

Automated Reasoning

Lecture 1: Introduction

Jacques Fleuriot
`jdf@inf.ed.ac.uk`

What is it to Reason?

- ▶ Reasoning is a process of deriving new statements (conclusions) from other statements (premises) by argument.
- ▶ For reasoning to be correct, this process should generally **preserve truth**. That is, the arguments should be **valid**.
- ▶ How can we be sure our arguments are valid?
- ▶ Reasoning takes place in many different ways in everyday life:
 - ▶ **Word of Authority:** derive conclusions from a trusted source.
 - ▶ **Experimental science:** formulate hypotheses and try to confirm or falsify them by experiment.
 - ▶ **Sampling:** analyse evidence statistically to identify patterns.
 - ▶ **Mathematics:** we derive conclusions based on deductive *proof*.
- ▶ Are any of the above methods **valid**?

What is a Proof? (I)

- ▶ For centuries, mathematical proof has been the hallmark of logical validity.
- ▶ But there is still a **social aspect** as peers have to be convinced by argument.

A proof is a repeatable experiment in persuasion

— Jim Horning¹

- ▶ This process is open to flaws: e.g., Kempe's acclaimed 1879 “proof” of the Four Colour Theorem, etc.



¹https://en.wikipedia.org/wiki/Jim_Horning

What is a Formal Proof?

- We can be sure there are no hidden premises, or unjustified steps, by reasoning according to **logical form** alone.

Example

Suppose all humans are mortal. Suppose Socrates is human.
Therefore, Socrates is mortal.

- The validity of this proof is independent of the meaning of “human”, “mortal” and “Socrates”.
- Even a nonsense substitution gives a valid sentence:

Example

Suppose all borogroves are mimsy. Suppose a mome rath is a borogrove. Therefore, a mome rath is mimsy.²

Example

Suppose all P s are Q . Suppose x is a P . Therefore, x is a Q .

²https://en.wikipedia.org/wiki/Mimsy_Were_the_Borogoves

Symbolic Logic

- ▶ The modern notion of **symbolic proof** was developed in the late-19th and 20th century by logicians and mathematicians such as Bertrand Russell, Gottlob Frege, David Hilbert, Kurt Gödel, Alfred Tarski, Julia Robinson, ...
- ▶ The benefit of formal logic is that it is based on a **pure syntax: a precisely defined symbolic language with procedures for transforming symbolic statements into other statements, based solely on their form.**
- ▶ **No intuition or interpretation is needed**, merely applications of agreed upon rules to a set of agreed upon formulae.

Symbolic Logic (II)

But!

- ▶ Formal proofs are bloated!

I find nothing in [formal logic] but shackles. It does not help us at all in the direction of conciseness, far from it; and if it requires 27 equations to establish that 1 is a number, how many will it require to demonstrate a real theorem?

— Poincaré

- ▶ Can automation help?

Automated Reasoning

- ▶ Automated Reasoning (AR) refers to reasoning in a computer using logic.
- ▶ AR has been an active area of research since the 1950s.
- ▶ Traditionally viewed as part of Artificial Intelligence (AI ≠ Machine Learning!).
- ▶ It uses deductive reasoning to tackle problems such as
 - ▶ constructing formal mathematical proofs;
 - ▶ verifying that programs meet their specifications;
 - ▶ modelling human reasoning.

Mathematical Reasoning

Mechanical mathematical theorem proving is an exciting field. Why?

- ▶ Intelligent, often non-trivial activity.
- ▶ Circumscribed domain with bounds that help control reasoning.
- ▶ Mathematics is based around logical proof and – in principle – reducible to formal logic.
- ▶ Numerous applications
 - ▶ the need for formal mathematical reasoning is increasing: need for well-developed theories;
 - ▶ e.g. **hardware and software verification**;
 - ▶ e.g. research mathematics, where formal proofs are starting to be accepted.

Understanding mathematical reasoning

- ▶ Two main aspects have been of interest
 - ▶ **Logical:** how should we reason; what are the valid modes of reasoning?
 - ▶ **Psychological:** how do we reason?
- ▶ Both aspects contribute to our understanding
- ▶ (Mathematical) Logic:
 - ▶ shows how to represent mathematical knowledge and inference;
 - ▶ does not tell us how to **guide** the reasoning process.
- ▶ Psychological studies:
 - ▶ do not provide a detailed and precise recipe for how to reason, but can provide advice and hints or **heuristics**;
 - ▶ heuristics are especially valuable in automatic theorem proving – but finding good ones is a hard task.

Mechanical Theorem Proving

- ▶ Many systems: Isabelle, Coq, HOL Light, PVS, Vampire, E, ...
 - ▶ provide a mechanism to formalise proof;
 - ▶ user-defined concepts in an **object-logic**;
 - ▶ user expresses formal conjectures about concepts.
- ▶ Can these systems find proofs **automatically**?
 - ▶ In some cases, yes!
 - ▶ But sometimes it is too difficult.
- ▶ Complicated verification tasks are usually done in an **interactive** setting.

Interactive Proof

- ▶ User guides the inference process to prove a conjecture (hopefully!)
- ▶ Systems provide:
 - ▶ tedious bookkeeping;
 - ▶ standard libraries (e.g., arithmetic, lists, real analysis);
 - ▶ guarantee of correct reasoning;
 - ▶ varying degrees of automation:
 - ▶ powerful simplification procedures;
 - ▶ may have decision procedures for decidable theories such as linear arithmetic, propositional logic, etc.;
 - ▶ call fully-automatic first-order theorem provers on (sub-)goals and incorporating their output e.g. Isabelle's sledgehammer.

What is it like?

- ▶ Interactive proof can be challenging, but also rewarding.
- ▶ It combines aspects of **programming** and **mathematics**.
- ▶ Large-scale interactive theorem proving is relatively new and unexplored:
 - ▶ Many potential application areas are under-explored
 - ▶ Not at all clear what The Right Thing To Do is in many situations
 - ▶ New ideas are needed all the time
 - ▶ This is what makes it **exciting!**
- ▶ What we do know: **Representation** matters!

$\sqrt{2}$ is irrational in Isabelle

```
theorem sqrt_prime_irrational:
  assumes "prime (p::nat)"
  shows "sqrt p ∉ ℚ"
proof
  from ‹prime p› have p: "1 < p" by (simp add: prime_nat_def)
  assume "sqrt p ∈ ℚ"
  then obtain m n :: nat where
    n: "n ≠ 0" and sqrt_rat: "|sqrt p| = m / n"
    and gcd: "gcd m n = 1" by (rule Rats_abs_nat_div_natE)
  from n and sqrt_rat have "m = |sqrt p| * n" by simp
  then have "m2 = (sqrt p)2 * n2" by (auto simp add: power2_eq_square)
  also have "(sqrt p)2 = p" by simp
  also have "... * n2 = p * n2" by simp
  finally have eq: "m2 = p * n2" ..
  then have "p dvd m2" ..
  with ‹prime p› have dvd_m: "p dvd m" by (rule prime_dvd_power_nat)
  then obtain k where "m = p * k" ..
  with eq have "p * n2 = p2 * k2" by (auto simp add: power2_eq_square ac_simps)
  with p have "n2 = p * k2" by (simp add: power2_eq_square)
  then have "p dvd n2" ..
  with ‹prime p› have "p dvd n" by (rule prime_dvd_power_nat)
  with dvd_m have "p dvd gcd m n" by (rule gcd_greatest_nat)
  with gcd have "p dvd 1" by simp
  then have "p ≤ 1" by (simp add: dvd_imp_le)
  with p show False by simp
qed

corollary sqrt_2_not_rat: "sqrt 2 ∉ ℚ"
  using sqrt_prime_irrational[of 2] by simp
```

Limitations (I)

Do you think formalised mathematics is:

- 1. Complete:** can every statement be proved or disproved?
- 2. Consistent:** no statement can be both true and false?
- 3. Decidable:** there exists a terminating procedure to determine the truth or falsity of any statement?

Limitations (II)

- ▶ Gödel's Incompleteness Theorems showed that, if a formal system can prove certain facts of basic arithmetic, then there are other statements that cannot be proven or refuted in that system.
- ▶ In fact, if such a system is consistent, it cannot prove that it is so.
- ▶ Moreover, Church and Turing showed that first-order logic is undecidable.
- ▶ Do not be disheartened!
- ▶ We can still prove many interesting results using logic.

What is a proof? (II)

- ▶ Computerised proofs are causing controversy in the mathematical community
 - ▶ proof steps may be in the hundreds of thousands;
 - ▶ they are impractical for mathematicians to check by hand;
 - ▶ it can be hard to guarantee proofs are not flawed;
 - ▶ e.g., Hales's proof of the Kepler Conjecture.
- ▶ The acceptance of a computerised proof can rely on
 - ▶ formal specifications of concepts and conjectures;
 - ▶ **soundness** of the prover used;
 - ▶ size of the community using the prover;
 - ▶ **surveyability** of the proof;
 - ▶ (for specialists) the kind of logic used.

Isabelle

In this course we will be using the popular interactive theorem prover **Isabelle/HOL**:

- ▶ It is based on the simply typed λ -calculus with rank-1 (ML-style) polymorphism.
- ▶ It has an extensive **theory library**.
- ▶ It supports two styles of proof: procedural ('apply'-style) and declarative (structured).
- ▶ It has a powerful simplifier, classical reasoner, decision procedures for decidable fragments of theories.
- ▶ It can call automatic first-order theorem provers.
- ▶ Widely accepted as a **sound** and **rigorous** system.

Soundness in Isabelle

- ▶ Isabelle follows the **LCF approach** to ensure soundness.
- ▶ We declare our conjecture as a goal, and then we can:
 - ▶ use a known theorem or axiom to prove the goal;
 - ▶ use a **tactic** to prove the goal;
 - ▶ use a tactic to transform the goal into new subgoals.
- ▶ Tactics construct the formal proof in the background.
- ▶ Axioms are generally discouraged; definitions are preferred.
- ▶ New concepts should be **conservative extensions** of old ones.

Course Contents (in brief)

- ▶ **Logics:** first-order, aspects of higher-order logic.
- ▶ **Reasoning:** unification, rewriting, natural deduction.
- ▶ **Interactive theorem proving:** introduction to theorem proving with Isabelle/HOL.
 - ▶ Representation: definitions, locales etc.
 - ▶ Proofs: procedural and structured (Isar) proofs.
- ▶ **Formalised mathematics.**

Module Outline

- ▶ 2 lectures per week 14:10–15:00:
 - ▶ Tuesday: 1.02, 21 Buccleuch Place, Central Campus
 - ▶ Thursday: G.02 - Classroom 2, High School Yards Teaching Centre, Central Campus
- ▶ 7 tutorials (starting Week 3)
- ▶ Lab sessions (drop-in):
 - ▶ Mondays 09:00–11:00 (starting Week 3, to be confirmed)
 - ▶ 4.12, Appleton Tower
- ▶ 1 assignment and 1 exam:
 - ▶ Examination: 60%
 - ▶ Coursework: 40% (so this is a non-trivial part of the course)
- ▶ Lecturer:
 - ▶ Jacques Fleuriot
 - ▶ Office: IF 2.15
- ▶ TA:
 - ▶ Imogen Morris
 - ▶ Email: s1402592@sms.ed.ac.uk

Useful Course Material

- ▶ AR web pages:
<http://www.inf.ed.ac.uk/teaching/courses/ar>.
- ▶ Lecture slides are on the course website.
- ▶ Recommended course textbooks:
 - ▶ T. Nipkow and G. Klein. *Concrete Semantics with Isabelle/HOL*, Springer, 2014.
 - ▶ M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge University Press, 2nd Ed. 2004.
 - ▶ J. Harrison. *Handbook of Practical Logic and Automated Reasoning*, Cambridge University Press, 2009.
 - ▶ A. Bundy. *The Computational Modelling of Mathematical Reasoning*, Academic Press, 1983 available on-line at
<http://www.inf.ed.ac.uk/teaching/courses/ar/book>.
- ▶ Other material – recent research papers, technical reports, etc. will be added to the AR webpage.
- ▶ Class discussion forum (open for registration):
<http://piazza.com/ed.ac.uk/fall2019/infr09042>.

Automated Reasoning

Lecture 2: Propositional Logic and Natural Deduction

Jacques Fleuriot
`jdf@inf.ed.ac.uk`

Logic Puzzles

1. Tomorrow will be sunny or rainy.

Tomorrow will not be sunny.

What will the weather be tomorrow?

2. I like classical or pop music.

If I like classical music, then I am sophisticated.

I don't like pop music.

Am I sophisticated?

3. Fred bought milk or Fred bought lemonade.

Fred bought milk or Fred bought water.

Fred did not buy both water and lemonade.

