Discrete Mathematics

Jeremy Siek

Spring 2010

Outline of Lecture 1

1. Course Information
2. Overview of Discrete Mathematics

Course Information

- Class web page:

  `http://ecee.colorado.edu/~siek/ecen3703/spring10`
- Textbooks:
  - *Discrete Mathematics and its Applications, 6th Edition*, by Rosen. (At the CU bookstore.)
  - *A Tutorial Introduction to Structured Isar Proofs*, by Nipkow. (Available online.)
  - *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, by Nipkow, Paulson, and Wenzel. (Available online.)
  - *How to Prove It: A Structured Approach*, by Daniel J. Velleman.
- Grading:

Quizzes	30%
Midterm exam	30%
Final exam	40%

Course Information: Homework

- There are weekly homework assignments.
- The quizzes and exams are based on the homework.
- Every students gets a personal tutor named Isabelle. Isabelle is a logic language, a programming language, and a most importantly, a proof checker.
  `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`
- You know your proofs are correct when you convince Isabelle.

Discrete

Mathematics

Mathematics

- What is Math anyways?

Mathematics

- What is Math anyways?
- Is it the study of numbers?

Mathematics

- What is Math anyways?
- Is it the study of numbers?
- Mathematics is actually much more broad.

Mathematics

- What is Math anyways?
- Is it the study of numbers?
- Mathematics is actually much more broad.

Definition

Mathematics is the study of any **truth** regarding **well-defined** concepts.

Numbers are just one kind of well-defined concept.

Discrete

Definition

Something is *discrete* if is it composed of distinct, separable parts. (In contrast to continuous.)

Discrete	**Continuous**
integers	real numbers
graphs	rational numbers
state machines	differential equations
digital computer	radios
quantum physics	Newtonian physics

Definition

Discrete Mathematics is the study of any truth regarding discrete entities.

- That's pretty broad. So what is it really?
- Discrete math is the foundation for the rigorous understanding of computer systems.

A Discrete Problem: Sudoku

7		3		9				2
	1			3	6		7	9
			1	8		3	6	
	5			7			9	3
6	4					2		
3					2	7	1	
8	2	6	7		4	9	3	
	7	4	3			8	2	
		5	8	2	1		4	

- What are the rules of Sudoku?

A Discrete Problem: Sudoku

7		3		9				2
	1			3	6		7	9
			1	8		3	6	
	5			7			9	3
6	4					2		
3					2	7	1	
8	2	6	7		4	9	3	
	7	4	3			8	2	
		5	8	2	1		4	

- What are the rules of Sudoku?
- Spend the next few minutes filling in this board.

A Discrete Problem: Sudoku

7		3		9				2
	1			3	6		7	9
			1	8		3	6	
	5			7			9	3
6	4					2		
3					2	7	1	
8	2	6	7		4	9	3	
	7	4	3			8	2	
		5	8	2	1		4	

- What are the rules of Sudoku?
- Spend the next few minutes filling in this board.
- Write down the rules of Sudoku on a sheet of paper.

A Discrete Problem: Sudoku

7		3		9				2
	1			3	6		7	9
			1	8		3	6	
	5			7			9	3
6	4					2		
3					2	7	1	
8	2	6	7		4	9	3	
	7	4	3			8	2	
		5	8	2	1		4	

- What are the rules of Sudoku?
- Spend the next few minutes filling in this board.
- Write down the rules of Sudoku on a sheet of paper.
- Pass your paper to the person on your right. Are the rules that you've been passed correct? If not, give an example.

Abstracting Sudoku

7		3		9				2
	1			3	6		7	9
			1	8		3	6	
	5			7			9	3
6	4					2		
3					2	7	1	
8	2	6	7		4	9	3	
	7	4	3			8	2	
		5	8	2	1		4	

- What aspects of the game of Sudoku don't really matter?
- What could you change such that an expert Sudoku player would immediately be an expert of the modified game?
- What aspects of the game really matter?

Sudoku Solver

7		3		9				2
	1			3	6		7	9
			1	8		3	6	
	5			7			9	3
6	4					2		
3					2	7	1	
8	2	6	7		4	9	3	
	7	4	3			8	2	
		5	8	2	1		4	

- Write down a pseudo-code algorithm for solving Soduku.
- What data structures did you use?
- What kind of algorithm did you use?
- Does your algorithm always solve the puzzle?
- How long does your algorithm take to finish in the worst case?

Why Study Discrete Mathematics?

- It's the basic **language** used to discuss computer systems. You need to learn the language if you want to converse with other computer professionals.
- It's a **toolbox** full of the problem-solving techniques that you will use over and over in your career.
- But best of all, studying discrete math will **enhance your mind**, turning it into a high-precision machine!

Uses of Discrete Math are Everywhere

- Circuit design
- Computer architecture
- Computer networks
- Operating systems
- Programming: algorithms and data structures
- Programming languages
- Computer security, encryption
- Error correcting codes
- Graphics algorithms, game engines
- ...

Themes in Discrete Math

Mathematical Reasoning: read, understand, and create precise arguments.

Discrete Structures: model discrete systems and study their properties.

Algorithmic Thinking: create algorithms, verify that they work, analyze their time and space requirements.

Combinatorial Analysis: counting (not always as easy as it sounds!)

Advice

- Read in advance.
- Do the homework.
- Form a study group.
- Form an intense love/hate relationship with Isabelle.

Outline of Lecture 2

1. Propositional Logic
2. Syntax and Meaning of Propositional Logic

Logic

- Logic defines the ground rules for establishing truths.
- Mathematical logic spells out these rules in complete detail, defining what constitutes a *formal proof*.
- Learning mathematical logic is a good way to learn logic because it puts you on a firm foundation.
- Writing formal proofs in mathematical logic is a lot like computer programming. The rules of the game are clearly defined.

Propositional Logic

- Propositional logic is a language that abstracts away from content and focuses on the logical connectives.
- Uppercase letters like `P` and `Q` are *meta-variables* that are placeholders for propositions.
- The following rules define what is a *proposition*.

Propositional Logic

- Propositional logic is a language that abstracts away from content and focuses on the logical connectives.
- Uppercase letters like `P` and `Q` are *meta-variables* that are placeholders for propositions.
- The following rules define what is a *proposition*.
  - A propositional variable (lowercase letters `p, q, r`) is a proposition. These variables model true/false statements.

Propositional Logic

- Propositional logic is a language that abstracts away from content and focuses on the logical connectives.
- Uppercase letters like `P` and `Q` are *meta-variables* that are placeholders for propositions.
- The following rules define what is a *proposition*.
  - A propositional variable (lowercase letters `p, q, r`) is a proposition. These variables model true/false statements.
  - The negation of a proposition `P`, written `¬ P`, is a proposition.

Propositional Logic

- Propositional logic is a language that abstracts away from content and focuses on the logical connectives.
- Uppercase letters like `P` and `Q` are *meta-variables* that are placeholders for propositions.
- The following rules define what is a *proposition*.
  - A propositional variable (lowercase letters `p, q, r`) is a proposition. These variables model true/false statements.
  - The negation of a proposition `P`, written `¬ P`, is a proposition.
  - The conjunction (and) of two propositions, written `P ∧ Q`, is a proposition.

