1 :
     s_execute empty_st []
       [SPush 5; SPush 3; SPush 1; SMinus]
   = [2; 5].
Proof. simpl. reflexivity. Qed.

Example s_execute2 :
     s_execute (X !-> 3) [3;4]
       [SPush 4; SLoad X; SMult; SPlus]
   = [15; 4].
Proof. simpl. reflexivity. Qed.

Fixpoint s_compile (e : aexp) : list sinstr :=
  match e with
  | ANum x => [SPush x]
  | AId k => [SLoad k]
  | APlus a1 a2 => (s_compile a1) ++ (s_compile a2) ++ [SPlus]
  | AMinus a1 a2 => (s_compile a1) ++ (s_compile a2) ++ [SMinus]
  | AMult a1 a2 => (s_compile a1) ++ (s_compile a2) ++ [SMult]
  end.

Example s_compile1 :
  s_compile <{ X - (2 * Y) }>
  = [SLoad X; SPush 2; SLoad Y; SMult; SMinus].
Proof. simpl. reflexivity. Qed.
```

```
1  Theorem execute_app : forall st p1 p2 stack,
2    s_execute st stack (p1 ++ p2) =
3    s_execute st (s_execute st stack p1) p2.
4  Proof.
5    intros.
6    generalize dependent st.
7    generalize dependent stack.
8    generalize dependent p2.
9    induction p1; intros.
10   - reflexivity.
```

```
  - destruct a; simpl; apply IHp1.
Qed.
```

```
Lemma s_compile_correct_aux : forall st e stack,
  s_execute st stack (s_compile e) = aeval st e :: stack.
Proof.
  intros.
  generalize dependent stack.
  generalize dependent st.
  induction e; simpl; intros;
    try reflexivity;
    repeat rewrite execute_app;
    rewrite IHe1, IHe2;
    reflexivity.
Qed.

Theorem s_compile_correct : forall (st : state) (e : aexp),
  s_execute st [] (s_compile e) = [ aeval st e ].
Proof.
  intros.
  apply s_compile_correct_aux.
Qed.
```

```
Fixpoint beval_ss (st : state) (b : bexp) : bool :=
  match b with
  | < true >  => true
  | < false > => false
  | < a1 = a2 >  => (aeval st a1) =? (aeval st a2)
  | < a1 <> a2 > => negb ((aeval st a1) =? (aeval st a2))
  | < a1 <= a2 > => (aeval st a1) <=? (aeval st a2)
  | < a1 > a2 >  => negb ((aeval st a1) <=? (aeval st a2))
  | < ~ b1 >   => negb (beval_ss st b1)
  | < b1 && b2 > => if (beval_ss st b1)
                    then (beval_ss st b2)
                    else false
  end.

Theorem beval_beval_ss : forall st b,
  beval st b = beval_ss st b.
Proof.
  intros.
  destruct b; reflexivity.
Qed.
```

```
Inductive com : Type :=
  | CSkip
  | CBreak                        (* <--- NEW *)
  | CAsgn (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com).

Notation "'break'" := CBreak (in custom com at level 0).
```

```
Notation "'skip'"  :=
         CSkip (in custom com at level 0) : com_scope.
Notation "x := y"  :=
         (CAsgn x y)
            (in custom com at level 0, x constr at level 0,
             y at level 85, no associativity) : com_scope.
Notation "x ; y" :=
         (CSeq x y)
           (in custom com at level 90, right associativity) : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" :=
         (CIf x y z)
           (in custom com at level 89, x at level 99,
            y at level 99, z at level 99) : com_scope.
Notation "'while' x 'do' y 'end'" :=
         (CWhile x y)
            (in custom com at level 89, x at level 99, y at level 99) : com_scope.

Inductive result : Type :=
  | SContinue
  | SBreak.

Reserved Notation "st '=[' c ']=>' st' '/' s"
     (at level 40, c custom com at level 99, st' constr at next level).