What did Fred buy?

Syntax of Propositional Logic

Propositional Logic represents the problems we have just seen by using symbols to represent (atomic) propositions.

These can be combined using the following **connectives**:

Name	symbol	usage
not	\neg	$\neg P$
and	\wedge	$P \wedge Q$
or	\vee	$P \vee Q$
implies	\rightarrow	$P \rightarrow Q$
if and only if	\leftrightarrow	$P \leftrightarrow Q$

↑
precedence

Treat all binary connectives as right associative (following Isabelle)

Example

1. $(\text{SunnyTomorrow} \vee \text{RainyTomorrow}) \wedge (\neg \text{SunnyTomorrow})$
2. $(\text{Class} \vee \text{Pop}) \wedge (\text{Class} \rightarrow \text{Soph}) \wedge \neg \text{Pop}$
3. $(M \vee L) \wedge (M \vee W) \wedge \neg(L \wedge W)$

Syntax and Ambiguity

The meanings of some statements can (appear to) be ambiguous:

$$\text{Class} \vee \text{Pop} \wedge \text{Class} \rightarrow \text{Soph} \rightarrow \neg \text{Pop}$$

We can use brackets (parentheses) to disambiguate a statement:

$$(\text{Class} \vee \text{Pop}) \wedge (\text{Class} \rightarrow \text{Soph} \rightarrow \neg \text{Pop})$$

Note that, based on our choice of precedence (on the previous slide),

$$A \vee B \wedge C \quad \text{denotes} \quad A \vee (B \wedge C)$$

Also note that implication is right associative, so:

$$P \rightarrow Q \rightarrow R \quad \text{denotes} \quad P \rightarrow (Q \rightarrow R)$$

Formal Syntax

A syntactically correct formula is called a **well-formed formula (wff)**

Given a (possible infinite) alphabet of propositional symbols \mathcal{L} , the set of wffs is the smallest set such that

- ▶ any symbol $A \in \mathcal{L}$ is a wff;
- ▶ if P and Q are wffs, so are $\neg P$, $P \vee Q$, $P \wedge Q$, $P \rightarrow Q$, and $P \leftrightarrow Q$;
- ▶ if P is a wff, then (P) is a wff.

When we are interested in *abstract* syntax (tree-structure of formulas) rather than *concrete* syntax, we forget the last clause.

Semantics

Each wff is assigned a meaning or **semantics**, T or F, depending on whether its constituent wffs are assigned T or F.

Truth tables are one way to assign truth values to wffs.

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

P	$\neg P$
T	F
F	T

P	Q	$P \vee Q$
T	T	T
T	F	T
F	T	T
F	F	F

P	Q	$P \rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

Semantics of the weather problem

- ① Tomorrow will be sunny or rainy
- ② Tomorrow will not be sunny

What will the weather be tomorrow?

SunnyTomorrow	RainyTomorrow	$\textcircled{1}$ $S \vee R$	$\textcircled{2}$ $\neg S$	$\textcircled{1} \wedge \textcircled{2}$ $(S \vee R) \wedge \neg S$
T	T			
T	F			
F	T			
F	F			

Semantics of the weather problem

- ① Tomorrow will be sunny or rainy
- ② Tomorrow will not be sunny

What will the weather be tomorrow?

SunnyTomorrow	RainyTomorrow	$\textcircled{1}$ $S \vee R$	$\textcircled{2}$ $\neg S$	$\textcircled{1} \wedge \textcircled{2}$ $(S \vee R) \wedge \neg S$
T	T	T		
T	F			
F	T			
F	F			

Semantics of the weather problem

- ① Tomorrow will be sunny or rainy
- ② Tomorrow will not be sunny

What will the weather be tomorrow?

SunnyTomorrow	RainyTomorrow	$\textcircled{1}$ $S \vee R$	$\textcircled{2}$ $\neg S$	$\textcircled{1} \wedge \textcircled{2}$ $(S \vee R) \wedge \neg S$
T	T	T		
T	F	T		
F	T			
F	F			

Semantics of the weather problem

- ① Tomorrow will be sunny or rainy
- ② Tomorrow will not be sunny

What will the weather be tomorrow?

SunnyTomorrow	RainyTomorrow	$\textcircled{1}$ $S \vee R$	$\textcircled{2}$ $\neg S$	$\textcircled{1} \wedge \textcircled{2}$ $(S \vee R) \wedge \neg S$
T	T	T		
T	F	T		
F	T	T		
F	F			

Semantics of the weather problem

- ① Tomorrow will be sunny or rainy
- ② Tomorrow will not be sunny

What will the weather be tomorrow?

SunnyTomorrow	RainyTomorrow	$\textcircled{1}$ $S \vee R$	$\textcircled{2}$ $\neg S$	$\textcircled{1} \wedge \textcircled{2}$ $(S \vee R) \wedge \neg S$
T	T	T		
T	F	T		
F	T	T		
F	F	F		

Semantics of the weather problem

- ① Tomorrow will be sunny or rainy
- ② Tomorrow will not be sunny

What will the weather be tomorrow?

SunnyTomorrow	RainyTomorrow	$\textcircled{1}$ $S \vee R$	$\textcircled{2}$ $\neg S$	$\textcircled{1} \wedge \textcircled{2}$ $(S \vee R) \wedge \neg S$
T	T	T	F	
T	F	T		
F	T	T		
F	F	F		

Semantics of the weather problem

- ① Tomorrow will be sunny or rainy
- ② Tomorrow will not be sunny

What will the weather be tomorrow?

SunnyTomorrow	RainyTomorrow	$\textcircled{1}$ $S \vee R$	$\textcircled{2}$ $\neg S$	$\textcircled{1} \wedge \textcircled{2}$ $(S \vee R) \wedge \neg S$
T	T	T	F	
T	F	T	F	
F	T	T		
F	F	F		

Semantics of the weather problem

- ① Tomorrow will be sunny or rainy
- ② Tomorrow will not be sunny

What will the weather be tomorrow?

SunnyTomorrow	RainyTomorrow	$\textcircled{1}$ $S \vee R$	$\textcircled{2}$ $\neg S$	$\textcircled{1} \wedge \textcircled{2}$ $(S \vee R) \wedge \neg S$
T	T	T	F	
T	F	T	F	
F	T	T	T	
F	F	F		

Semantics of the weather problem

- ① Tomorrow will be sunny or rainy
- ② Tomorrow will not be sunny

What will the weather be tomorrow?

SunnyTomorrow	RainyTomorrow	$\textcircled{1}$ $S \vee R$	$\textcircled{2}$ $\neg S$	$\textcircled{1} \wedge \textcircled{2}$ $(S \vee R) \wedge \neg S$
T	T	T	F	
T	F	T	F	
F	T	T	T	
F	F	F	T	

Semantics of the weather problem

- ① Tomorrow will be sunny or rainy
- ② Tomorrow will not be sunny

What will the weather be tomorrow?

SunnyTomorrow	RainyTomorrow	$\textcircled{1}$ $S \vee R$	$\textcircled{2}$ $\neg S$	$\textcircled{1} \wedge \textcircled{2}$ $(S \vee R) \wedge \neg S$
T	T	T	F	F
T	F	T	F	
F	T	T	T	
F	F	F	T	

Semantics of the weather problem

- ① Tomorrow will be sunny or rainy
- ② Tomorrow will not be sunny

What will the weather be tomorrow?

SunnyTomorrow	RainyTomorrow	$\textcircled{1}$ $S \vee R$	$\textcircled{2}$ $\neg S$	$\textcircled{1} \wedge \textcircled{2}$ $(S \vee R) \wedge \neg S$
T	T	T	F	F
T	F	T	F	F
F	T	T	T	
F	F	F	T	

Semantics of the weather problem

- ① Tomorrow will be sunny or rainy
- ② Tomorrow will not be sunny

What will the weather be tomorrow?

SunnyTomorrow	RainyTomorrow	$\textcircled{1}$ $S \vee R$	$\textcircled{2}$ $\neg S$	$\textcircled{1} \wedge \textcircled{2}$ $(S \vee R) \wedge \neg S$
T	T	T	F	F
T	F	T	F	F
F	T	T	T	T
F	F	F	T	

Semantics of the weather problem

- ① Tomorrow will be sunny or rainy
- ② Tomorrow will not be sunny

What will the weather be tomorrow?

SunnyTomorrow	RainyTomorrow	$\textcircled{1}$ $S \vee R$	$\textcircled{2}$ $\neg S$	$\textcircled{1} \wedge \textcircled{2}$ $(S \vee R) \wedge \neg S$
T	T	T	F	F
T	F	T	F	F
F	T	T	T	T
F	F	F	T	F

Semantics of the weather problem

- ① Tomorrow will be sunny or rainy
- ② Tomorrow will not be sunny

What will the weather be tomorrow?

SunnyTomorrow	RainyTomorrow	$\textcircled{1}$ $S \vee R$	$\textcircled{2}$ $\neg S$	$\textcircled{1} \wedge \textcircled{2}$ $(S \vee R) \wedge \neg S$
T	T	T	F	F
T	F	T	F	F
F	T	T	T	T
F	F	F	T	F

So it will rain tomorrow.

Semantics: Definitions

Definition (Interpretation)

An *interpretation* (or *valuation*) is a **truth assignment** to the symbols in the alphabet \mathcal{L} : it is a function V from \mathcal{L} to $\{\text{T}, \text{F}\}$.

Semantics: Definitions

Definition (Interpretation)

An *interpretation* (or *valuation*) is a **truth assignment** to the symbols in the alphabet \mathcal{L} : it is a function V from \mathcal{L} to $\{\text{T}, \text{F}\}$.

An interpretation V of \mathcal{L} is extended to an interpretation of a wff P by induction on its structure:

$$\begin{array}{lll} \llbracket A \rrbracket_V & = & V(A) \\ \llbracket P \wedge Q \rrbracket_V & = & \llbracket P \rrbracket_V \text{ and } \llbracket Q \rrbracket_V \\ \llbracket P \vee Q \rrbracket_V & = & \llbracket P \rrbracket_V \text{ or } \llbracket Q \rrbracket_V \end{array} \quad \begin{array}{lll} \llbracket \neg P \rrbracket_V & = & \text{not } \llbracket P \rrbracket_V \\ \llbracket P \rightarrow Q \rrbracket_V & = & \llbracket P \rrbracket_V \text{ implies } \llbracket Q \rrbracket_V \end{array}$$

This is the Tarski definition of truth: the truth value of a compound sentence is established by breaking it down until we get to atomic propositions.

Semantics: Definitions

Definition (Satisfaction)

An interpretation V satisfies a wff P if $\llbracket P \rrbracket_V = T$

Definition (Satisfiable)

A wff is **satisfiable** if there exists an interpretation that satisfies it.

A wff is unsatisfiable if it is not satisfiable.

Definition (Valid or Tautology)

A wff is **valid** or is a **tautology** if every interpretation satisfies it.

Example

$(S \vee R) \wedge \neg S$ is **satisfiable**

(there is a state of affairs that makes it true)

$((S \vee R) \wedge \neg S) \rightarrow R$ is **valid**

(it is always true, no matter what the state of affairs)

Semantics: Definitions

Definition (Entailment)

The wffs P_1, P_2, \dots, P_n **entail** Q if for any interpretation which satisfies all of P_1, P_2, \dots, P_n also satisfies Q .

We then write $P_1, P_2, \dots, P_n \models Q$.

Note If there is **no** interpretation which satisfies all of P_1, P_2, \dots, P_n then $P_1, P_2, \dots, P_n \models Q$ for **any** Q . Contradictory assumptions entail everything!

Note Everything entails a tautology. If Q is a tautology, then $P_1, P_2, \dots, P_n \models Q$ holds not matter what P_1, P_2, \dots, P_n are.

We write $\models Q$ when Q is a tautology.

Example

Is $\neg P, Q \models Q \wedge (P \rightarrow Q)$ a valid entailment?

Proof, Inference Rules and Deductive Systems

For propositional logic, it is possible to reason on a computer directly using the semantics:

- ▶ Satisfiability, validity and entailment are decidable

So, in theory, we make conjectures and the computer checks them.

But this is not always possible:

- ▶ Propositional logic is not very expressive; but
- ▶ Expressive logics like FOL and HOL are not decidable; and
- ▶ Even for propositional logic, checking satisfiability, validity, entailment is not always feasible.

So we encode a notion of *proof*:

- ▶ A **formal deductive system** is a set of **valid inference rules** that tell us what conclusions we can draw from some premises.
- ▶ Inference rules can be applied manually or automatically.
- ▶ We will look at **Natural Deduction**, developed by Gentzen and Prawitz.