Propositional Logic

- Propositional logic is a language that abstracts away from content and focuses on the logical connectives.
- Uppercase letters like `P` and `Q` are *meta-variables* that are placeholders for propositions.
- The following rules define what is a *proposition*.
  - A propositional variable (lowercase letters `p, q, r`) is a proposition. These variables model true/false statements.
  - The negation of a proposition `P`, written `¬ P`, is a proposition.
  - The conjunction (and) of two propositions, written `P ∧ Q`, is a proposition.
  - The disjunction (or) of two propositions, written `P ∨ Q`, is a proposition.

Propositional Logic

- Propositional logic is a language that abstracts away from content and focuses on the logical connectives.
- Uppercase letters like `P` and `Q` are *meta-variables* that are placeholders for propositions.
- The following rules define what is a *proposition*.
  - A propositional variable (lowercase letters `p, q, r`) is a proposition. These variables model true/false statements.
  - The negation of a proposition `P`, written $\neg$ `P`, is a proposition.
  - The conjunction (and) of two propositions, written `P` $\land$ `Q`, is a proposition.
  - The disjunction (or) of two propositions, written `P` $\lor$ `Q`, is a proposition.
  - The conditional statement (implies), written `P` $\longrightarrow$ `Q`, is a proposition.

Propositional Logic

- Propositional logic is a language that abstracts away from content and focuses on the logical connectives.
- Uppercase letters like `P` and `Q` are *meta-variables* that are placeholders for propositions.
- The following rules define what is a *proposition*.
  - A propositional variable (lowercase letters `p, q, r`) is a proposition. These variables model true/false statements.
  - The negation of a proposition `P`, written `¬ P`, is a proposition.
  - The conjunction (and) of two propositions, written `P ∧ Q`, is a proposition.
  - The disjunction (or) of two propositions, written `P ∨ Q`, is a proposition.
  - The conditional statement (implies), written `P ⟶ Q`, is a proposition.
  - The Boolean values `True` and `False` are propositions.

Propositional Logic

- Different authors include different logical connectives in their definitions of Propositional Logic. However, these differences are not important.
- In each case, the missing connectives can be defined in terms of the connectives that are present.
- For example, I left out exclusive or, $\texttt{P} \oplus \texttt{Q}$, but

$$\texttt{P} \oplus \texttt{Q} = (\texttt{P} \land \neg\ \texttt{Q}) \lor \neg\ \texttt{P} \land \texttt{Q}$$

Propositional Logic

- How expressive is Propositional Logic?
- Can you write down the rules for Sudoku in Propositional Logic?

Propositional Logic

- How expressive is Propositional Logic?
- Can you write down the rules for Sudoku in Propositional Logic?
- It's rather difficult if not impossible to express the rules of Sudoku in Propositional Logic.
- But Propositional Logic is a good first step towards more powerful logics.

Meaning of Propositions

- A *truth assignment* maps propositional variables to True or False. The following is an example:

$$A \equiv \{p \mapsto \texttt{True}, q \mapsto \texttt{False}, r \mapsto \texttt{True}\}$$

$$A(p) = \texttt{True} \qquad A(q) = \texttt{False} \qquad A(r) = \texttt{True}$$

- The meaning of a proposition is a function from truth assignments to True or False. We use the notation $[\![P]\!]$ for the meaning of proposition P.

$$[\![p]\!](A) = A(p)$$

$$[\![\neg P]\!](A) = \begin{cases} \texttt{True} & \text{if } [\![P]\!](A) = \texttt{False} \\ \texttt{False} & \text{otherwise} \end{cases}$$

Meaning of Propositions, cont'd

$$[\![P \wedge Q]\!](A) = \begin{cases} \texttt{True} & \text{if } [\![P]\!](A) = \texttt{True}, [\![Q]\!](A) = \texttt{True} \\ \texttt{False} & \text{otherwise} \end{cases}$$

$$[\![P \vee Q]\!](A) = \begin{cases} \texttt{False} & \text{if } [\![P]\!](A) = \texttt{False}, [\![Q]\!](A) = \texttt{False} \\ \texttt{True} & \text{otherwise} \end{cases}$$

$$[\![P \longrightarrow Q]\!](A) = \begin{cases} \texttt{False} & \text{if } [\![P]\!](A) = \texttt{True}, [\![Q]\!](A) = \texttt{False} \\ \texttt{True} & \text{otherwise} \end{cases}$$

Example Propositions

Suppose $A = \{p \mapsto \texttt{True}, q \mapsto \texttt{False}\}$.

Example Propositions

Suppose $A = \{p \mapsto \texttt{True}, q \mapsto \texttt{False}\}$.

- $[\![p]\!](A) = \texttt{True}$

Example Propositions

Suppose $A = \{p \mapsto \texttt{True}, q \mapsto \texttt{False}\}$.

- $[\![p]\!](A) = \texttt{True}$
- $[\![q]\!](A) = \texttt{False}$

Example Propositions

Suppose $A = \{p \mapsto \texttt{True}, q \mapsto \texttt{False}\}$.

- $[\![p]\!](A) = \texttt{True}$
- $[\![q]\!](A) = \texttt{False}$
- $[\![p \wedge p]\!](A) = \texttt{True}$

Example Propositions

Suppose $A = \{p \mapsto \texttt{True}, q \mapsto \texttt{False}\}$.

- $[\![p]\!](A) = \texttt{True}$
- $[\![q]\!](A) = \texttt{False}$
- $[\![p \land p]\!](A) = \texttt{True}$
- $[\![p \land q]\!](A) = \texttt{False}$

Example Propositions

Suppose $A = \{p \mapsto \texttt{True}, q \mapsto \texttt{False}\}$.

- $[\![p]\!](A) = \texttt{True}$
- $[\![q]\!](A) = \texttt{False}$
- $[\![p \wedge p]\!](A) = \texttt{True}$
- $[\![p \wedge q]\!](A) = \texttt{False}$
- $[\![p \vee q]\!](A) = \texttt{True}$

Example Propositions

Suppose $A = \{p \mapsto \texttt{True}, q \mapsto \texttt{False}\}$.

- $[\![p]\!](A) = \texttt{True}$
- $[\![q]\!](A) = \texttt{False}$
- $[\![p \land p]\!](A) = \texttt{True}$
- $[\![p \land q]\!](A) = \texttt{False}$
- $[\![p \lor q]\!](A) = \texttt{True}$
- $[\![p \longrightarrow p]\!](A) = \texttt{True}$

Example Propositions

Suppose $A = \{p \mapsto \texttt{True}, q \mapsto \texttt{False}\}$.

- $[\![p]\!](A) = \texttt{True}$
- $[\![q]\!](A) = \texttt{False}$
- $[\![p \wedge p]\!](A) = \texttt{True}$
- $[\![p \wedge q]\!](A) = \texttt{False}$
- $[\![p \vee q]\!](A) = \texttt{True}$
- $[\![p \longrightarrow p]\!](A) = \texttt{True}$
- $[\![q \longrightarrow p]\!](A) = \texttt{True}$

Example Propositions

Suppose $A = \{p \mapsto \texttt{True}, q \mapsto \texttt{False}\}$.

- $[\![p]\!](A) = \texttt{True}$
- $[\![q]\!](A) = \texttt{False}$
- $[\![p \wedge p]\!](A) = \texttt{True}$
- $[\![p \wedge q]\!](A) = \texttt{False}$
- $[\![p \vee q]\!](A) = \texttt{True}$
- $[\![p \longrightarrow p]\!](A) = \texttt{True}$
- $[\![q \longrightarrow p]\!](A) = \texttt{True}$
- $[\![p \longrightarrow q]\!](A) = \texttt{False}$

Example Propositions

Suppose $A = \{p \mapsto \texttt{True}, q \mapsto \texttt{False}\}$.