Inductive ceval : com -> state -> result -> state -> Prop :=
  | E_Skip : forall st,
    st =[ CSkip ]=> st / SContinue
  | E_Break : forall st,
    st =[ CBreak ]=> st / SBreak
  | E_Ass : forall st a1 n x,
    aeval st a1 =  n ->
    st =[ x := a1 ]=> (t_update st x n) / SContinue
  | E_IfTrue : forall st st' b c1 c2 cont,
    beval st b = true ->
    st =[ c1 ]=> st' / cont ->
    st =[ if b then c1 else c2 end ]=> st' / cont
  | E_IfFalse : forall st st' b c1 c2 cont,
    beval st b = false ->
    st =[ c2 ]=> st' / cont ->
    st =[ if b then c1 else c2 end ]=> st' / cont
  | E_SeqBreak : forall st st' c1 c2,
    st =[ c1 ]=> st' / SBreak ->
    st =[ c1 ; c2 ]=> st' /SBreak
  | E_SeqContinue : forall st st' st'' c1 c2 prog,
    st =[ c1 ]=> st' / SContinue ->
    st' =[ c2 ]=> st'' / prog ->
    st =[ c1 ; c2 ]=> st'' / prog
  | E_WhileEnd : forall st b c,
    beval st b = false ->
    st =[ while b do c end ]=> st / SContinue
  | E_WhileLoopBreak : forall st st' b c,
    beval st b = true ->
    st =[ c ]=> st' / SBreak ->
```

```
      st =[ while b do c end ]=> st' / SContinue
  | E_WhileLoopContinue : forall st st' st'' b c,
      beval st b = true ->
      st =[ c ]=> st' / SContinue ->
      st' =[ while b do c end ]=> st'' / SContinue ->
      st =[ while b do c end ]=> st'' / SContinue

  where "st '=[' c ']=>' st' '/' s" := (ceval c st s st').
```

```
Theorem break_ignore : forall c st st' s,
     st =[ break; c ]=> st' / s ->
     st = st'.
Proof.
  intros.
  inversion H; subst.
  - inversion H5. reflexivity.
  - inversion H2.
Qed.

Theorem while_continue : forall b c st st' s,
  st =[ while b do c end ]=> st' / s ->
  s = SContinue.
Proof.
  intros.
  inversion H; reflexivity.
Qed.

Theorem while_stops_on_break : forall b c st st',
  beval st b = true ->
  st =[ c ]=> st' / SBreak ->
  st =[ while b do c end ]=> st' / SContinue.
Proof.
  intros.
  inversion H0; subst;
  apply E_WhileLoopBreak; assumption.
Qed.

Theorem seq_continue : forall c1 c2 st st' st'',
  st =[ c1 ]=> st' / SContinue ->
  st' =[ c2 ]=> st'' / SContinue ->
  st =[ c1 ; c2 ]=> st'' / SContinue.
Proof. Admitted.

Theorem seq_stops_on_break : forall c1 c2 st st',
  st =[ c1 ]=> st' / SBreak ->
  st =[ c1 ; c2 ]=> st' / SBreak.
Proof. Admitted.

(** **** Exercise: 3 stars, advanced, optional (while_break_true) *)
Theorem while_break_true : forall b c st st',
  st =[ while b do c end ]=> st' / SContinue ->
  beval st' b = true ->
  exists st'', st'' =[ c ]=> st' / SBreak.
```