Inference Rules

An inference rule tells us how one wff can be derived in one step from zero, one, or more other wffs. We write

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{Q} (R)$$

if wff Q is derived from wffs P_1, P_2, \dots, P_n using the rule R .

Example inference rules (with their corresponding Isabelle names):

$$\frac{P \quad Q}{P \wedge Q} (\text{conjI})$$

$$\frac{P \quad P \rightarrow Q}{Q} (\text{mp})$$

The P and Q here are **meta-variables** (denoted by $?P$ and $?Q$ in Isabelle) and mp is the **modus ponens** rule of inference.

This **rule schema** characterises an infinite number of **rule instances**, obtained by substituting wffs for the P and Q . Example of an instance of mp is:

$$\frac{\overbrace{A \wedge B}^P \quad \overbrace{(A \wedge B) \rightarrow C}^P \quad \overbrace{C}^Q}{\overbrace{Q}^Q}$$

Validity

- ▶ Inference rules must be **valid**. They must preserve truth.
- ▶ Formally, for all instances of

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{Q} \text{ (R)}$$

of the rule R , we must have $P_1, P_2, \dots, P_n \models Q$.

- ▶ Inference is **transitive**. If we can infer R from Q and we can infer Q from P , then we can infer R from P . This means we can chain deductions together to form a deduction *tree*.

Introduction and Elimination

In Natural Deduction (ND), rules are split into two groups:

Introduction rules : how to derive $P \square Q$

$$\frac{P \quad Q}{P \wedge Q} \text{ (conjI)} \qquad \frac{P}{P \vee Q} \text{ (disjI1)} \qquad \frac{Q}{P \vee Q} \text{ (disjI2)}$$

Elimination rules : what can be derived from $P \square Q$?

$$\frac{P \wedge Q}{P} \text{ (conjunct1)} \qquad \frac{P \wedge Q}{Q} \text{ (conjunct2)}$$

$$\frac{\begin{array}{c} [P] \qquad [Q] \\ \vdots \qquad \vdots \\ P \vee Q \qquad R \qquad R \end{array}}{R} \text{ (disjE)}$$

A proof: distributivity of \wedge and \vee

A proof that $P \wedge (Q \vee R) \vDash (P \wedge Q) \vee (P \wedge R)$.

$$\frac{\begin{array}{c} P \wedge (Q \vee R) \\ \hline \begin{array}{c} P \\ [Q] \end{array} \end{array} \quad \begin{array}{c} P \wedge (Q \vee R) \\ \hline \begin{array}{c} P \\ [R] \end{array} \end{array}}{\begin{array}{c} P \wedge Q \\ (P \wedge Q) \vee (P \wedge R) \\ \hline (P \wedge Q) \vee (P \wedge R) \end{array}}$$

Note Each proof step will normally be annotated with the name of its associated inference rule (e.g. disjE for the bottom most step).

Ways of applying rules

Inference rules are applied in two basic ways.

- ▶ **Forward proof** if we derive new wffs from existing wffs by applying rules top down.
- ▶ **Backward proof** if we conjecture some wff true and apply rules bottom-up to produce new wffs from which the original wff is derived.

In Isabelle,

- ▶ procedural proof very often proceeds backwards, from the goal.
Forward proof is also possible, though.
- ▶ structured proof tends to be via forward reasoning.

Summary

- ▶ Propositional logic
 - ▶ Syntax (atomic propositions, wffs) (H&R 1.1+1.3)
 - ▶ Semantics (interpretations, satisfaction, satisfiability, validity, entailment) (H&R 1.4.1, first part of 1.4.2)
- ▶ Natural deduction (H&R 1.2)
 - ▶ Introduction and Elimination rules;
 - ▶ Proofs are trees, with assumptions at the leaves.
- ▶ Next time
 - ▶ More on Natural Deduction (\rightarrow , \leftrightarrow , \neg);
 - ▶ Natural deduction in Isabelle/HOL.

Automated Reasoning

Lecture 3: Natural Deduction and Starting with Isabelle

Jacques Fleuriot
`jdf@inf.ed.ac.uk`

Recap

- ▶ Last time I introduced **natural deduction**
- ▶ We saw the rules for \wedge and \vee :

$$\frac{P \quad Q}{P \wedge Q} \text{ (conjI)} \qquad \frac{P}{P \vee Q} \text{ (disjI1)} \qquad \frac{Q}{P \vee Q} \text{ (disjI2)}$$

$$\frac{P \wedge Q}{P} \text{ (conjunct1)} \qquad \frac{P \wedge Q}{Q} \text{ (conjunct2)}$$

$$\frac{\begin{array}{c} [P] \qquad [Q] \\ \vdots \qquad \vdots \\ P \vee Q \end{array}}{\frac{R \qquad R}{R}} \text{ (disjE)}$$

But what about the other connectives \rightarrow , \leftrightarrow and \neg ?

Rules for Implication

$$\frac{\begin{array}{c} [P] \\ \vdots \\ Q \end{array}}{P \rightarrow Q} \text{ (implI)}$$

IMPI forward: If on the assumption that P is true, Q can be shown to hold, then we can conclude $P \rightarrow Q$.

IMPI backward: To prove $P \rightarrow Q$, assume P is true and prove that Q follows.

$$\frac{P \rightarrow Q \quad P}{Q} \text{ (mp)}$$

The **modus ponens** rule.

$$\frac{\begin{array}{c} [Q] \\ \vdots \\ R \end{array}}{P \rightarrow Q \quad P \quad R} \text{ (impE)}$$

Another possible implication rule is this one. Note: this is not necessarily a standard ND rule but may be useful in mechanized proofs.

Rules for \leftrightarrow

$$\begin{array}{c} [Q] \qquad [P] \\ \vdots \qquad \vdots \\ \frac{P \qquad Q}{P \leftrightarrow Q} \text{ (iffI)} \qquad \frac{P \leftrightarrow Q \qquad P}{Q} \text{ (iffD1)} \\ \\ \frac{P \leftrightarrow Q \qquad Q}{P} \text{ (iffD2)} \end{array}$$

These rules are derivable from the rules for \wedge and \rightarrow , using the abbreviation $P \leftrightarrow Q \equiv (P \rightarrow Q) \wedge (Q \rightarrow P)$.

Note: In Isabelle, the \leftrightarrow is also denoted by =

Rules for False and Negation

It is convenient to introduce a 0-ary connective \perp to represent false.
The connective \perp has the rules:

no introduction rule for \perp

$$\frac{\perp}{P} \text{ (FalseE)}$$

Note \perp is written `False` in Isabelle.

$$P$$

$$\vdots$$

$$\frac{\perp}{\neg P} \text{ (notI)}$$

$$\frac{P \quad \neg P}{\perp} \text{ (notE)}$$

Note: we could *define* $\neg P$ to be $P \rightarrow \perp$

Note: In Isabelle, notE is different:

$$\frac{P \quad \neg P}{R} \text{ (notE)}$$

In this course, you can use either version in your proofs.

Proof

Recall the logic problems from lecture 2: we can now prove

$$((\text{Sunny} \vee \text{Rainy}) \wedge \neg \text{Sunny}) \rightarrow \text{Rainy}$$

which we previously knew only by semantic means.

$$\frac{\frac{\frac{[(S \vee R) \wedge \neg S]_1}{S \vee R} \quad \frac{\frac{[S]_2}{R}}{\neg S} \quad \frac{[R]_2}{R}}{R} \quad ((S \vee R) \wedge \neg S) \rightarrow R}{(disjE_2)} \quad (impI_1)$$

The subscripts $[\cdot]_1$ and $[\cdot]_2$ on the assumptions refer to the rule instances (also with subscripts) where they are discharged. This makes the proof easier to follow.

Note: For a full proof, the names of *all* the ND rules being used should be given (i.e. not just impI and disjE as in the above).

Soundness and Completeness

Theorem (Soundness)

If Q is provable from assumptions P_1, \dots, P_n , then $P_1, \dots, P_n \models Q$.

This follows because all our rules are *valid*.

Is the converse true?

Can't prove Pierce's law: $((A \rightarrow B) \rightarrow A) \rightarrow A$

Can prove it using the *law of excluded middle*: $P \vee \neg P$.

So far, our proof system is sound and complete for Intuitionistic Logic. Intuitionistic logic rejects the law of excluded middle.

Rules for classical reasoning

$$\frac{\begin{array}{c} [\neg P] \\ \vdots \\ \perp \end{array}}{P} \text{ (ccontr)}$$
$$\frac{}{\neg P \vee P} \text{ (excluded_middle)}$$

Either one suffices.

Theorem (Completeness)

If $P_1, \dots, P_n \models Q$, then Q is provable from the assumptions P_1, \dots, P_n .

Proof: more complicated, see H&R 1.4.4.

Sequents

We have been representing proofs with assumptions like so:

$$\frac{\begin{array}{c} P_2 \\ P_1 \qquad \vdots \qquad P_n \\ \vdots \qquad \vdots \qquad \dots \qquad \vdots \end{array}}{Q}$$

Another notation is sequent-style or Fitch-style:

$$P_1, P_2, \dots, P_n \vdash Q$$

The assumptions are usually collectively referred to using Γ :

$$\Gamma \vdash Q$$

This style is fiddlier on paper, but easier to prove meta-theoretic properties for, and easier to represent on a computer.

Natural Deduction Sequents

New rule: $\frac{P \in \Gamma}{\Gamma \vdash P}$ (assumption)

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \text{ (conjI)}$$

$$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \text{ (conjunct1)}$$

$$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q} \text{ (conjunct2)}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \text{ (disjI1)}$$

$$\frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q} \text{ (disjI2)}$$

$$\frac{\Gamma \vdash P \vee Q \quad \Gamma, P \vdash R \quad \Gamma, Q \vdash R}{\Gamma \vdash R} \text{ (disjE)}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \text{ (impI)}$$

$$\frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \text{ (mp)}$$

No introduction rule for \perp

$$\frac{\Gamma \vdash \perp}{\Gamma \vdash P} \text{ (FalseE)}$$

$$\frac{\Gamma, P \vdash \perp}{\Gamma \vdash \neg P} \text{ (notI)}$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash \neg P}{\Gamma \vdash \perp} \text{ (notE)}$$

$$\frac{}{\Gamma \vdash P \vee \neg P} \text{ (excluded_middle)}$$

Natural Deduction in Isabelle/HOL

Isabelle represents the sequent $P_1, P_2, \dots, P_n \vdash Q$ with the following notation:

$$P_1 \implies (P_2 \implies \dots \implies (P_n \implies Q) \dots)$$

which is also written as: $\llbracket P_1; P_2; \dots; P_n \rrbracket \implies Q$

Note: To enable the bracket notation for sequents in Isabelle, select: Plugins → Plugin Options in the Isabelle JEdit menu bar. Then select Isabelle → General and enter *brackets* in the Print Mode box.

The symbol \implies is *meta-implication*.

Meta-implication is used to represent the relationship between premises and conclusions of rules.

$$\frac{\begin{array}{c} [P] \\ \vdots \\ Q \end{array}}{P \rightarrow Q} \quad \text{is written as} \quad (?P \implies ?Q) \implies (?P \rightarrow ?Q)$$

Natural Deduction Rules in Isabelle

A selection of natural deduction rules in Isabelle notation:

$$\frac{P \quad Q}{P \wedge Q} \text{ (conjI)}$$

$$[\![?P; ?Q]\!] \implies ?P \wedge ?Q$$

$$\frac{P \wedge Q}{P} \text{ (conjunct1)}$$

$$[\![?P \wedge ?Q]\!] \implies ?P$$

$$\frac{P}{P \vee Q} \text{ (disjI1)}$$

$$[\![?P]\!] \implies ?P \vee ?Q$$

$$\begin{array}{c} [P] \\ \vdots \end{array} \qquad \begin{array}{c} [Q] \\ \vdots \end{array}$$

$$\frac{\begin{array}{ccc} P \vee Q & R & R \end{array}}{R} \text{ (disjE)} \quad \begin{array}{c} [\![?P \vee ?Q; ?P \implies ?R; ?Q \implies ?R]\!] \\ \implies ?R \end{array}$$

Doing Proofs in Isabelle: Theory Set-up

Syntax:

```
theory MyTh
imports T1 ... Tn
begin
(definitions, theorems, proofs, ...)*
end
```

MyTh: name of theory. Must live in file *MyTh.thy*

T_i: names of *imported* theories. Import is transitive.

Often: imports Main

Doing Proofs in Isabelle

A declaration like so enters proof mode:

theorem K: " $A \rightarrow B \rightarrow A$ "

Isabelle responds:

proof (prove)

goal (1 subgoal):

1. $A \rightarrow B \rightarrow A$

We now apply proof methods (tactics) that affect the subgoals.
Either:

- ▶ generate new subgoal(s), breaking the problem down; or
- ▶ solve the subgoal

When there are no more subgoals, then the proof is complete.