- $[\![p]\!](A) = \texttt{True}$
- $[\![q]\!](A) = \texttt{False}$
- $[\![p \land p]\!](A) = \texttt{True}$
- $[\![p \land q]\!](A) = \texttt{False}$
- $[\![p \lor q]\!](A) = \texttt{True}$
- $[\![p \longrightarrow p]\!](A) = \texttt{True}$
- $[\![q \longrightarrow p]\!](A) = \texttt{True}$
- $[\![p \longrightarrow q]\!](A) = \texttt{False}$
- $[\![(p \lor q) \longrightarrow q]\!](A) = \texttt{False}$

Tautologies

Definition

A *tautology* is a proposition that is true in any truth assignment.

Examples:

- $p \longrightarrow p$
- $q \vee \neg q$
- $(p \wedge q) \longrightarrow (p \vee q)$

There are two ways to show that a proposition is a tautology:

1. Check the meaning of the proposition for every possible truth assignment. This is called *model checking*.
2. Contruct a *proof* that the proposition is a tautology.

Model Checking

- One way to simplify the checking is to only consider truth assignments that include the variables that matter. For example, to check $p \longrightarrow p$, we only need to consider two truth assignments.
  1. $A_1 = \{p \mapsto \texttt{True}\}$, $[\![p \longrightarrow p]\!](A_1) = \texttt{True}$
  2. $A_2 = \{p \mapsto \texttt{False}\}[\![p \longrightarrow p]\!](A_2) = \texttt{True}$
- However, in real systems there are many variables, and the number of possible truth assignments grows quickly: it is 2^n for n variables.
- There are many researchers dedicated to discovering algorithms that speed up model checking.

Stuff to Rememeber

Propositional Logic:

- The kind of propositions.
- The meaning of propositions.
- How to check that a proposition is a tautology.

Outline of Lecture 3

1. Proofs and Isabelle
2. Proof Strategy, Forward and Backwards Reasoning
3. Making Mistakes

Theorems and Proofs

- In the context of propositional logic, a theorem is just a tautology.
- In this course, we'll be writing theorems and their proofs in the Isabelle/Isar proof language.
- Here's the syntax for a theorem in Isabelle/Isar.

  theorem `"P"`
  proof -
  $\quad step\ 1$
  $\quad step\ 2$
  $\quad \vdots$
  $\quad step\ n$
  qed

- Each step applies an *inference rule* to establish the truth of some proposition.

Inference Rules

- When applying inference rules, use the keyword **have** to establish intermediate truths and use the keyword **show** to conclude the surrounding theorem or sub-proof.
- Most inference rules can be categorized as either an introduction or elimination rule.
- *Introduction rules* are for creating bigger propositions.
- *Elimination rules* are for using propositions.
- We write "L_i *proves* P" if there is a preceeding step or assumption in the proof that is labeled L_i and whose proposition is P.

Introduction Rules

And If L_i proves P and L_j proves Q, then write

from L_i L_j **have** L_k: `"P ∧ Q"` ..

Or (1) If L_i proves P, then write

from L_i **have** L_k: `"P ∨ Q"` ..

Or (2) If L_i proves Q, then write

from L_i **have** L_k: `"P ∨ Q"` ..

Implies

have L_k: `"P ⟶ Q"`
proof
  assume L_i: `"P"`
  $\vdots$
  $\cdots$ **show** `"Q"` $\cdots$
qed

Introduction Rules, cont'd

Not

have L_k: `"¬ P"`
proof
  assume L_i: `"P"`
  $\vdots$
  $\cdots$ **show** `"False"` $\cdots$
qed

Hint: The Appendix of our text *Isabelle/HOL – A Proof Assistant for Higher-Order Logic* lists the logical connectives, such as $\longrightarrow$ and $\neg$, and for each of them gives two ways to input them as ASCI text. If you use Emacs (or XEmacs) to edit your Isabelle files, then the `x-symbol` package can be used to display the logic connectives in their traditional form.

Using Assumptions

- Sometimes the thing you need to prove is already an assumption. In this case your job is really easy!
- If L_i proves P, write

  from L_i **have** `"P"` .

Example Proof

```
theorem "p ⟶ p"
proof -
  show "p ⟶ p"
  proof
    assume 1: "p"
    from 1 show "p" .
  qed
qed
```

Instead of **proof** -, you can apply the introduction rule right away.

```
theorem "p ⟶ p"
proof
  assume 1: "p"
  from 1 show "p" .
qed
```

Exercise

theorem `"p` $\longrightarrow$ `(p` $\wedge$ `p)"`

Solution

```
theorem "p ⟶ (p ∧ p)"
proof
  assume 1: "p"
  from 1 1 show "p ∧ p" ..
qed
```

Elimination Rules

$\boxed{\text{And (1)}}$ If L_i proves $P \wedge Q$, then write

from L_i **have** L_k: `"P"` ..

$\boxed{\text{And (2)}}$ If L_i proves $P \wedge Q$, then write

from L_i **have** L_k: `"Q"` ..

$\boxed{\text{Or}}$ If L_i proves $P \vee Q$, then write

note L_i
moreover { **assume** L_j: `"P"`
$\vdots$
$\cdots$ **have** `"R"` $\cdots$
} **moreover** { **assume** L_m: `"Q"`
$\vdots$
$\cdots$ **have** `"R"` $\cdots$
} **ultimately have** L_k: `"R"` ..

Elimination Rules, cont'd

Implies If L_i proves $P \longrightarrow Q$ and L_j proves P, then write
from L_i L_j **have** L_k: `"Q"` ..
(This rule is known as *modus ponens*.)

Not If L_i proves $\neg P$ and L_j proves P, then write
from L_i L_j **have** L_k: `"Q"` ..

False If L_i proves `False`, then write
from L_i **have** L_k: `"P"` ..

Example Proof

```
theorem "(p ∧ q) ⟶ (p ∨ q)"
proof
  assume 1: "p ∧ q"
  from 1 have 2: "p" ..
  from 2  show "p ∨ q" ..
qed
```

Another Proof

```
theorem "(p ∨ q) ∧ (p ⟶ r) ∧ (q ⟶ r) ⟶ r"
proof
  assume 1: "(p ∨ q) ∧ (p ⟶ r) ∧ (q ⟶ r)"
  from 1 have 2: "p ∨ q" ..
  from 1 have 3: "(p ⟶ r) ∧ (q ⟶ r)" ..
  from 3 have 4: "p ⟶ r" ..
  from 3 have 5: "q ⟶ r" ..
  note 2
  moreover { assume 6: "p"
    from 4 6 have "r" ..
  } moreover { assume 7: "q"
    from 5 7 have "r" ..
  } ultimately show "r" ..
qed
```

Exercise

theorem `"(p` $\longrightarrow$ `q)` $\wedge$ `(q` $\longrightarrow$ `r)` $\longrightarrow$ `(p` $\longrightarrow$ `r)"`

Solution

```
theorem "(p ⟶ q) ∧ (q ⟶ r) ⟶ (p ⟶ r)"
proof
  assume 1: "(p ⟶ q) ∧ (q ⟶ r)"
  from 1 have 2: "p ⟶ q" ..
  from 1 have 3: "q ⟶ r" ..
  show "p ⟶ r"
  proof
    assume 4: "p"
    from 2 4 have 5: "q" ..
    from 3 5 show "r" ..
  qed
qed
```

Forward and Backwards Reasoning

$\boxed{\text{And-Intro (forward)}}$ If L_i proves P and L_j proves Q, then write

from L_i L_j **have** L_k: `"P ∧ Q"` ..