```
45 Proof.
46   intros.
47   remember (<{ while b do c end }>) as loopdef eqn:Eq.
48   induction H; try inversion Eq; try subst.
49   - rewrite H0 in H. discriminate H.
50   -  exists st. exact H1.
51   - apply IHceval2.
52     + reflexivity.
53     + exact H0.
54 Qed.
55
56 Theorem ceval_deterministic: forall (c:com) st st1 st2 s1 s2,
57      st =[ c ]=> st1 / s1 ->
58      st =[ c ]=> st2 / s2 ->
59      st1 = st2 /\ s1 = s2.
60 Proof. Admitted.
61
```

참고 문헌

[1] *Intuitionistic Type Theory*, Per Martin-Löf (1984), Bibliopolis

[2] *Interactive Theorem Proving and Applications*, Yves Bertot, Pierre Castéran (2004), Springer

[3] *Type Theory and Formal Proof*, Rob Nederpelt, Herman Geuvers (2014), Cambridge University Press

[4] *Software Foundations*, Benjamin C. Pierce et al. [softwarefoundations.cis.upenn.edu]

[5] *The Coq Proof Assistant*, [https://coq.inria.fr]

[6] *The Coq Reference Manual*, [https://coq.inria.fr/distrib/current/refman]

찾아보기

Abort, 17
add_0_r, 32
admit
    postpone the proof of assertion, 35
Admitted, 17
and
    logical connective, 102
andb, 16
anonymous function, 69
apply, 13
    constructor as argument, 137
    전칭문 가설에 대한, 57
    조건문 가설에 대한, 80, **87**
apply with, 81
argument
    of application of tactic, 86
Arguments
    implicit type, 66
assert, 33
    증명을 뒤로 미룸, 35
Axiom, 128

base case, 32
BHK
    interpretation, 227
bool, 15
Boolean, 15
bullet
    in case analysis, 26

case analysis, 156
    proof by, 25
Check, 7
    for theorems, 25
Church numeral, 75
clear, 189
combine, 68
comment, 8
Compute, 7
    for function evaluation, 10
conditional, 22
Conjecture, 136
constructor, 8
    polymorphic, 63
    with parameter, 18
context, 12
    of goal, 23
coqide, 3
Currying, 20
currying, 72

Definition
    of constant, 7
    of function, 9
dependent type, 209
destruct, 25
    conjunctive hypothesis, 105
    disjunctive hypothesis, 106
    existential hypothesis, 111
    intro patterns, 46
    of evidence, 148
    on False hypothesis, 107
    on compound expression, 95
    on term expression, 42
    one constructor case, 59
determiner, 23
discriminate, 85
disjointness
    of constructors, 83
double, 41
double negation elimination, 126

elim
    existential hypothesis, 114
End, 18
eqb, 21
eqb_eq, 123
eqb_refl, 33
eqb_true, 92
Eval

compute in *exp*, 9
Even
unary predicate, 111
even
boolean function, 19
evidence, 137
constructor, 145
ex falso, 218
exact, 23, 57
exfalso, 108
exists, 23, 111
existsb, 99

f_equal, 86
Fact, 15
Fail, 65
False, 107
false, 15
falsum, 218
filter, 69
Fixpoint, 19
flat_map, 70
fold, 71
forall, 23
forallb, 99
fun
anonymous function, 69

generalize dependent, 92
on hypothesis, 167
goal
in Coq Goals window, 12
goal prop, 12

higher-order function, 68

I, 109
if-then-else, 16
iff, 109
In, 114
induction
on evidence, 151
induction goal
for inductively defined type, 234
induction hypothesis, 32
for inductively defined type, 233
induction principle
for inductively defined proposition, 234
for inductively defined type, 229
Inductive
define predicates, 136–140
enumerated type, 7
type defn., polymorphic, 63
type defn., recursive, 19, 47
type defn. with parameter, 18
inductive case, 32
inhabitant, 9
injection, 83
get multiple equalities, 84
injectivity
of constructors, 83
instantiate
univerally bound variable, 23
intro, 22
compared to intros, 14
intro pattern, 27, 46
intros, 22, **24**
existential antecedent, 111
inversion
lemma for evidence, 149
on evidence, 148
inversion
tactic for evidence, 149

judgement
hypothetical, 14, 23

Law of Excluded Middle, 130
leb, 21
left, 106
LEM, 130
Lemma, 15
let ... in ..., 70
let .. in .., 280
list
of nat, 47
local function, 70
Locate, 54
ltb, 21

map
dictionary, 251

  partial, 59, 256
  total, 252
map, 70
match
  regex and string, 168
match, 9
  appearing in goals, 42
  term expression, 40
exp, 20
minus, 20
mult, 20
module, 18
Module, 18

natlist, 47
negb, 16
NNPP
  double negation elimination, 126