The assumption Method

Given a subgoal of the form:

$$[\![A; B]\!] \implies A$$

This subgoal is solvable because we want to prove A under the assumption that A is true.

We can solve this subgoal using the assumption method:

apply assumption

The rule Method

To apply an inference rule backward, we use `rule`.

Consider the theorem `disjI1`

$$?P \implies ?P \vee ?Q$$

Using the command

`apply (rule disjI1)`

on the goal

$$[A; B; C] \implies (A \wedge B) \vee D$$

yields the subgoal

$$[A; B; C] \implies A \wedge B$$

Using `rule` can be viewed as a way of breaking down the problem into subproblems.

Matching and Unification

In applying rule (with the ? in front of variables omitted)

$$P \implies \textcolor{red}{P} \vee \textcolor{blue}{Q}$$

to goal

$$[\![A; B; C]\!] \implies (\textcolor{red}{A} \wedge \textcolor{red}{B}) \vee \textcolor{blue}{D}$$

The pattern $P \vee Q$ is **matched** with the target $(A \wedge B) \vee D$ to yield the instantiations $P \mapsto A \wedge B$, $Q \mapsto D$ which make the pattern and target the same. The following goal results

$$[\![A; B; C]\!] \implies A \wedge B$$

In general, if the goal conclusion contains schematic variables, the rule and goal conclusions are **unified** i.e. both are instantiated so as to make them the same.

More on **unification** later!

Summary

- ▶ More natural deduction (H&R 1.2, 1.4)
 - ▶ The rules for \rightarrow , \leftrightarrow and \neg
 - ▶ Rules for classical reasoning
 - ▶ Soundness and completeness properties
 - ▶ Sequent-style presentation
- ▶ Starting with proofs in Isabelle
- ▶ Next time:
 - ▶ More on using Isabelle to do proofs
 - ▶ N-style vs. L-style proof systems

Automated Reasoning

Lecture 4: Propositional Reasoning in Isabelle

Jacques Fleuriot
`jdf@inf.ed.ac.uk`

Recap

Last lecture:

- ▶ Completed the natural deduction system for propositional logic
- ▶ Started on proving propositions in Isabelle

Today:

- ▶ More details on proving propositions in Isabelle
- ▶ Alternative inference rules (*L*-system, a.k.a. “Sequent Calculus”)
- ▶ Why should we trust Isabelle?

The rule Method

To apply an inference rule, we use `rule`.

Consider the theorem `disjI1`

$$?P \implies ?P \vee ?Q$$

Using the command

```
apply (rule disjI1)
```

on the goal

$$[A; B; C] \implies (A \wedge B) \vee D$$

yields the subgoal

$$[A; B; C] \implies A \wedge B$$

General definition of method rule

When we apply the method rule `someRule` where

$$\text{someRule} : \llbracket P_1; \dots; P_m \rrbracket \implies Q$$

to the goal

$$\llbracket A_1; \dots; A_n \rrbracket \implies C$$

where Q and C can be unified, we generate the goals

$$\llbracket A'_1; \dots; A'_n \rrbracket \implies P'_1$$

⋮

$$\llbracket A'_1; \dots; A'_n \rrbracket \implies P'_m$$

where $A'_1, A'_2, \dots, A'_n, P'_1, P'_2, \dots, P'_m$ are the results of applying the substitution which unifies Q and C to $A_1, A_2, \dots, A_n, P_1, P_2, \dots, P_m$.

We must now derive each of the rule's assumptions using our goal's assumptions.

A Problem with rule

Consider the `disjE` rule:

$$\text{disjE} : \llbracket P \vee Q; P \implies R; Q \implies R \rrbracket \implies R$$

If we have the goal:

$$\llbracket (A \wedge B) \vee C; D \rrbracket \implies B \vee C$$

Then applying rule `disjE` produces three new goals:

$$\llbracket (A \wedge B) \vee C; D \rrbracket \implies ?P \vee ?Q$$

$$\llbracket (A \wedge B) \vee C; D; ?P \rrbracket \implies B \vee C$$

$$\llbracket (A \wedge B) \vee C; D; ?Q \rrbracket \implies B \vee C$$

We then solve the first subgoal by applying `assumption`.

This seems pointlessly roundabout... we often want to *use* one of our assumptions in our proof.

The erule Method

Used when the conclusion of theorem matches the conclusion of the current goal and the first premise of theorem matches a premise of the current goal.

Consider the theorem `disjE`

$$[(P \vee Q; P \implies R; Q \implies R) \implies R]$$

Applying `erule disjE` to goal

$$[(A \wedge B) \vee C; D] \implies B \vee C$$

yields the subgoals

$$[D; (A \wedge B)] \implies B \vee C \quad [D; C] \implies B \vee C$$

General definition of method erule

When we apply the method `erule someRule` where

$$\text{someRule} : \llbracket P_1; \dots; P_m \rrbracket \implies Q$$

to the goal

$$\llbracket A_1; \dots; A_n \rrbracket \implies C$$

where P_1 and A_1 are unifiable and Q and C are unifiable, we generate the goals:

$$\llbracket A'_2; \dots; A'_n \rrbracket \implies P'_2$$

⋮

$$\llbracket A'_2; \dots; A'_n \rrbracket \implies P'_m$$

where $A'_2, \dots, A'_n, P'_2, \dots, P'_m$ are the results of applying the substitution which unifies P_1 to A_1 and Q to C to $A_2, \dots, A_n, P_2, \dots, P_m$.

We **eliminate** an assumption from the rule and the goal, and must derive the rule's other assumptions using our goal's other assumptions.

General definition of method drule

When we apply the method `drule someRule` where

$$\text{someRule} : \llbracket P_1; \dots; P_m \rrbracket \implies Q$$

to the goal

$$\llbracket A_1; \dots; A_n \rrbracket \implies C$$

where P_1 and A_1 are unifiable, we generate the goals:

$$\llbracket A'_2; \dots; A'_n \rrbracket \implies P'_2$$

\vdots

$$\begin{aligned} &\llbracket A'_2; \dots; A'_n \rrbracket \implies P'_m \\ &\llbracket Q'; A'_2; \dots; A'_n \rrbracket \implies C' \end{aligned}$$

where $A'_2, A'_3, \dots, A'_n, P'_2, P'_3, \dots, P'_m, Q', C'$ are the results of applying the substitution which unifies P_1 and A_1 to $A_2, A_3, \dots, A_n, P_2, P_3, \dots, P_m, Q, C$.

We **delete** an assumption, replacing it with the conclusion of the rule.

General definition of method `frule`

When we apply the method `frule someRule` where

$$\text{someRule} : \llbracket P_1; \dots; P_m \rrbracket \implies Q$$

to the goal

$$\llbracket A_1; \dots; A_n \rrbracket \implies C$$

where P_1 and A_1 are unifiable, we generate the goals:

$$\llbracket A'_1; A'_2; \dots; A'_n \rrbracket \implies P'_2$$

\vdots

$$\llbracket A'_1; A'_2; \dots; A'_n \rrbracket \implies P'_m$$

$$\llbracket Q'; A'_1; A'_2; \dots; A'_n \rrbracket \implies C'$$

where $A'_1, A'_2, \dots, A'_n, P'_2, \dots, P'_m, Q', C'$ are the results of applying the substitution which unifies P_1 and A_1 to $A_1, A_2, \dots, A_n, P_2, \dots, P_m, Q, C$.

This is like `drule` except the assumption in our goal is kept.

More Methods

- ▶ `rule_tac`, `erule_tac`, `drule_tac` and `frule_tac` are like their counterparts, but you can give substitutions for variables in the rule before they are applied.

Example

```
apply (erule_tac Q=" $B \wedge D$ " in conjE)
```

applied to the subgoal

$$[\![A \wedge B; C \wedge B \wedge D]\!] \implies B \wedge D$$

generates the new goal

$$[\![A \wedge B; C; B \wedge D]\!] \implies B \wedge D$$

- ▶ Isabelle also provides advanced tactics, like `simp` and `auto` which perform some **automatic deduction**.

L-systems/Sequent Calculus

The `erule` tactic points to another way of phrasing a system of inference rules in a system with sequents $\Gamma \vdash A$.

Instead of *elimination* rules:

$$\frac{\Gamma \vdash P \vee Q \quad \Gamma, P \vdash R \quad \Gamma, Q \vdash R}{\Gamma \vdash R} \text{ (disjE)}$$

Have *left introduction rules* (all the introduction rules in natural deduction introduce connectives on the right-hand side of the \vdash):

$$\frac{\Gamma, P \vdash R \quad \Gamma, Q \vdash R}{\Gamma, P \vee Q \vdash R}$$

This corresponds to applying rules using `erule` in Isabelle.

The *left introduction rules* are often much easier to use in a backwards, goal-directed style.

L-systems/Sequent Calculus

The following *L*-System (a.k.a. Sequent Calculus) rules are an alternative sound and complete proof system for propositional logic:

$$\frac{}{\Gamma, P \vdash P} \text{ (assumption)}$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \text{ (conjI)}$$

$$\frac{\Gamma, P, Q \vdash R}{\Gamma, P \wedge Q \vdash R} \text{ (e conjE)}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \text{ (disjI1)}$$

$$\frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q} \text{ (disjI2)}$$

$$\frac{\Gamma, P \vdash R \quad \Gamma, Q \vdash R}{\Gamma, P \vee Q \vdash R} \text{ (e disjE)}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \text{ (impI)}$$

$$\frac{\Gamma \vdash P \quad \Gamma, Q \vdash R}{\Gamma, P \rightarrow Q \vdash R} \text{ (e impE)}$$

no right-intro rule for \perp

$$\frac{}{\Gamma, \perp \vdash P} \text{ (e FalseE)}$$

$$\frac{\Gamma, P \vdash \perp}{\Gamma \vdash \neg P} \text{ (notI)}$$

$$\frac{\Gamma, P, \neg P \vdash R}{\Gamma \vdash \neg P} \text{ (e notE)}$$

$$\frac{}{\Gamma \vdash P \vee \neg P} \text{ (excluded_middle)}$$

Note: `e someRule` is short for `erule someRule`.

Note: in the above presentation left-hand-sides are *sets* of formulas.

An Old Friend Revisited

$\frac{S, \neg S \vdash R \quad (\text{e notE}) \quad R, \neg S \vdash R \quad (\text{assumption})}{(S \vee R), \neg S \vdash R \quad (\text{e disjE})}$
$\frac{(S \vee R), \neg S \vdash R \quad (\text{e disjE})}{(S \vee R) \wedge \neg S \vdash R \quad (\text{e ConjE})}$

Re-using proofs: The Cut rule

So far, all proofs have been self-contained; they have only used the pre-existing rules of inference.

By the completeness theorem, this suffices to prove everything that is true, but can lead to extremely repetitive proofs.

Re-using proofs: The Cut rule

So far, all proofs have been self-contained; they have only used the pre-existing rules of inference.

By the completeness theorem, this suffices to prove everything that is true, but can lead to extremely repetitive proofs.

The cut rule: (we “cut” P into the proof)

$$\frac{\Gamma \vdash P \quad \Gamma, P \vdash Q}{\Gamma \vdash Q}$$

allows the use of a *lemma* P in a proof of Q . We can now reuse P multiple times in the proof of Q .

Re-using proofs: The Cut rule

So far, all proofs have been self-contained; they have only used the pre-existing rules of inference.

By the completeness theorem, this suffices to prove everything that is true, but can lead to extremely repetitive proofs.

The cut rule: (we “cut” P into the proof)

$$\frac{\Gamma \vdash P \quad \Gamma, P \vdash Q}{\Gamma \vdash Q}$$

allows the use of a *lemma* P in a proof of Q . We can now reuse P multiple times in the proof of Q .

In Isabelle:

- `cut_tac lemmaName` – adds the conclusion of `lemmaName` as a new assumption, and its assumptions as new subgoals
- `subgoal_tac P` – adds P as a new assumption, and introduces P as a new subgoal.

Why should you believe Isabelle?

When Isabelle says “No subgoals!” why should we believe that we have *really* proved something? Is Isabelle sound?

It is doing non-trivial work behind the scenes: unification, rewriting, maintaining a database of theorems+assumptions, automatic proof.

Why should you believe Isabelle?

When Isabelle says “No subgoals!” why should we believe that we have *really* proved something? Is Isabelle sound?

It is doing non-trivial work behind the scenes: unification, rewriting, maintaining a database of theorems+assumptions, automatic proof.