$\boxed{\text{And-Intro (backwards)}}$

have L_k: `"P ∧ Q"`
proof

$\vdots$

$\cdots$ **show** `"P"` $\cdots$

next

$\vdots$

$\cdots$ **show** `"Q"` $\cdots$

qed

Forward and Backwards Reasoning, cont'd

$\boxed{\text{Or-Intro (1) (forwards)}}$ If L_i proves P, then write

from L_i **have** L_k: `"P ∨ Q"` ..

$\boxed{\text{Or-Intro (1) (backwards)}}$

have L_k: `"P ∨ Q"`
proof (rule disjI1)

$\vdots$

$\cdots$ **show** `"P"` $\cdots$

qed

Forward and Backwards Reasoning, cont'd

Or-Intro (2) (forwards) If L_i proves Q, then write

from L_i **have** L_k: `"P ∨ Q"` ..

Or-Intro (2) (backwards)

have L_k: `"P ∨ Q"`
proof (rule disjI2)

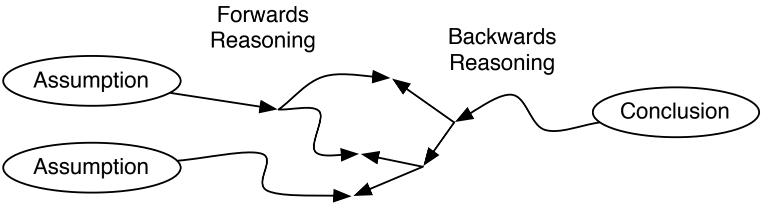
$\vdots$

$\cdots$ **show** `"Q"` $\cdots$

qed

Strategy

- Let the proposition you're trying to prove guide your proof.
- Find the top-most logical connective.
- Apply the introduction rule, backwards, for that connective.
- Keep doing that until what you need to prove no longer contains any logical connectives.
- Then work forwards from your assumptions (using elimination rules) until you've proved what you need.



Making Mistakes

- To err is human.
- Isabelle will catch your mistakes.
- Unfortunately, Isabelle is bad at describing your mistake.
- Consider the following attempted proof

```
theorem "p ⟶ (p ∧ p)"
proof -
  show "p ⟶ (p ∧ p)"
  proof
    assume 1: "p"
    from 1 show "p ∧ p"
```

- When Isabelle gets to **from** 1 **show** `"p ∧ p"` (adding .. at the end), it gives the following response:

```
Failed to finish proof
At command "..".
```

Making Mistakes, cont'd

- In this case, the mistake was a missing label in the **from** clause. Conjuction introduction requires two premises, not one. Here's the fix:

```
theorem "p ⟶ (p ∧ p)"
proof -
  show "p ⟶ (p ∧ p)"
  proof
    assume 1: "p"
    from 1 1 show "p ∧ p" ..
  qed
qed
```

- When Isablle says "no", double check the inference rule. If that doesn't work, get a classmate to look at it. If that doesn't work, email the instructor with the minimal Isabelle file that exhibits your problem.

Making Mistakes, cont'd

- Here's another proof with a typo:

  theorem "p $\longrightarrow$ p"
  proof
      assume 1: "p"
      from 1 **show** "q" .
  qed

- Isabelle responds with:

```
Local statement will fail to refine any pending goal
Failed attempt to solve goal by exported rule:
  (p) ==> q
At command "show".
```

- The problem here is that the proposition in the **show** "q", does not match what we are trying to prove, which is p.

Stuff to Rememeber

- How to write Isabelle/Isar proofs of tautologies in Propositional Logic.
- The introduction and elimination rules.
- Forwards and backwards reasoning.

Outline of Lecture 4

1. Overview of First-Order Logic
2. Beyond Booleans: natural numbers, integers, etc.
3. Universal truths: “for all”
4. Existential truths: “there exists”

Overview of First-Order Logic

- First-order logic is an extension of propositional logic, adding the ability to reason about well-defined entities and operations.
- Isabelle provides many entities, such as natural numbers, integers, and lists.
- Isabelle also provides the means to define new entities and their operations.
- First-order logic adds two new kinds of propositions, “for all” ($\forall$) and “there exists” ($\exists$), that enable quantification over these entities.
- For example, first-order logic can express $\forall x :: nat.\ x = x$.

Beyond Booleans

- Natural numbers: $0, 1, 2, \ldots$
- Integers: $\ldots, -1, 0, 1, \ldots$
- How does Isabelle know the difference between 0 (the natural number) and 0 (the integer)?
- Sometimes it can tell from context, sometimes it can't. (When it can't, you'll see things like `0::'a`)
- You can help Isabelle by giving a type annotation, such as `0` or `0`.
- We use natural numbers a lot, integers not so much.

Natural Numbers

- There's only two ways to construct a natural number:
  - `0`
  - If `n` is a natural number, then so is `Suc n`.
    (`Suc` is for successor. Think of `Suc n` as `n + 1`.)
- Isabelle provides shorthands for numerals:
  - `1 = Suc 0`
  - `2 = Suc (Suc 0)`
  - `3 = Suc (Suc (Suc 0))`

Arithmetic on Natural Numbers

- Isabelle provides arithmetic operations and many other functions on natural numbers.
- Warning: arithmetic on naturals is sometimes similar and sometimes different than integers. See `/Isabelle/src/HOL/Nat.thy`.
- For example,

$$1 + 1 - 2 = 0$$
$$1 - 2 + 1 = 1$$

Universal Truths

- How do we express that a property is true *for all* natural numbers?
- Let `P` be some proposition that may mention `n`, then the following is a proposition:

$$\forall\texttt{n. P}$$

- Example:
  - $\forall$`i j k. i + (j + k) = i + j + k`
  - $\forall$`i j k. i = j` $\wedge$ `j = k` $\longrightarrow$ `i = k`

Introduction and Elimination Rules

For all-Intro

have L_k: `"∀ n.  P"`
proof
  fix n
  ⋮
  ··· **show** `"P"` ···
qed

For all-Elim If L_i proves $\forall$`n. P`, then write

from L_i **have** L_k: `"[n↦m]P"` ..

where `m` is any entity of the same type as `n`.

The notation `[n↦m]P` (called *substitution*) refers to the proposition that is the same as `P` except that all free occurences of `n` in `P` are replaced by `m`.

Substitution

- $[x \mapsto 1]x = 1$

Substitution

- $[x \mapsto 1]x = 1$
- $[x \mapsto 1]y = y$

Substitution

- $[x \mapsto 1]x = 1$
- $[x \mapsto 1]y = y$
- $[x \mapsto 1](x \wedge y) = (1 \wedge y)$

Substitution

- $[x \mapsto 1]x = 1$
- $[x \mapsto 1]y = y$
- $[x \mapsto 1](x \wedge y) = (1 \wedge y)$
- $[x \mapsto 1](\forall y.\ x) = (\forall y.\ 1)$

Substitution

- $[x \mapsto 1]x = 1$
- $[x \mapsto 1]y = y$
- $[x \mapsto 1](x \wedge y) = (1 \wedge y)$
- $[x \mapsto 1](\forall y.\ x) = (\forall y.\ 1)$
- $[x \mapsto 1](\forall x.\ x) = (\forall x.\ x)$ (The x under $\forall x$ is not free, it is bound by $\forall x$.)