Notation, 17
nth_error
  polymorphic, 68

opam, 2
option
  for nat, 58
  polymorphic, 68
or
  logical connective, 106
orb, 16
outermost-leftmost, 34

pair
  polymorphic, 67
partial map, 59, 256
particularize
  univerally bound variable, 23
permutation
  binary relation on lists, 142
plus, 20
plus_n_Sm, 33
polymorphic constructor, 63
polymorphic pair, 67
polymorphic type, 63
polymorphism, 63
pose proof .. as .., 201
Print, 7
  for function definition, 10
  for theorems, 207
proof
  script, 12
  term, object, 12
Proof, 12
proof object, 206
proof script, 206, 207
proof term, 141, 206
  how to see, 207
Prop, 9
  False, 107
  True, 109
proposition, 10

Qed, 12
quantifier
  universal, existential, 23

reflect
  improve proof by reflection, 193
reflection
  proof by, 124
reflexivity
  tactics, 15, 72
regex, 168
regular expression, 168
remember, 178
repeat split, 182
replace
  tactics, 35
Reserved
  Notation, 139
revert, 225
rewrite, 24
  .. with .., 35
  automatic substitution, 25
  for iff, 110
  order of applications, 34
right, 106
Rule-C, 111

scope
  of quantifier, 23
Search, 53
Set, 8

setoid, 110
Show Proof, 207
simpl, 15, **17**
    simpl in, 86
    with destruct, unfold, 72
specialize
    (H2 H1), 88
    .. with .., 88
split
    for conjunctive goals, 104
symmetry, 57
syntactic sugar, 101, 210

tactic, 22
    apply, 13
        constructor as argument, 137
        전칭문 가설에 대한, 57
        조건문 가설에 대한, 80, **87**
    apply with, 81
    assert, 33
        증명을 뒤로 미룸, 35
    clear, 189
    destruct, 25
        conjunctive hypothesis, 105
        disjunctive hypothesis, 106
        existential hypothesis, 111
        intro patterns, 46
        on False hypothesis, 107
        on compound expression, 95
        on term expression, 42
        one constructor case, 59
    discriminate, 85
    elim
        existential hypothesis, 114
    exact, 23, 57
    exfalso, 108
    exists, 111
    f_equal, 86
    generalize dependent, 92
        on hypothesis, 167
    induction, 32
    injection, 83
        get multiple equalities, 84
    intro, 22
        compared to intros, 14
    intros, 22, **24**
        existential antecedent, 111
    inversion
        for evidence, 149
    left, 106
    pose proof .. as .., 201
    reflexivity, 15, 72
    remember, 178
    repeat split, 182
    replace, 35
    revert, 225
    rewrite, 24
        .. with .., 35
        automatic substitution, 25
        for iff, 110
        order of applications, 34
    right, 106
    simpl, 15, **17**
        simpl in, 86
        with destruct, unfold, 72
    specialize
        (H2 H1), 88
        .. with .., 88
    split
        for conjunctive goals, 104
    symmetry, 57
    transitivity, 82
    unfold, 93
        on function, 72
        on proposition, 13
tactical, 263
target
    of application of tactic, 86
term, 8
Theorem, 15
total map, 252
transitive closure, 139
transitivity, 82
True, 109
true, 15
type, 7
    polymorphic, 63
type inference, 66

unfold, 93
    on function, 72
    on proposition, 13

unit test, 16
universal proposition, 22

VsCoq, 2

zip, 68

고울
    Coq Goals 화면에 나타난, 12
고울 프랍, 12
고차 함수, 68
귀납 고울
    귀납적으로 정의된 타입에 대한, 234
귀납 원리
    귀납적으로 정의된 타입에 대한, 229
    귀납적으로 정의된 프랍에 대한, 234
귀납가설, 32
    귀납적으로 정의된 타입에 대한, 233
귀납단계, 32
기저단계, 32

다형 생성자, 63
다형 순서쌍, 67
다형 타입, 63
다형성, 63
단사성
    생성자의, 83
레겍스, 168
리스트
    자연수의, 47
매치
    레겍스와 문자열, 168
모듈, 18
배타성
    생성자 간의, 83
범위
    한정사의, 23
부분맵, 256
부울리언, 15
불릿
    분할 증명에서 사용되는, 26

생성자, 8
    다형, 63
생성자별 분석, 156
설탕구문, 101, 210
순서쌍
    다형, 67
에비던스, 137
의존 타입, 209
익명 함수, 69
인수
    책략 적용의, 86
저지먼트
    가설, 23
전맵, 252
전칭문, 22
정규표현식, 168
조건문, 22
증거 생성자, 145
증거항, 141, 206
    보는 방법, 207
증명
    경우로 나누기, 25
    증거항, 12
    증명 스크립트, 12
증명 스크립트, 206, 207
지역 함수, 70
집합, 7
책략, 22
쳐지 뉴머럴, 75
추이 폐포, 139
커링, 72
컨텍스트, 12
    증명 고울의, 23
코멘트, 8

타겟
    책략 적용의, 86
타입
    다형, 63
타입 추론, 66
택티컬, 263
특정
    묶인변수를 ..하다, 23
표현항, 8
프랍, 10, 205
한정기호
    전칭, 존재, 23
한정사, 23