Isabelle uses two strategies to maintain soundness:

- ▶ A small trusted kernel: internally, every proof is broken down into primitive rule applications which are checked by a small piece of hand-verified code. This is the “LCF” model. So new *proof procedures* cannot introduce unsoundness.
- ▶ Encourages *definitional extension of the logic*: new concepts are introduced by definition rather than axiomatisation (more on this in Lecture 6). So new definitions cannot introduce unsoundness.

Why should you believe Isabelle?

When Isabelle says “No subgoals!” why should we believe that we have *really* proved something? Is Isabelle sound?

It is doing non-trivial work behind the scenes: unification, rewriting, maintaining a database of theorems+assumptions, automatic proof.

Isabelle uses two strategies to maintain soundness:

- ▶ A small trusted kernel: internally, every proof is broken down into primitive rule applications which are checked by a small piece of hand-verified code. This is the “LCF” model. So new *proof procedures* cannot introduce unsoundness.
- ▶ Encourages *definitional extension of the logic*: new concepts are introduced by definition rather than axiomatisation (more on this in Lecture 6). So new definitions cannot introduce unsoundness.

Threats to (practical) soundness still exist, including: Have we proved what we thought we proved? Are the formulas displayed on screen correctly? ...

See: Pollack, R. *How to Believe a Machine-Checked Proof*, 1997 (non-examinable).

Summary

- ▶ More tools for proving propositions in Isabelle
 - ▶ The `erule`, `drule`, `frule` methods
 - ▶ Their `_tac` variants
 - ▶ *L*-systems, and Cut rules (`cut_tac`, `subgoal_tac`).
 - ▶ See the propositional logic exercises and examples:
 - ▶ Tutorial 1 and Additional Exercise on the AR webpage;
 - ▶ The Isabelle theory file `Prop.thy`;
 - ▶ Start using Isabelle (if you haven't done so already).
- ▶ How Isabelle maintains soundness
 - ▶ Small trusted kernel
 - ▶ Definitional extension instead of axiomatic extension
- ▶ Next time:
 - ▶ First-Order Logic: $\forall x.P$ and $\exists x.P$

Automated Reasoning

Lecture 5: First-Order Logic

Jacques Fleuriot
`jdf@inf.ac.uk`

Recap

- ▶ Over the last three lectures, we have looked at:
 - ▶ Propositional logic, semantics and proof systems
 - ▶ Doing propositional logic proofs in Isabelle
- ▶ Today:
 - ▶ Syntax and Semantics of First-Order Logic (FOL)
 - ▶ Natural Deduction rules for FOL
 - ▶ Doing FOL proofs in Isabelle

Problem

Consider the following problem:

1. If someone cheats then everyone loses the game.
2. If everyone who cheats also loses, then I lose the game.
3. Did I lose the game?

Is Propositional Logic rich enough to formally represent and reason about this problem?

The finer logical structure of this problem would not be captured by the constructs we have so far encountered.

We need a richer language!

A Richer Language

First-order (predicate) logic (FOL) extends propositional logic:

- ▶ Atomic formulas are assertions about *properties of individual(s)*.
e.g. an individual might have the property of being a cheat.
- ▶ We can use *variables* to denote arbitrary individuals.
e.g. x is a cheater.
- ▶ We can *bind* variables with quantifiers \forall (for all) and \exists (exists).
e.g. for all x , x is a cheater.
- ▶ We can use connectives to compose formulas:
e.g. for all x , if x is a cheater then x loses.
- ▶ We can use quantifiers on subformulas.
e.g. we can formally distinguish between: "if *anyone* cheats we lose the game" and "if *everyone* cheats, we lose the game".

Terms of First-Order Logic

Given:

- ▶ a countably infinite set of (individual) variables
 $\mathcal{V} = \{x, y, z, \dots\};$
- ▶ a finite or countably infinite set of function letters \mathcal{F} each assigned a unique arity (possibly 0)

then the set of (well-formed) terms is the smallest set such that

- ▶ any variable $v \in \mathcal{V}$ is a term;
- ▶ if $f \in \mathcal{F}$ has arity n , and t_1, \dots, t_n are terms, so is $f(t_1, \dots, t_n)$.

Remarks

- ▶ If f has arity 0, we usually write f rather than $f()$, and call f a **constant**
- ▶ In practice, we use infix notation when appropriate: e.g., $x + y$ instead of $+(x, y)$.

Formulas of First-Order Logic

Given a countably infinite set of predicates \mathcal{P} , each assigned a unique arity (possibly 0), the set of wffs is the smallest set such that

- ▶ if $A \in \mathcal{P}$ has arity n , and t_1, \dots, t_n are terms, then $A(t_1, \dots, t_n)$ is a wff;
- ▶ if P and Q are wffs, so are $\neg P, P \vee Q, P \wedge Q, P \rightarrow Q, P \leftrightarrow Q$,
- ▶ if P is a wff, so are $\exists x. P$ and $\forall x. P$ for any $x \in \mathcal{V}$;
- ▶ if P is a wff, then (P) is a wff.

Remarks

- ▶ If A has arity 0, we usually write A rather than $A()$, and call A a **propositional variable**. This way, propositional logic wffs look like a subset of FOL wffs. Also, use infix notation where appropriate.
- ▶ We assume $\exists x$ and $\forall x$ bind more weakly than any of the propositional connectives.

$\exists x. P \wedge Q$ is $\exists x. (P \wedge Q)$, not $(\exists x. P) \wedge Q$.
(NB: H&R assume $\exists x$ and $\forall x$ bind like \neg .)

Example: Problem Revisited

We can now formally represent our problem in FOL:

- ▶ **Assumption 1:** If someone cheats then everyone loses the game: $(\exists x. \text{Cheats}(x)) \rightarrow \forall x. \text{Loses}(x)$.
- ▶ **Assumption 2:** If everyone who cheats also loses, then I lose the game : $(\forall x. \text{Cheats}(x) \rightarrow \text{Loses}(x)) \rightarrow \text{Loses(me)}$.

To answer the question *Did I lose the game?* we need to prove either Loses(me) or $\neg \text{Loses(me)}$ from these assumptions.

More on this later.

Free and Bound Variables

- ▶ An occurrence of a variable x in a formula P is **bound** if it is in the scope of a $\forall x$ or $\exists x$ quantifier.
- ▶ A variable occurrence x is **in the scope of** a quantifier occurrence $\forall x$ or $\exists x$ if the quantifier occurrence is the first occurrence of a quantifier over x in a traversal from the variable occurrence position to the root of the formula tree.
- ▶ If a variable occurrence is **not bound**, it is **free**

Example

In

$$P(x) \wedge (\forall x. P(y) \rightarrow P(x))$$

The first occurrence of x and the occurrence of y are free, while the second occurrence of x is bound.

Free and Bound Variables

- ▶ An occurrence of a variable x in a formula P is **bound** if it is in the scope of a $\forall x$ or $\exists x$ quantifier.
- ▶ A variable occurrence x is **in the scope** of a quantifier occurrence $\forall x$ or $\exists x$ if the quantifier occurrence is the first occurrence of a quantifier over x in a traversal from the variable occurrence position to the root of the formula tree.
- ▶ If a variable occurrence is **not** bound, it is **free**

Example

In

$$P(\textcolor{red}{x}) \wedge (\forall x. P(y) \rightarrow P(x))$$

↑
free

The first occurrence of x and the occurrence of y are free, while the second occurrence of x is bound.

Free and Bound Variables

- ▶ An occurrence of a variable x in a formula P is **bound** if it is in the scope of a $\forall x$ or $\exists x$ quantifier.
- ▶ A variable occurrence x is **in the scope of** a quantifier occurrence $\forall x$ or $\exists x$ if the quantifier occurrence is the first occurrence of a quantifier over x in a traversal from the variable occurrence position to the root of the formula tree.
- ▶ If a variable occurrence is **not bound**, it is **free**

Example

In

$$P(\boxed{x}) \wedge (\forall x. P(\boxed{y}) \rightarrow P(x))$$

free free

The first occurrence of x and the occurrence of y are free, while the second occurrence of x is bound.

Free and Bound Variables

- ▶ An occurrence of a variable x in a formula P is **bound** if it is in the scope of a $\forall x$ or $\exists x$ quantifier.
- ▶ A variable occurrence x is **in the scope** of a quantifier occurrence $\forall x$ or $\exists x$ if the quantifier occurrence is the first occurrence of a quantifier over x in a traversal from the variable occurrence position to the root of the formula tree.
- ▶ If a variable occurrence is **not** bound, it is **free**

Example

In

$$P(\boxed{x}) \wedge (\forall x. P(\boxed{y}) \rightarrow P(\boxed{x}))$$

free free bound

The first occurrence of x and the occurrence of y are free, while the second occurrence of x is bound.

Substitution Rules

If P is a formula, s is a term and x is a variable, then

$$P[s/x]$$

is the formula obtained by substituting s for all free occurrences of x throughout P .

Example

$$\begin{aligned} (\exists x. P(x, y)) [3/y] &\equiv \exists x. P(x, 3) \\ (\exists x. P(x, y)) [2/x] &\equiv \exists x. P(x, y). \end{aligned}$$

If necessary, bound variables in P must be **renamed** to avoid capture of free variables in s .

$$(\exists x. P(x, y)) [f(x)/y] = \exists z. P(z, f(x))$$

Semantics of First-Order Logic Formulas

(Recall that an *interpretation* for propositional logic maps atomic propositions to truth values.)

Informally, an **interpretation** of a formula maps its function letters to actual functions, and its predicate symbols to actual predicates.

The interpretation also **specifies some domain of discourse \mathcal{D}** (a non-empty set or universe) on which the functions and relations are defined.

A formal definition requires some work!

Semantics of FOL Formulas (II)

Definition (Interpretation)

Let \mathcal{F} be a set of function symbols and \mathcal{P} be a set of predicate symbols.

An **interpretation** \mathcal{I} consists of a **non-empty set** \mathcal{D} of concrete values, called the domain of the interpretation, together with the following mappings

1. each **predicate symbol** $P \in \mathcal{P}$ of arity $n > 0$ is assigned to a subset $P^{\mathcal{I}} \subseteq \mathcal{D}^n$ of n -tuples of \mathcal{D} . Each **nullary predicate** is assigned either T or F.
2. Each **function symbol** $f \in \mathcal{F}$ of arity $n > 0$ is assigned to a concrete function $f^{\mathcal{I}} : \mathcal{D}^n \rightarrow \mathcal{D}$. Each **nullary function (constant)** is assigned to a concrete value in \mathcal{D} .

Example of Interpretation

Consider the following statement, containing constant a :

$$P(a) \wedge \exists x. Q(a, x) \quad (*)$$

In one possible interpretation \mathcal{I} :

- ▶ the domain \mathcal{D} is the set of natural numbers $\mathbb{N} = \{0, 1, 2, 3, \dots\}$;
- ▶ Mappings:
 - ▶ 2 to a i.e. $a^{\mathcal{I}} \equiv 2$,
 - ▶ the property of being even to P i.e. $P^{\mathcal{I}} \equiv \{0, 2, 4, \dots\}$, and
 - ▶ the relation of being greater than to Q , i.e. the set of pairs $Q^{\mathcal{I}} \equiv \{(1, 0), \dots, (2, 0), (2, 1), \dots, (89, 27), \dots\}$;
- ▶ under this interpretation: $(*)$ affirms that 2 is even and there exists a natural number that 2 is greater than. Is $(*)$ satisfied under this interpretation?

Example of Interpretation

Consider the following statement, containing constant a :

$$P(a) \wedge \exists x. Q(a, x) \quad (*)$$

In one possible interpretation \mathcal{I} :

- ▶ the domain \mathcal{D} is the set of natural numbers $\mathbb{N} = \{0, 1, 2, 3, \dots\}$;
- ▶ Mappings:
 - ▶ 2 to a i.e. $a^{\mathcal{I}} \equiv 2$,
 - ▶ the property of being even to P i.e. $P^{\mathcal{I}} \equiv \{0, 2, 4, \dots\}$, and
 - ▶ the relation of being greater than to Q , i.e. the set of pairs $Q^{\mathcal{I}} \equiv \{(1, 0), \dots, (2, 0), (2, 1), \dots, (89, 27), \dots\}$;
- ▶ under this interpretation: $(*)$ affirms that 2 is even and there exists a natural number that 2 is greater than. Is $(*)$ satisfied under this interpretation? — Yes.
- ▶ Such a satisfying interpretation is sometimes known as a **model**.
Note: In H&R, a model is *any* interpretation (so be careful).

Semantics of FOL Formulas (III)

Definition (Assignment)

Given an interpretation \mathcal{I} , an *assignment* s assigns a value from the domain \mathcal{D} to each variable in \mathcal{V} i.e. $s : \mathcal{V} \rightarrow \mathcal{D}$.