Substitution

- $[x \mapsto 1]x = 1$
- $[x \mapsto 1]y = y$
- $[x \mapsto 1](x \wedge y) = (1 \wedge y)$
- $[x \mapsto 1](\forall y.\ x) = (\forall y.\ 1)$
- $[x \mapsto 1](\forall x.\ x) = (\forall x.\ x)$ (The x under $\forall x$ is not free, it is bound by $\forall x$.)
- $[x \mapsto 1]((\forall x.x) \wedge x) = ((\forall x.\ x) \wedge 1)$

Example Proof using ∀

```
theorem
  assumes 1: "∀ x.  man(x) ⟶ human(x)"
  and 2: "∀ x. human(x) ⟶ hastwolegs(x)"
  shows "∀ x. man(x) ⟶ hastwolegs(x)"
proof
  fix m
  show "man(m) ⟶ hastwolegs(m)"
  proof
    assume 3: "man(m)"
    from 1 have 4: "man(m) ⟶ human(m)" ..
    from 4 3 have 5: "human(m)" ..
    from 2 have 6: "human(m) ⟶ hastwolegs(m)" ..
    from 6 5 show "hastwolegs(m)" ..
  qed
qed
```

Exercise using ∀

Prove the universal modus ponens rule in Isabelle:

$$\texttt{(}\forall\texttt{x. P x} \longrightarrow \texttt{Q x)} \land \texttt{P a} \longrightarrow \texttt{Q a}$$

Example of Proof by Cases

```
theorem fixes n::nat shows "n ≤ n^2"
proof (cases n)
  case 0
  have 1: "(0::nat) ≤ 0^2" by simp
  from 1 show "n ≤ n^2" by (simp only: 0)
next
  case (Suc m)
  have "Suc m ≤ (Suc m) * (Suc m)" by simp
  also have "... = (Suc m)^2"
    by (rule Groebner_Basis.class_semiring.semiring_rules)
  finally have 1: "Suc m ≤ (Suc m)^2" .
  from 1 show "n ≤ n^2" by (simp only: Suc)
qed
```

- The **fixes** is like a $\forall$ for the variable `n`.
- The **by** `simp` performs arithmetic and equational reasoning.
- The `also`/`finally` combination provides a shorthand for equational reasoning. The ... stands for the right-hand side of the previous line.

Existential Truths

- How do we express that a property is true "for some" natural number?
- Or equivalenty, expressing that "there exists" a natural number with the property.
- Let `P` be some proposition that may mention variable `n`, then the following is a proposition:

$$\exists \texttt{n. P}$$

Introduction and Elimination Rules for ∃

Exists-Intro If L_i proves P, then write
from L_i **have** L_k: `"∃ n.P"` ..

Exists-Elim If L_i proves `∃n. P`, then write
from L_i **obtain** `m` **where** L_k: `"[n↦m]P"` ..

Exercise Proof Using ∃

Given the following definitions:

$$
\begin{aligned}
even(n) &\equiv \exists m.\ n = 2m \\
odd(n) &\equiv \exists m.\ n = 2m + 1
\end{aligned}
$$

Prove on paper that if n and m are odd, then $n + m$ is even.

Proof Using ∃

Theorem

If n and m are odd, then $n + m$ is even.

Proof.

Because n is odd, there exists a k where $n = 2k + 1$. Because m is odd, there exists a q where $m = 2q + 1$. So $n + m = 2k + 2q + 2 = 2(k + q + 1)$. Thus $\exists p.\ n + m = 2p$, and by definition, $n + m$ is even. □

Isabelle Definitions

```
definition even :: "nat ⇒ bool" where
  "even n ≡ ∃ m. n = 2 * m"

definition odd :: "nat ⇒ bool" where
  "odd n ≡ ∃ m. n = 2 * m + 1"
```

- **definition** is a way to create simple functions.
- Definitions may not be recursive.
- **by** `simp` does not automatically unfold definitions, need to use **unfolding** (see next slide).

Proof In Isabelle Using Definitions and ∃

```
theorem assumes 1: "odd n" and 2: "odd m"
  shows "even (n + m)"
proof -
  from 1 have 3: "∃ k. n = 2 * k + 1" unfolding odd_def .
  from 3 obtain k where 4: "n = 2 * k + 1" ..
  from 2 have 5: "∃ q. m = 2 * q + 1" unfolding odd_def .
  from 5 obtain q where 6: "m = 2 * q + 1" ..
  from 4 6 have 7: "n + m = 2 * (k + q + 1)" by simp
  from 7 have 8: "∃ p. n + m = 2 * p" ..
  from 8 show "even (n + m)" unfolding even_def .
qed
```

First-Order Logic over Natural Numbers

- How expressive is First-Order Logic over Natural Numbers?

First-Order Logic over Natural Numbers

- How expressive is First-Order Logic over Natural Numbers?
- Can you write down the rules for Sudoku?

First-Order Logic over Natural Numbers

- How expressive is First-Order Logic over Natural Numbers?
- Can you write down the rules for Sudoku?
- What's missing?

Stuff to Rememeber

- First-Order Logic adds the ability to reason about well-defined entities and adds $\forall$ and $\exists$.
- Natural numbers.
- Proof rules for $\forall$ and $\exists$.
- New from Isabelle: **by** `simp`, **also/finally**, **unfolding**, **fix**, **obtain/where**, **definition**.

Outline of Lecture 5

1. Proof by induction
2. Functions, defined by primitive recursion

Induction

- Induction is the primary way we *prove* universal truths about entities of unbounded size (like natural numbers).
- (If the size is bounded, then we can do proof by cases.)
- Induction is also the way we *define* things about entities of unbounded size.

Motivation: Dominos

- Domino Principle: Line up any number of dominos in a row; knock the first one over and they all fall down.
- Let F_k be the statement that the kth domino falls.
- We know that, for any k, if F_k falls down, then so does F_{k+1}.
- We knock down F_0.
- It's clear that for any n, F_n falls down, i.e., $\forall n.\ F_n$.

Mathematical Induction

To show that some property P is universally true of natural numbers

$$\forall \texttt{n. P n}$$

you need to prove

- `P 0`
- $\forall$`n. P n` $\longrightarrow$ `P (n + 1)`

Example Proof by Mathematical Induction

Theorem

$\forall n.\ 0 + 1 + \cdots + n = \frac{n(n+1)}{2}$.

Proof.

The proof is by mathematical induction on n.

- **Base Step:** We need to show that $0 = \frac{0(0+1)}{2}$, but that's obviously true.
- **Inductive Step:** The inductive hypothesis (IH) is $0 + 1 + \cdots + n = \frac{n(n+1)}{2}$.

$$\begin{aligned} 0 + 1 + \cdots + n + (n+1) &= (n+1) + \frac{n(n+1)}{2} \qquad \text{(by the IH)} \\ &= \frac{2(n+1) + n(n+1)}{2} = \frac{(n+1)(n+2)}{2} \\ &= \frac{(n+1)((n+1)+1)}{2}. \end{aligned}$$

$\square$

Primitive Recursive Functions in Isabelle

- First, we need to express $0 + 1 + \cdots + n$ in Isabelle. We can define a function that sums up the natural numbers.
- Isabelle provides a mechanism, called `primrec`, for defining simple recursive functions.
- There is one clause in the `primrec` for each way of creating the input value. (Recall the two ways to create a natural.)
- You may recursively call the function on a sub-part of the input, in this case the `n` within `Suc n`. In Isabelle, function call doesn't require parenthesis, just list the argumetns after the function.
- The $\Rightarrow$ symbol is for function types. The input type (the *domain*) is to the left of the arrow and the output type (the *codomain*) is to the right.

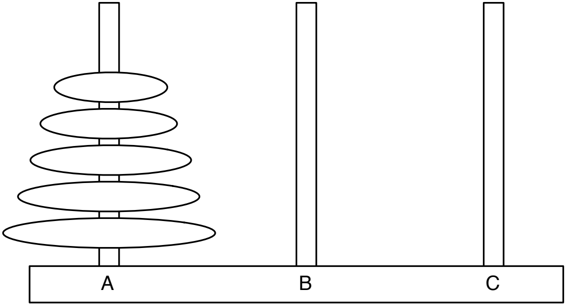
```
primrec sumto :: "nat ⇒ nat" where
  "sumto 0 = 0" |
  "sumto (Suc n) = Suc n + sumto n"
```

Mathematical Induction in Isabelle

```
theorem "sumto n = (n*(n + 1)) div 2"
proof (induct n)
  show "sumto 0 = 0*(0 + 1) div 2" by simp
next
  fix n assume IH: "sumto n = n*(n + 1) div 2"
  have "sumto(Suc n) = Suc n + sumto n" by simp
  also from IH have "... = Suc n + (n*(n+1) div 2)" by simp
  also have "... = (Suc n * (Suc n + 1)) div 2" by simp
  finally show "sumto(Suc n) = (Suc n * (Suc n + 1)) div 2" .
qed
```

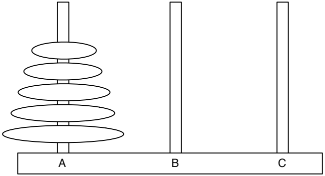
Tower of Hanoi

- Can you move all of the discs from peg A to peg C?
- Complication: you are not allowed to put larger discs on top of smaller discs.



- How long does your algorithm take?

Tower of Hanoi, cont'd



- Algorithm: To move n discs from peg A to peg C:
  1. Move $n-1$ discs from A to B.
  2. Move disc $\#n$ from A to C.
  3. Move $n-1$ discs from B to C so they sit on disc $\#n$.
- Let's characterize the number of moves needed for a tower of n discs.

$$
\begin{aligned}
T(0) &= 0 \\
T(n) &= 2T(n-1) + 1
\end{aligned}
$$

Tower of Hanoi, cont'd

$$T(0) = 0$$
$$T(n) = 2T(n-1) + 1$$

- The above is an example of a recurrence relation.
- It's a valid definition, but a bit difficult to understand and a bit expensive to evaluate (suppose n is large!). Can you think of a non-recursive expression for $T(n)$?

Tower of Hanoi, cont'd

$$T(0) = 0$$
$$T(n) = 2T(n-1) + 1$$

- The above is an example of a recurrence relation.
- It's a valid definition, but a bit difficult to understand and a bit expensive to evaluate (suppose n is large!). Can you think of a non-recursive expression for $T(n)$?
- Here's a closed form solution:

$$T(n) = 2^n - 1$$

- On paper, prove that the closed form solution is correct.

Exercise, Tower of Hanoi in Isabelle

- Create a `primrec` for $T(n)$.

$$T(0) = 0$$
$$T(n) = 2T(n-1) + 1$$

- Prove that $T(n) = 2^n - 1$ in Isabelle.
- In addition to **by** `simp`, you will need to use **by** `arith`, which performs slightly more advanced arithmetical reasoning.
- Hint: if Isabelle rejects one of the steps in your proof, try creating a new step that is a smaller "distance" from the previous step.

Solution for Tower of Hanoi

```
primrec moves :: "nat ⇒ nat" where
  "moves 0 = 0" |
  "moves (Suc n) = 2 * (moves n) + 1"

theorem "moves n = 2^n - 1"
proof (induct n)
  show "moves 0 = 2^0 - 1" by simp
next
  fix n assume IH: "moves n = 2 ^ n - 1"
  have 1: "(2::nat) ≤ 2 ^ (Suc n)" by simp
  have "moves (Suc n) = 2 * (moves n) + 1" by simp
  also from IH have "... = 2 * ((2 ^ n) - 1) + 1" by simp
  also have "... = 2 ^ (Suc n) - 2 + 1" by simp
  also from 1 have "... = 2 ^ (Suc n) - 1" by arith
  finally show "moves (Suc n) = 2 ^ (Suc n) - 1" .
qed
```

Stuff to Rememeber

- Mathematical induction.
- New from Isabelle: **by** `arith`, **primrec**.

Outline of Lecture 6

1. More proof by induction and recursive functions
2. Repeated function composition example.

Some Suggestions

1. Use a peice of scratch paper to sketch out the main ideas of the proof.
2. Dedicate one part of the paper to things that you know (assumptions, stuff you've proven),
3. Dedicate another part of the paper to things that you'd like to know.
4. After your sketch is complete, write a nicely organized and clean version of the proof.
5. Now let's look at more examples of induction.

Repeated Function Composition

```
primrec rep :: "('a ⇒ 'a) ⇒ nat ⇒ 'a ⇒ 'a"   where
    "rep f 0 x = x"
  | "rep f (Suc n) x = rep f n (f x)"
```

First Attempt

```
theorem rep_add: "rep f (m + n) x = rep f n (rep f m x)"
proof (induct m)
  show "rep f (0 + n) x = rep f n (rep f 0 x)" by simp
next
  fix k assume IH: "rep f (k + n) x = rep f n (rep f k x)"
  have "rep f ((Suc k) + n) x = rep f (Suc (k + n)) x" by simp
  also have "... = rep f (k + n) (f x)" by simp
  — Stuck, we can’t apply the IH. We need to add a “forall” for x.
  show "rep f ((Suc k) + n) x = rep f n (rep f (Suc k) x)"
    oops
```

Generalized Theorem

```
theorem rep_add: "∀ x. rep f (m + n) x = rep f n (rep f m x)"
proof (induct m)
  show "∀ x. rep f (0 + n) x = rep f n (rep f 0 x)" by simp
next
  fix k assume IH: "∀ x. rep f (k + n) x = rep f n (rep f k x)"
  show "∀ x. rep f ((Suc k) + n) x = rep f n (rep f (Suc k) x)"
  proof
    fix x
    have "rep f ((Suc k) + n) x = rep f (Suc (k + n)) x" by simp
    also have "... = rep f (k + n) (f x)" by simp
    also from IH have "... = rep f n (rep f k (f x))" by simp
    finally show "rep f ((Suc k)+n) x = rep f n (rep f (Suc k) x)"
      by simp
  qed
qed
```

Repeated Function, Difference

theorem `rep_diff:`
  assumes `nm: "n` $\leq$ `m"` **shows** `"rep f (m - n) (rep f n x) = rep f m x"`
  oops