We extend this assignment s to all terms inductively by saying that

1. if \mathcal{I} maps the n -ary function letter f to the function $f^{\mathcal{I}}$, and
2. if terms t_1, \dots, t_n have been assigned concrete values
 $a_1, \dots, a_n \in \mathcal{D}$

then we can assign value $f^{\mathcal{I}}(a_1, \dots, a_n) \in \mathcal{D}$ to the term
 $f(t_1, \dots, t_n)$.

An assignment s of values to **variables** is also commonly known as an **environment** and we denote by $s[x \mapsto a]$ the environment that maps $x \in \mathcal{V}$ to a (and any other variable $y \in \mathcal{V}$ to $s(y)$).

Remark: The interpretation I is fixed before we interpret a formula, but the assignment s will vary as we interpret the quantifiers.

Semantics of FOL Formulas (IV)

Definition (Satisfaction)

Given an interpretation \mathcal{I} and an assignment $s : \mathcal{V} \rightarrow \mathcal{D}$

1. any wff which is a nullary predicate letter A is satisfied if and only if the interpretation in \mathcal{I} of A is T ;
2. suppose we have a wff P of the form $A(t_1 \dots t_n)$, where A is interpreted as relation $A^{\mathcal{I}}$ and t_1, \dots, t_n have been assigned concrete values a_1, \dots, a_n by s . Then P is satisfied if and only if $(a_1, \dots, a_n) \in A^{\mathcal{I}}$;
3. any wff of the form $\forall x.P$ is satisfied if and only if P is satisfied with respect to assignment $s[x \mapsto a]$ for all $a \in \mathcal{D}$;
4. any wff of the form $\exists x.P$ is satisfied if and only if P is satisfied with respect to assignment $s[x \mapsto a]$ for some $a \in \mathcal{D}$;
5. any wffs of the form $P \vee Q, P \wedge Q, P \rightarrow Q, P \leftrightarrow Q, \neg P$ are satisfied according to the truth-tables for each connective (e.g. $P \vee Q$ is satisfied if and only if P is satisfied or Q is satisfied).

Example: Assignment and Satisfaction

Consider the wff ϕ :

$$R(f(x), g(y, a))$$

where $x, y \in \mathcal{V}$ i.e. are variables and a is a nullary function symbol i.e. a constant.

Given the interpretation \mathcal{I} where

- ▶ the domain \mathcal{D} is the set of integers \mathbb{Z}
- ▶ $a^{\mathcal{I}} \equiv -5$
- ▶ $R^{\mathcal{I}} \equiv <$ (less than)
- ▶ $f^{\mathcal{I}} \equiv -$ (minus)
- ▶ $g^{\mathcal{I}} \equiv +$ (addition)

and the environment $s[x \mapsto 3, y \mapsto 2]$ then under this interpretation and assignment:

Example: Assignment and Satisfaction

Consider the wff ϕ :

$$R(f(x), g(y, a))$$

where $x, y \in \mathcal{V}$ i.e. are variables and a is a nullary function symbol i.e. a constant.

Given the interpretation \mathcal{I} where

- ▶ the domain \mathcal{D} is the set of integers \mathbb{Z}
- ▶ $a^{\mathcal{I}} \equiv -5$
- ▶ $R^{\mathcal{I}} \equiv <$ (less than)
- ▶ $f^{\mathcal{I}} \equiv -$ (minus)
- ▶ $g^{\mathcal{I}} \equiv +$ (addition)

and the environment $s[x \mapsto 3, y \mapsto 2]$ then under this interpretation and assignment:

$$\phi^{\mathcal{I}} \equiv -3 < 2 + -5 = -3 < -3$$

is not satisfied.

Example: Satisfaction & Validity

Consider the following statement:

$$\forall x \ y. \ R(x, y) \rightarrow \exists z. R(x, z) \wedge R(z, y)$$

1. Is it satisfiable?
2. Is it valid?

Example: Satisfaction & Validity

Consider the following statement:

$$\forall x \ y. R(x, y) \rightarrow \exists z. R(x, z) \wedge R(z, y)$$

1. Is it satisfiable?
2. Is it valid?

Answers:

Example: Satisfaction & Validity

Consider the following statement:

$$\forall x \ y. \ R(x, y) \rightarrow \exists z. R(x, z) \wedge R(z, y)$$

1. Is it satisfiable?
2. Is it valid?

Answers:

1. Yes: Domain is the real numbers and R is interpreted as the less-than relation.

Example: Satisfaction & Validity

Consider the following statement:

$$\forall x \ y. \ R(x, y) \rightarrow \exists z. R(x, z) \wedge R(z, y)$$

1. Is it satisfiable?
2. Is it valid?

Answers:

1. Yes: Domain is the real numbers and R is interpreted as the less-than relation.
2. No: Domain? Interpretation for R ?

Semantics of FOL Formulas (V)

Definition (Entailment)

We write $I \models_s P$ to mean that wff P is satisfied by interpretation I and assignment s .

We say that the wffs P_1, P_2, \dots, P_n entail wff Q and write

$$P_1, P_2, \dots, P_n \models Q$$

if, for any interpretation I and assignment s for which $I \models_s P_i$ for all i , we also have $I \models_s Q$.

As with propositional logic, we must ensure that our inference rules are *valid*. That is, if

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{Q}$$

then we must have $P_1, P_2, \dots, P_n \models Q$.

More Introduction Rules

We now consider the additional natural deduction rules for FOL.

The introduction rules for the quantifiers are:

- ▶ Universal quantification: Provided that x_0 is **not** free in the assumptions,

$$\frac{P[x_0/x]}{\forall x. P} \text{ (allI)}$$

- ▶ Existential quantification:

$$\frac{P[t/x]}{\exists x. P} \text{ (exI)}$$

Existential Elimination

$$\frac{\begin{array}{c} [P[x_0/x]] \\ \vdots \\ \exists x.P \qquad Q \\ \hline Q \end{array}}{\text{(exE)}}$$

Provided x_0 does **not** occur in Q or any assumption other than $P[x_0/x]$ on which the derivation of Q from $P[x_0/x]$ depends.

Universal Elimination

Specialisation rule:

$$\frac{\forall x.P}{P[t/x]} \text{ (spec)}$$

An alternative universal elimination rule is allE:

$$\frac{\begin{array}{c} [P[t/x]] \\ \vdots \\ \forall x.P \end{array}}{\frac{Q}{Q}} \text{ (allE)}$$

Example Proof

Prove that $\exists y. P(y)$ is true, given that $\forall x. P(x)$ holds.

$$\frac{\frac{\forall x.P(x)}{P(a)} \text{ (spec)}}{\exists y.P(y)} \text{ (exI)}$$

Example Proof

Prove that $\exists y. P(y)$ is true, given that $\forall x. P(x)$ holds.

$$\frac{\frac{\forall x. P(x)}{P(a)} \text{ (spec)}}{\exists y. P(y)} \text{ (exI)}$$

Side note: semantically, we implicitly use the fact that our domain of discourse is non-empty. It doesn't matter what a is, but we have to have something.

Why the side conditions on allI and exE?

A (non-)proof of: $\vdash x > 5 \rightarrow \forall x. x > 5$:

$$\frac{\frac{x > 5 \vdash x > 5 \quad (\text{assumption})}{x > 5 \vdash \forall x. x > 5} \quad (\text{allI})}{\vdash x > 5 \rightarrow \forall x. x > 5} \quad (\text{impl})$$

But it is clearly false that if a particular x is greater than 5, then every x is greater than 5. We have “proven” that $x > 5$, but not for an **arbitrary** x , only for the **particular** x we had already made an assumption about.

Exercise: Give a non-proof for exE.

Machine assistance: Isabelle keeps track of which variable names are allowed where, so we can only apply the rules in a sound way.

Example Proof (II)

Prove that $\forall x. Q(x)$ is true, given $\forall x. P(x)$ and $(\forall x. P(x) \rightarrow Q(x))$.

$$\frac{\frac{\frac{\frac{\frac{\forall x. P(x)}{\forall x. P(x) \rightarrow Q(y)}}{Q(y)} \quad [P(y) \rightarrow Q(y)]_1 \quad [P(y)]_2}{Q(y)} \quad (\text{allE}_2)}{Q(y)} \quad (\text{allE}_1)}{\forall x. Q(x)} \quad (\text{allI})$$

Note: Subscripts are attached to rules being applied and the assumption(s) that they introduce e.g. allE₁ and $[P(y) \rightarrow Q(y)]_1$.

Problem (III)

Prove that $\text{Loses}(me)$ given that

1. $(\exists x. \text{Cheats}(x)) \rightarrow \forall x. \text{Loses}(x)$
2. $(\forall x. \text{Cheats}(x) \rightarrow \text{Loses}(x)) \rightarrow \text{Loses}(me)$

	$\frac{\text{assumption1} \quad \frac{[\text{Cheats}(y)]_1}{\exists x. \text{Cheats}(x)} \text{ (exI)}}{\forall x. \text{Loses}(x)} \text{ (mp)}$	
	$\frac{\forall x. \text{Loses}(x) \quad \frac{\text{Loses}(y)}{\frac{\text{Cheats}(y) \rightarrow \text{Loses}(y)}{\text{Cheats}(y) \rightarrow \text{Loses}(y)}} \text{ (spec)}}{\text{Cheats}(y) \rightarrow \text{Loses}(y)} \text{ (impI}_1\text{)}$	
assumption2	$\frac{\frac{\text{Cheats}(y) \rightarrow \text{Loses}(y)}{\forall x. \text{Cheats}(x) \rightarrow \text{Loses}(x)}}{\text{Loses}(me)} \text{ (allI)}$	
		(mp)

FOL in Isabelle/HOL

Isabelle's HOL object logic is richer than the FOL so far presented. One difference is that all variables, terms and formulas have **types**.

The type language is built using

- ▶ **base types** such as *bool* (the type of truth values) and *nat* (the type of natural numbers).
- ▶ **type constructors** such as *list* and *set* which are written postfix, e.g., *nat list* or *nat set*.
- ▶ **function types** written using \Rightarrow ; e.g.

$$nat \times nat \Rightarrow nat$$

which is a function taking two arguments of type *nat* and returning an object of type *nat*.

- ▶ **type variables** such as '*a*', '*b*' etc. These give rise to polymorphic types such as '*a* \Rightarrow '*a*'.

FOL in Isabelle/HOL (II)

- ▶ Consider the mathematical predicate $a = b \text{ mod } n$. We could formalise this operator as:

```
definition mod :: "int ⇒ int ⇒ int ⇒ bool"  
where "mod a b n ≡ ∃k. a = k * n + b"
```

- ▶ Isabelle performs **type inference**, allowing us to write:

$$\forall x y n. \text{mod } x y n \rightarrow \text{mod } y x n$$

instead of

$$\forall(x :: \text{int}) (y :: \text{int}) (n :: \text{int}). \text{mod } x y n \rightarrow \text{mod } y x n$$

FOL L-System Sequent Calculus Rules

$$\frac{\Gamma \vdash P[x_0/x]}{\Gamma \vdash \forall x. P} \text{ (allI)}$$

$$\frac{\Gamma, P[t/x] \vdash Q}{\Gamma, \forall x. P \vdash Q} \text{ (e allE t)}$$

$$\frac{\Gamma, \forall x. P, P[t/x] \vdash Q}{\Gamma, \forall x. P \vdash Q} \text{ (f spec t)}$$

$$\frac{\Gamma \vdash P[t/x]}{\Gamma \vdash \exists x. P} \text{ (r exI t)}$$

$$\frac{\Gamma, P[x_0/x] \vdash Q}{\Gamma, \exists x. P \vdash Q} \text{ (e exE)}$$

$$\frac{\Gamma, \forall x. \neg P \vdash \perp}{\Gamma \vdash \exists x. P} \text{ (exCIF)}$$

- ▶ Rule prefixes: e = erule, f = frule, r = rule
- ▶ x_0 is some variable **not** free in hypotheses or conclusion. Isabelle automatically picks fresh names (to ensure soundness!)
- ▶ When t suffix is used above (e.g., as in "e allE t"), then the term t can be explicitly specified in Isabelle method using a variant of the existing method. e.g., apply (erule_tac x="t" in allE).
- ▶ Rule exCIF is a variation on the standard Isabelle rule exCI introduced in the FOL.thy file on the course webpage. It does not exist as an explicit Isabelle inference rule but can be derived (see FOL.thy).