Repeated Function, Difference

This proof is easy, a direct consequence of the `rep_add` theorem.

```
theorem rep_diff:
  assumes nm: "n ≤ m" shows "rep f (m - n) (rep f n x) = rep f m x"
proof -
  from nm have 1: "n + (m - n) = m" by simp
  from 1 show "rep f (m - n) (rep f n x) = rep f m x"
    using rep_add[of f n "m - n"] by simp
qed
```

Outline of Lecture 7

1. In class exercise concerning repeated function composition

Repeated Function, Cycle

- Which natural number should we do induction on, `m` or `n`?
- Sometimes you just have to try both and see which one works.
- Sometimes you can foresee which one is better.

```
lemma rep_cycle: "rep f n x = x ⟶ rep f (m*n) x = x"
  oops
```

Repeated Function, Cycle

Let's try to do induction on `n`.

```
lemma rep_cycle: "rep f n x = x ⟶ rep f (m*n) x = x"
proof (induct n)
  show "rep f 0 x = x ⟶ rep f (m*0) x = x" by simp
next
  fix k assume IH: "rep f k x = x ⟶ rep f (m*k) x = x"
  show "rep f (Suc k) x = x ⟶ rep f (m*(Suc k)) x = x"
  proof
    assume 1: "rep f (Suc k) x = x"
    — Problem: we can't use the IH because we can't prove that rep f k x = x
    oops
```

Repeated Function, Cycle

Now let's try induction on `m`.

```
lemma rep_cycle: "rep f n x = x ⟶ rep f (m*n) x = x"
proof (induct m)
  show "rep f n x = x ⟶ rep f (0*n) x = x"
  proof
    assume "rep f n x = x" — We dont' use this assumption
    show "rep f (0*n) x = x" by simp
  qed
next
  fix k assume IH: "rep f n x = x ⟶ rep f (k*n) x = x"
  show "rep f n x = x ⟶ rep f ((Suc k)*n) x = x"
  proof
    assume 1: "rep f n x = x"
    have "rep f ((k+1)*n) x = rep f (n + k*n) x" by simp
    also have "... = rep f (k*n) (rep f n x)" using rep_add by force
    also from 1 have "... = rep f (k*n) x" by simp
    also from 1 IH have "... = x" by simp
    finally show "rep f ((Suc k)*n) x = x" by simp
  qed
qed
```

Outline of Lecture 8

1. Lists (to represent finite sequences).
2. More induction

Lists

- Isabelle's lists are descended from the Lisp language, they are built up using two operations:
  1. The empty list: `[]`
  2. If `x` is an object, and `ls` is a list of objects, then `x # ls` is a new list with `x` at the front and the rest being the same as `ls`.
- Also, lists can be created from a comma-separated list enclosed in brackets: `[1, 2, 3, 4]`.
- All the objects in a list must have the same type.

Functions on Lists

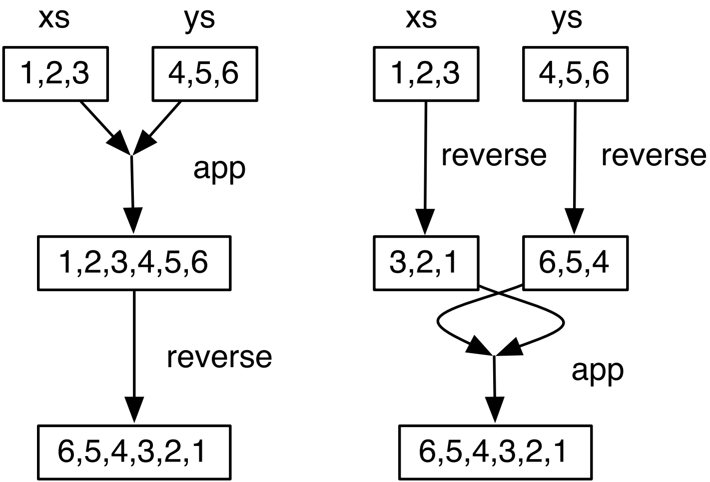
- You can write primitive recursive functions over lists:

```
primrec app :: "'a list ⇒ 'a list ⇒ 'a list" where
  "app [] ys = ys" |
  "app (x#xs) ys = x # (app xs ys)"

lemma "app [1,2] [3,4] = [1,2,3,4]" by simp

primrec reverse :: "'a list ⇒ 'a list" where
  "reverse [] = []" |
  "reverse (x#xs) = app (reverse xs) [x]"

lemma "reverse [1,2,3,4] = [4,3,2,1]" by simp
```

Induction on Lists and the Theorem Proving Process

```
theorem rev_rev_id: "reverse (reverse xs) = xs"
proof (induct xs)
  show "reverse (reverse []) = []" by simp
next
  fix a xs assume IH: "reverse (reverse xs) = xs"
  — We can expand the LHS of the goal as follows
  have "reverse (reverse (a # xs))
      = reverse (app (reverse xs) [a])" by simp
  — But then we're stuck. How can we use the IH?
  — Can we push the outer reverse under the app?
  show "reverse (reverse (a # xs)) = a # xs"
    oops
```

Reverse-Append Lemma



reverse(app(xs,ys)) = app(reverse(ys), reverse(xs))

Reverse-Append Lemma

```
lemma rev_app:
  "reverse (app xs ys) = app (reverse ys) (reverse xs)"
proof (induct xs)
  have 1: "reverse (app [] ys) = reverse ys" by simp
  have 2: "app (reverse ys) (reverse []) = app (reverse ys) []"
by simp
  — but no we're stuck
  show "reverse (app [] ys) = app (reverse ys) (reverse [])"
    oops
```

Exercise: what additional lemma do we need? Prove the additional lemma.

The Append-Nil Lemma

```
lemma app_nil: "(app xs []) = xs"
proof (induct xs)
  show "app [] [] = []" by simp
next
  fix a xs assume IH: "app xs [] = xs"
  have "app (a # xs) [] = a # (app xs [])" by simp
  also from IH have "... = a # xs" by simp
  finally show "app (a # xs) [] = a # xs" .
qed
```

Back to Reverse-Append Lemma

```
lemma rev_app:
  "reverse (app xs ys) = app (reverse ys) (reverse xs)"
proof (induct xs)
  show "reverse (app [] ys) = app (reverse ys) (reverse [])"
    using app_nil[of "reverse ys"] by simp
next
  fix a xs assume IH: "reverse (app xs ys)
                       = app (reverse ys) (reverse xs)"
  have "reverse (app (a # xs) ys)
                 = reverse (a # (app xs ys))" by simp
  also have "... = app (reverse (app xs ys) ) [a]" by simp
  also have "... = app (app (reverse ys) (reverse xs)) [a]"
    using IH by simp
  — We're stuck again! What lemma do we need this time?
  show "reverse (app (a # xs) ys)
      = app (reverse ys) (reverse (a # xs))"
    oops
```

Associativity of Append

lemma `app_assoc: "app (app xs ys) zs = app xs (app ys zs)"`
  oops

Associativity of Append

```
lemma app_assoc: "app (app xs ys) zs = app xs (app ys zs)"
proof (induct xs)
  show "app (app [] ys) zs = app [] (app ys zs)" by simp
next
  fix a xs assume IH: "app (app xs ys) zs = app xs (app ys zs)"
  from IH
  show "app (app (a # xs) ys) zs = app (a # xs) (app ys zs)"
    by simp
qed
```