Example II as a FOL Sequent Proof

$$\frac{\frac{\frac{P(y) \vdash P(y)}{} \quad \frac{P(y), Q(y) \vdash Q(y)}{} \text{ (e impE)}}{P(y) \rightarrow Q(y), P(\textcolor{red}{y}) \vdash Q(y)} \text{ (e allE y)}}{P(\textcolor{red}{y}) \rightarrow Q(\textcolor{red}{y}), \forall x. P(x) \vdash Q(y)} \text{ (e allE y)}$$
$$\frac{\frac{P(\textcolor{red}{y}) \rightarrow Q(\textcolor{red}{y}), \forall x. P(x) \vdash Q(y)}{\forall x. P(x) \rightarrow Q(x), \forall x. P(x) \vdash Q(\textcolor{red}{y})} \text{ (allI)}}{\forall x. P(x) \rightarrow Q(x), \forall x. P(x) \vdash \forall x. Q(x)} \text{ (allI)}$$

Summary

- ▶ Introduction to First-Order Logic (H&R 2.1-2.4)
 - ▶ Syntax and Semantics
 - ▶ Substitution
 - ▶ Natural Deduction rules for quantifiers
- ▶ Isabelle and First-Order Logic
 - ▶ Defining predicates
 - ▶ A brief look at types
 - ▶ Try FOL.thy on the course webpage in Isabelle.

Automated Reasoning

Lecture 6: Representation

Jacques Fleuriot

jdf@inf.ed.ac.uk

Recap

- ▶ Last time: First-Order Logic
- ▶ This time: Representing mathematical concepts

Representing Knowledge

So far, we have:

- ▶ Seen the primitive rules of (first-order) logic
- ▶ Reasoned about abstract P s, Q s, and R s

But we usually want to reason in some mathematical theory. For example: number theory, real analysis, automata theory, euclidean geometry, ...

How do we **represent** this theory so we can prove theorems about it?

- ▶ **Which logic do we use?** – Propositional, FOL, Temporal, Hoare Logic, HOL?
- ▶ Do we **axiomatise** our theory, or **define** it in terms of more primitive concepts?
- ▶ What style do we use? e.g. **functions vs. relations**

Further Issues

What are the important theorems in our theory?

- ▶ Which formalisation is most useful?
- ▶ Is it **easy to understand**?
- ▶ Is it **natural**?
- ▶ How easy is it to **reason** with?

Often a matter of taste, or experience, or tradition, or efficiency of implementation, or following the idioms of the people you are working with. **No single right way!**

Granularity of the representation

- ▶ What primitives do we need? Consider geometry:
 - ▶ Define lines in terms of points? (Tarski)
 - ▶ Or take points and lines as primitive? (Hilbert)
- ▶ Or computing; should we treat programs as:
 - ▶ State transition systems? (operational)
 - ▶ Functions mapping inputs to outputs? (\sim denotational)

Axioms vs. Definitions

Let's say we want to reason using the natural numbers $\{0, 1, 2, 3, \dots\}$

Axiomatise? Assume a collection of function symbols and *unproven axioms*. For instance, the Peano axioms:

$$\forall x. \neg(0 = S(x))$$

$$\forall x. x + 0 = x$$

$$\forall x. x + S(y) = S(x + y)$$

...

Define? If our logic has sets as a primitive (or are definable), then we can *define* the natural numbers via the von Neumann ordinals:

$$0 = \emptyset, 1 = \{\emptyset\}, 2 = \{\emptyset, \{\emptyset\}\}, \dots$$

Then we can *prove* the Peano axioms for this definition.

Axioms vs. Definitions

Axiomatisation:

- ▶ (+) Sometimes less work – finding a good definition, and (formally) working with it can be hard.
- ▶ (-) How do we know that our axiomatisation is adequate for our purposes, or is complete?
- ▶ (-) How do we know that our axiomatisation is consistent? Can we prove \perp from our axioms (and hence everything)?

Definition:

- ▶ (-) Can be a lot of work, sometimes needing some ingenuity.
- ▶ (+++) If the underlying logic is consistent, then we are *guaranteed* to be consistent (c.f., “Why should you believe Isabelle” from Lecture 4). We have **relative consistency**.

Axiomatisation, an example: Set Theory

Let's take FOL, a binary atomic predicate \in and the following axiom for every formula P with one free variable x :

$$\exists y. \forall x. x \in y \leftrightarrow P(x)$$

“For every predicate P there is a set y such that its members are exactly those that satisfy P ”

We can now define empty set, pairing, union, intersection...

Axiomatisation, an example: Set Theory

Let's take FOL, a binary atomic predicate \in and the following axiom for every formula P with one free variable x :

$$\exists y. \forall x. x \in y \leftrightarrow P(x)$$

“For every predicate P there is a set y such that its members are exactly those that satisfy P ”

We can now define empty set, pairing, union, intersection...

But it is **too powerful!** Let $P(x) \equiv \neg(x \in x)$. Then by the axiom there is a y such that:

$$y \in y \leftrightarrow y \notin y$$

This is Russell's paradox.

Axiomatisation, an example: Set Theory

Let's take FOL, a binary atomic predicate \in and the following axiom for every formula P with one free variable x :

$$\exists y. \forall x. x \in y \leftrightarrow P(x)$$

“For every predicate P there is a set y such that its members are exactly those that satisfy P ”

We can now define empty set, pairing, union, intersection...

But it is **too powerful!** Let $P(x) \equiv \neg(x \in x)$. Then by the axiom there is a y such that:

$$y \in y \leftrightarrow y \notin y$$

This is Russell's paradox.

Background: the axiom is called "unrestricted comprehension", it was replaced by:

$$\forall z. \exists y. \forall x. (x \in y \leftrightarrow (x \in z \wedge P(x)))$$

+ some other axioms to give ZF set theory.

Building up Definitions: Integers

Starting from the natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$, we can define:

- ▶ each integer $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ as an **equivalence class** of pairs of natural numbers under the relation
$$(a, b) \sim (c, d) \iff a + d = b + c;$$
- ▶ For example, -2 is represented by the equivalence class $[(0, 2)] = [(1, 3)] = [(100, 102)] = \dots$
- ▶ we define the sum and product of two integers as

$$[(a, b)] + [(c, d)] = [(a + c, b + d)] \\ [(a, b)] \times [(c, d)] = [(ac + bd, ad + bc)];$$

- ▶ we define the set of **negative** integers as the set
$$\{[(a, b)] \mid b > a\}.$$
- ▶ Exercise: show that the product of negative integers is non-negative.

Other Representation Examples

- ▶ The rationals \mathbb{Q} can be defined as pairs of integers. Reasoning about the rationals therefore reduces to reasoning about the integers.
- ▶ The reals \mathbb{R} can be defined as sets of rationals. Reasoning about the reals therefore reduces to reasoning about the rationals.
- ▶ The complex numbers \mathbb{C} can be defined as pairs of reals. Reasoning about the complex numbers therefore reduces to reasoning about the reals.
- ▶ In this way, we have **relative consistency**.
 - ▶ If the theory of natural numbers is consistent, so is the theory of complex numbers.

Functions or Predicates?

We can represent some property r holding between two objects x and y as:

- | | |
|--------------------------|------------|
| a function with equality | $r(x) = y$ |
| a predicate | $r(x, y)$ |

Is it better to use functions or predicates to represent properties?

It is not always clear which is best!

Functional Representation

For example, suppose we represent division of real numbers (/) by a function $\text{div} : \text{real} \times \text{real} \Rightarrow \text{real}$.

- ▶ We define $\text{div}(x, y)$ when $y \neq 0$ in normal way
- ▶ What about division-by-zero? What is the value of $\text{div}(x, 0)$?
- ▶ In first-order logic, functions are assumed to be **total**, so we have to pick a value!
- ▶ We could *choose* a convenient element: say 0. That way:

$$0 \leq x \rightarrow 0 \leq 1/x.$$

Predicate Representation

Q) Can we represent division of real numbers (/) by a relation
 $Div : \text{real} \times \text{real} \times \text{real} \Rightarrow \text{bool}$ such that $Div(x, y, z)$ is

- ▶ $x/y = z$ when $y \neq 0$, and
- ▶ \perp when $y = 0$?

A) Yes: $Div(x, y, z) \equiv x = y * z \wedge \forall w. x = y * w \rightarrow z = w$
That is, z is that *unique* value such that $x = y * z$.

But now formulas are more complicated.

$$x, y \neq 0 \rightarrow \frac{1}{((x/y)/x)} = y$$

becomes

$$Div(x, y, u) \wedge Div(u, x, v) \wedge Div(1, v, w) \wedge x, y \neq 0 \rightarrow w = y$$

Functional Representation

Can we represent the concept of *square roots* with a function

$$\sqrt{\cdot} : \text{real} \Rightarrow \text{real}$$

- ▶ All positive real numbers have *two* square roots, and yet a function maps points to *single* values.
- ▶ We can pick one of the values arbitrarily: say, the *positive (principal)* square root.
- ▶ Or we can have the function map every real to a *set*

$$\sqrt{\cdot} : \text{real} \Rightarrow \text{real set:}$$

$$\sqrt{x} \equiv \{y \mid x = y^2\}.$$

- ▶ But now we have two kinds of object: reals and sets of reals, and we cannot conveniently express:

$$(\sqrt{x})^2 = x$$

- ▶ Our representation of reals is no longer **self-contained**.

Predicate Representation

Q) Can we represent the concept of *square roots* with a relation
 $Sqrt : real \times real \Rightarrow bool$?

A) Yes. E.g. $Sqrt(x, y) \equiv x = y^2$.

Again drawback of formulas being more complicated

Functions, Predicates and Sets

We can translate back and forth. But too much translation makes a formalisation hard to use!

Any function $f : \alpha \rightarrow \beta$ can be represented as a relation $R : \alpha \times \beta \rightarrow \text{bool}$ or a set $S : (\alpha \times \beta) \text{set}$ by defining:

$$R(x, y) \equiv f(x) = y$$
$$S \equiv \{(x, y) \mid f(x) = y\}.$$

Any predicate P can be represented by a function f or a set S by defining:

$$f(x) \equiv \begin{cases} \text{True} & : P(x) \\ \text{False} & : \text{otherwise} \end{cases}$$
$$S \equiv \{x \mid P(x)\}.$$

Any set S can be represented by a function f or a predicate P by defining:

$$f(x) \equiv \begin{cases} \text{True} & : x \in S \\ \text{False} & : \text{otherwise} \end{cases}$$
$$P(x) \equiv x \in S$$

Set Theory, Functions, and HOL

In pure (without axioms) FOL, we **cannot** directly represent the statement:

there is a function that is larger on all arguments than the log function.

To formalise it, we would need to quantify over functions:

$$\exists f. \forall x. f(x) > \log x.$$

Likewise we cannot quantify over predicates.

Solutions in FOL:

- ▶ Represent all functions and predicates by **sets**, and quantify over these. This is the approach of first-order set theories such as *ZF*.
- ▶ Introduce sorts for predicates and functions. Not so elegant now having 2 kinds of each.

Summary

- ▶ This time:
 - ▶ Issues involved in representing mathematical theories
 - ▶ Axioms vs. Definitions
 - ▶ Functions vs. Predicates
 - ▶ Introduction to Higher-Order Logic
 - ▶ Reading: Bundy, Chapter 4 (contains further discussion of issues in representation, e.g. variadic functions).
- ▶ On the course web-page: some more exercises, asking you to “prove” `False` from the axioms of Naive Set Theory.

Automated Reasoning

Lecture 7: Introduction to Higher Order Logic in Isabelle

Jacques Fleuriot
jdf@inf.ed.ac.uk

Acknowledgement: Tobias Nipkow kindly provided some of the slides for this lecture

Recap

- ▶ Last time: Representing mathematical concepts
- ▶ This time: Higher-Order Logic (in Isabelle)

Higher-Order Logic (HOL)

In HOL, we represent sets and predicates by **functions**, often denoted by **lambda abstractions**.

Definition (Lambda Abstraction)

Lambda abstractions are **terms** that denote functions directly by the rules which define them, e.g. the square function is $\lambda x. x * x$.

This is a way of defining a function without giving it a name:

$$f(x) \equiv x * x \quad \text{vs} \quad f \equiv \lambda x. x * x$$

We can use lambda abstractions exactly as we use ordinary function symbols. E.g. $(\lambda x. x * x) 3 = 9$.

Higher-Order Functions

Using λ -notation, we can think about functions as individual objects.

E.g., we can define functions which map from and to other functions.

Example

The K -combinator maps some x to a function which sends any y to x .

$$\lambda x. \lambda y. x.$$

Example

The composition function maps two functions to their composition:

$$\lambda f. \lambda g. \lambda x. f(g x).$$

Representation of Logic in HOL I

- ▶ Types $bool$, ind (individuals) and $\alpha \Rightarrow \beta$ primitive. All others defined from these.
- ▶ Two primitive (families of) functions:

$$\begin{array}{ll} \text{equality} & (=_{\alpha}) : \alpha \Rightarrow \alpha \Rightarrow bool \\ \text{implication} & (\rightarrow) : bool \Rightarrow bool \Rightarrow bool \end{array}$$

All other functions defined using this, lambda abstraction and application.