Back to the Reverse-Append Lemma, Again

```
lemma rev_app:
  "reverse (app xs ys) = app (reverse ys) (reverse xs)"
proof (induct xs)
  show "reverse (app [] ys) = app (reverse ys) (reverse [])"
    using app_nil[of "reverse ys"] by simp
next
  fix a xs assume IH: "reverse (app xs ys)
                      = app (reverse ys) (reverse xs)"
  have "reverse (app (a # xs) ys)
                 = reverse (a # (app xs ys))" by simp
  also have "... = app (reverse (app xs ys) ) [a]" by simp
  also have "... = app (app (reverse ys) (reverse xs)) [a]"
    using IH by simp
  also have "... = app (reverse ys) (app (reverse xs) [a])"
    using app_assoc[of "reverse ys" "reverse xs" "[a]"] by simp
  also have "... = app (reverse ys) (reverse (a # xs))" by simp
  finally show "reverse (app (a # xs) ys)
              = app (reverse ys) (reverse (a # xs))" .
qed
```

Finally, Back to the Theorem!

```
theorem rev_rev_id: "reverse (reverse xs) = xs"
proof (induct xs)
  show "reverse (reverse []) = []" by simp
next
  fix a xs assume IH: "reverse (reverse xs) = xs"
  — We can expand the LHS of the goal as follows
  have "reverse (reverse (a # xs))
      = reverse (app (reverse xs) [a])" by simp
  also have "... = app (reverse [a]) (reverse (reverse xs))"
    using rev_app[of "reverse xs" "[a]"] by simp
  also from IH have "... = app (reverse [a]) xs" by simp
  also have "... = a # xs" by simp
  finally show "reverse (reverse (a # xs)) = a # xs" .
qed
```

More on Lists and the Theorem Proving Process

- When proving something about a recursive function, induct on the argument that is decomposed by the recursive function (e.g., the first argument of append).
- The pattern of getting stuck and then proving lemmas is normal.
- Isabelle provides many functions and theorems regarding lists. See `Isabelle/src/HOL/List.thy` for more details.

Stuff to Rememeber

- Use lists to represent finite sequences.
- Isabelle provides many functions and theorems regarding lists. See `Isabelle/src/HOL/List.thy` for more details.
- Proofs often require several lemmas.
- Generalize your lemmas to make the induction go through.

Outline of Lecture 9

1. Converting loops into recursive functions and accumulator passing style.
2. More generalizing theorems for induction

Iterative Reverse Algorithm

- The reverse function is inneficient because it uses the append function over and over again.
- The following iterative algorithm reverses a list in linear time (textbook page 317).

```
procedure iterative_reverse(list)
    xs = list
    ys = []
    while xs != []
        ys = hd(xs) # ys
        xs = tl(xs)
    return ys
```

Accumulator Passing Style

- The following `itrev` function is a recursive version of the iterative algorithm.
- The trick is to add an extra parameter for each variable that gets updated in the for loop of the iterative algorithm.

```
primrec itrev :: "'a list ⇒ 'a list ⇒ 'a list" where
  "itrev [] ys = ys" |
  "itrev (x#xs) ys = itrev xs (x#ys)"

lemma "itrev [1,2,3] [] = [3,2,1]"
proof -
  have "itrev [1,2,3] [] = itrev [2,3] [1]" by simp
  also have "... = itrev [3] [2,1]" by simp
  also have "... = itrev [] [3,2,1]" by simp
  also have "... = [3,2,1]" by simp
  finally show ?thesis .
qed
```

Correctness of `itrev`

Let's try to prove that `itrev` reverses a list.

lemma `"itrev xs [] = reverse xs"`
  oops

Generalizing in Proofs by Induction

```
lemma "itrev xs [] = reverse xs"
proof (induct xs)
  show "itrev [] [] = reverse []" by simp
next
  fix x xs assume IH: "itrev xs [] = reverse xs"
  have "itrev (x#xs) [] = itrev xs [x]" by simp
  oops
```

- The induction hypothesis does not apply to `itrev xs [x]`.
- We need to generalize the lemma, make it stronger, to give ourselves more to assume in the induction hypothesis.

Generalizing in Proofs by Induction

```
lemma "∀ ys. itrev xs ys = app (reverse xs) ys"
proof (induct xs)
  show "∀ ys. itrev [] ys = app (reverse []) ys" by simp
next
  fix x xs assume IH: "∀ ys. itrev xs ys = app (reverse xs) ys"
  show "∀ ys. itrev (x#xs) ys = app (reverse (x # xs)) ys"
  proof
    fix ys
    have "itrev (x#xs) ys = itrev xs (x#ys)" by simp
    also from IH have "... = app (reverse xs) (x#ys)" by simp
    also have "... = app (reverse xs) (app [x] ys)" by simp
    also have "... = app (app (reverse xs) [x]) ys"
      by (simp only: app_assoc)
    also have "... = app (reverse (x # xs)) ys" by simp
    finally show "itrev (x#xs) ys = app (reverse (x # xs)) ys" .
  qed
qed
```

Outline of Lecture 10

1. Mini-project regarding the Fibonacci function:
   1.1 practice converting loops into recursive functions.
   1.2 proving correctness of algorithms.
2. In-class discussion of the solution.

Definition of Fibonacci

```
fun fib :: "nat ⇒ nat" where
  "fib 0 = 0" |
  "fib (Suc 0) = 1" |
  "fib (Suc(Suc x)) = fib x + fib (Suc x)"
```

Iterative Fibonacci Algorithm

- The fib function is inneficient because it redundantly computes the same fibonacci number over and over.
- The following iterative algorithm computes Fibonacci numbers in linear time (textbook page 317).

```
procedure iterative_fibonacci(n)
  if n = 0 then
    y := 0
  else
    x := 0
    y : = 1
    for i := 1 to n - 1
      z := x + y
      x := y
      y := z
  return y
```

Project

1. Implement a recursive version of the iterative fibonacci algorithm. Use accumulator passing style.
2. Prove that your recursive function produces the same output as fib.

Accumulator Passing Fibonacci Function

```
primrec itfib :: "nat ⇒ nat ⇒ nat ⇒ nat" where
  "itfib f f' 0 = f" |
  "itfib f f' (Suc k) = itfib f' (f + f') k"
```

Proof of Correctness

```
theorem "∀ n. itfib (fib n) (fib (n + 1)) k = fib (n + k)"
proof (induct k)
  show "∀ n. itfib (fib n) (fib (n + 1)) 0 = fib (n + 0)" by simp
next
  fix k assume IH: "∀ n. itfib (fib n) (fib (n + 1)) k = fib (n + k)"
  show "∀ n. itfib (fib n) (fib (n + 1)) (Suc k) = fib (n + Suc k)"
  proof
    fix n
    have "itfib (fib n) (fib (n + 1)) (Suc k)
      = itfib (fib (n + 1)) (fib n + fib (n + 1)) k"
      by simp — by the definition of itfib
    also have "... = itfib (fib (n + 1)) (fib (n + 2))  k"
      by simp — by the definition of fib
    also have "... = fib (n + k + 1)"
    proof -
      from IH have 1: "itfib (fib (n + 1)) (fib ((n + 1) + 1))  k
        = fib ((n + 1) + k)" ..
      from 1 show ?thesis by simp
    qed
    finally show "itfib (fib n) (fib (n + 1)) (Suc k) = fib (n + Suc k)"
      by simp
  qed
qed
```