- ▶ Distinction between formulas and terms is dispensed with: formulas are just terms of type $bool$.
- ▶ Predicates are represented by functions $\alpha \Rightarrow bool$. Sets are represented as predicates.

Representation of Logic in HOL II

- ▶ True is defined as:

$$\top \equiv (\lambda x.x) = (\lambda x.x)$$

- ▶ Universal quantification as function equality:

$$\forall x. \phi \equiv (\lambda x. \phi) = (\lambda x. \top).$$

This works for x of any type: bool , $\text{ind} \Rightarrow \text{bool}$, ...

- ▶ Therefore, we can **quantify over functions, predicates and sets.**
- ▶ Conjunction and disjunction are defined:

$$P \wedge Q \equiv \forall R.(P \rightarrow Q \rightarrow R) \rightarrow R$$

$$P \vee Q \equiv \forall R.(P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R$$

- ▶ Define natural numbers (\mathbb{N}), integers (\mathbb{Z}), rationals (\mathbb{Q}), reals (\mathbb{R}), complex numbers (\mathbb{C}), vector spaces, manifolds, ...

Isabelle/HOL

Higher-Order Logic is the underlying logic of Isabelle/HOL, the theorem prover we are using.

The axiomatisation is slightly different to the one described on the previous slides, and a bit more powerful (but we won't be delving into this).

We are interested in Isabelle/HOL from a functional programming and logic standpoint.

HOL = Functional Programming + Logic

HOL = Higher-Order Logic

HOL = Functional Programming + Logic

HOL = Higher-Order Logic

HOL = Functional Programming + Logic

HOL = Functional Programming + Logic

HOL = Higher-Order Logic

HOL = Functional Programming + Logic

HOL has

- ▶ datatypes
- ▶ recursive functions
- ▶ logical operators

HOL = Functional Programming + Logic

HOL = Higher-Order Logic

HOL = Functional Programming + Logic

HOL has

- ▶ datatypes
- ▶ recursive functions
- ▶ logical operators

HOL is a programming language!

HOL = Functional Programming + Logic

HOL = Higher-Order Logic

HOL = Functional Programming + Logic

HOL has

- ▶ datatypes
- ▶ recursive functions
- ▶ logical operators

HOL is a programming language!

Higher-order = functions are values, too!

Isabelle/HOL Types

Basic syntax:

$\tau ::=$

Isabelle/HOL Types

Basic syntax:

$$\tau ::= (\tau)$$

Isabelle/HOL Types

Basic syntax:

$$\begin{array}{lcl} \tau & ::= & (\tau) \\ & | & \textit{bool} \mid \textit{nat} \mid \textit{int} \mid \dots \quad \text{base types} \end{array}$$

Isabelle/HOL Types

Basic syntax:

$$\begin{array}{lcl} \tau & ::= & (\tau) \\ & | & \textit{bool} \mid \textit{nat} \mid \textit{int} \mid \dots \quad \text{base types} \\ & | & 'a \mid 'b \mid \dots \quad \text{type variables} \end{array}$$

Isabelle/HOL Types

Basic syntax:

$$\begin{array}{lcl} \tau & ::= & (\tau) \\ | & \textit{bool} & | \textit{nat} & | \textit{int} & | \dots & \text{base types} \\ | & 'a & | & 'b & | \dots & \text{type variables} \\ | & \tau \Rightarrow \tau & & & & \text{functions} \end{array}$$

Isabelle/HOL Types

Basic syntax:

$\tau ::= (\tau)$	
<i>bool</i> <i>nat</i> <i>int</i> ...	base types
' <i>a</i> ' <i>b</i> ...	type variables
$\tau \Rightarrow \tau$	functions
$\tau \times \tau$	pairs (ascii: *)

Isabelle/HOL Types

Basic syntax:

$\tau ::= (\tau)$	
<i>bool</i> <i>nat</i> <i>int</i> ...	base types
' <i>a</i> ' <i>b</i> ...	type variables
$\tau \Rightarrow \tau$	functions
$\tau \times \tau$	pairs (ascii: *)
τ <i>list</i>	lists

Isabelle/HOL Types

Basic syntax:

$\tau ::= (\tau)$	
<i>bool</i> <i>nat</i> <i>int</i> ...	base types
' <i>a</i> ' <i>b</i> ...	type variables
$\tau \Rightarrow \tau$	functions
$\tau \times \tau$	pairs (ascii: *)
τ <i>list</i>	lists
τ <i>set</i>	sets

Isabelle/HOL Types

Basic syntax:

$\tau ::= (\tau)$	
<i>bool</i> <i>nat</i> <i>int</i> ...	base types
' <i>a</i> ' <i>b</i> ...	type variables
$\tau \Rightarrow \tau$	functions
$\tau \times \tau$	pairs (ascii: *)
τ <i>list</i>	lists
τ <i>set</i>	sets
...	user-defined types

Isabelle/HOL Types

Basic syntax:

$\tau ::= (\tau)$	
<i>bool</i> <i>nat</i> <i>int</i> ...	base types
' <i>a</i> ' ' <i>b</i> ' ...	type variables
$\tau \Rightarrow \tau$	functions
$\tau \times \tau$	pairs (ascii: *)
τ <i>list</i>	lists
τ <i>set</i>	sets
...	user-defined types

Convention: $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \equiv \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$

Isabelle/HOL Types

Basic syntax:

$\tau ::= (\tau)$	
<i>bool</i> <i>nat</i> <i>int</i> ...	base types
' <i>a</i> ' ' <i>b</i> ' ...	type variables
$\tau \Rightarrow \tau$	functions
$\tau \times \tau$	pairs (ascii: *)
τ <i>list</i>	lists
τ <i>set</i>	sets
...	user-defined types

Convention: $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \equiv \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$

A formula is simply a term of type *bool*.

Isabelle/HOL Terms

Terms can be formed as follows:

Isabelle/HOL Terms

Terms can be formed as follows:

- ▶ *Function application:* ft

Isabelle/HOL Terms

Terms can be formed as follows:

- ▶ *Function application:* ft
is the call of function f with argument t .

Isabelle/HOL Terms

Terms can be formed as follows:

- ▶ *Function application: ft*

is the call of function f with argument t .

If f has more arguments: $ft_1 t_2 \dots$

Isabelle/HOL Terms

Terms can be formed as follows:

- ▶ *Function application: ft*

is the call of function f with argument t .

If f has more arguments: $ft_1 t_2 \dots$

Examples: $\sin \pi$, $\text{plus } x\,y$

Isabelle/HOL Terms

Terms can be formed as follows:

- ▶ *Function application:* ft

is the call of function f with argument t .

If f has more arguments: $ft_1 t_2 \dots$

Examples: $\sin \pi$, $\text{plus } x \ y$

- ▶ *Function abstraction:* $\lambda x. t$

Isabelle/HOL Terms

Terms can be formed as follows:

- ▶ *Function application:* ft

is the call of function f with argument t .

If f has more arguments: $ft_1 t_2 \dots$

Examples: $\sin \pi$, $\text{plus } x \ y$

- ▶ *Function abstraction:* $\lambda x. t$

is the function with parameter x and result t

Isabelle/HOL Terms

Terms can be formed as follows:

- ▶ *Function application:* ft

is the call of function f with argument t .

If f has more arguments: $ft_1 t_2 \dots$

Examples: $\sin \pi$, $\text{plus } x \ y$

- ▶ *Function abstraction:* $\lambda x. t$

is the function with parameter x and result t ,

i.e. " $x \mapsto t$ ".

Isabelle/HOL Terms

Terms can be formed as follows:

- ▶ *Function application:* ft

is the call of function f with argument t .

If f has more arguments: $ft_1 t_2 \dots$

Examples: $\sin \pi$, $\text{plus } x y$

- ▶ *Function abstraction:* $\lambda x. t$

is the function with parameter x and result t ,

i.e. " $x \mapsto t$ ".

Example: $\lambda x. \text{plus } x x$

Isabelle/HOL Terms

Basic syntax:

$t ::=$

Isabelle/HOL Terms

Basic syntax:

$$t ::= (t)$$

Isabelle/HOL Terms

Basic syntax:

$$\begin{array}{lcl} t & ::= & (t) \\ & | & a \end{array} \quad \text{constant or variable (identifier)}$$

Isabelle/HOL Terms

Basic syntax:

$t ::=$	(t)	
	a	constant or variable (identifier)
	$t t$	function application

Isabelle/HOL Terms

Basic syntax:

t	$::=$	(t)
		a constant or variable (identifier)
		$t\ t$ function application
		$\lambda x.\ t$ function abstraction

Isabelle/HOL Terms

Basic syntax:

t	$::=$	(t)
		a constant or variable (identifier)
		$t\ t$ function application
		$\lambda x.\ t$ function abstraction
		...

Isabelle/HOL Terms

Basic syntax:

t	$::=$	(t)	
		a	constant or variable (identifier)
		$t\ t$	function application
		$\lambda x.\ t$	function abstraction
		...	lots of syntactic sugar

Examples: $f(g\ x)\ y$

Isabelle/HOL Terms

Basic syntax:

$t ::=$	(t)	
	a	constant or variable (identifier)
	$t t$	function application
	$\lambda x. t$	function abstraction
	...	lots of syntactic sugar

Examples: $f(g\ x)\ y$
 $h\ (\lambda x.\ f(g\ x))$

Isabelle/HOL Terms

Basic syntax:

t	$::=$	(t)
		a constant or variable (identifier)
		$t\ t$ function application
		$\lambda x.\ t$ function abstraction
		...
		lots of syntactic sugar

Examples: $f(g\ x)\ y$
 $h\ (\lambda x.\ f(g\ x))$

Convention: $ft_1\ t_2\ t_3 \equiv ((ft_1)\ t_2)\ t_3$

Isabelle/HOL Terms

Basic syntax:

t	$::=$	(t)
		a constant or variable (identifier)
		$t\ t$ function application
		$\lambda x.\ t$ function abstraction
		...
		lots of syntactic sugar

Examples: $f(g\ x)\ y$
 $h\ (\lambda x.\ f(g\ x))$

Convention: $ft_1\ t_2\ t_3 \equiv ((ft_1)\ t_2)\ t_3$

This language of terms is known as the *λ -calculus*.

β -reduction

The computation rule of the λ -calculus is the replacement of formal by actual parameters:

$$(\lambda x. t) u = t[u/x]$$

β -reduction

The computation rule of the λ -calculus is the replacement of formal by actual parameters:

$$(\lambda x. t) u = t[u/x]$$

where $t[u/x]$ is “ t with u substituted for x ”.

β -reduction

The computation rule of the λ -calculus is the replacement of formal by actual parameters:

$$(\lambda x. t) u = t[u/x]$$

where $t[u/x]$ is “ t with u substituted for x ”.

Example: $(\lambda x. x + 5) 3 = 3 + 5$

β -reduction

The computation rule of the λ -calculus is the replacement of formal by actual parameters:

$$(\lambda x. t) u = t[u/x]$$

where $t[u/x]$ is “ t with u substituted for x ”.

Example: $(\lambda x. x + 5) 3 = 3 + 5$

- ▶ The step from $(\lambda x. t) u$ to $t[u/x]$ is called β -reduction.

β -reduction

The computation rule of the λ -calculus is the replacement of formal by actual parameters:

$$(\lambda x. t) u = t[u/x]$$

where $t[u/x]$ is “ t with u substituted for x ”.

Example: $(\lambda x. x + 5) 3 = 3 + 5$

- ▶ The step from $(\lambda x. t) u$ to $t[u/x]$ is called β -reduction.
- ▶ Isabelle performs β -reduction automatically.

Well-typed Terms

Terms must be well-typed

Well-typed Terms

Terms must be well-typed

(the argument of every function call must be of the right type)

Well-typed Terms

Terms must be well-typed

(the argument of every function call must be of the right type)

Notation:

$t :: \tau$ means “ t is a well-typed term of type τ ”.

Well-typed Terms

Terms must be well-typed

(the argument of every function call must be of the right type)

Notation:

$t :: \tau$ means “ t is a well-typed term of type τ ”.

$$\frac{t :: \tau_1 \Rightarrow \tau_2 \quad u :: \tau_1}{t\ u :: \tau_2}$$

Type inference

Isabelle automatically computes the type of each variable in a term.

Type inference

Isabelle automatically computes the type of each variable in a term. This is called *type inference*.

Type inference

Isabelle automatically computes the type of each variable in a term. This is called *type inference*.

In the presence of *overloaded* functions (functions with multiple types) this is not always possible.

Type inference

Isabelle automatically computes the type of each variable in a term. This is called *type inference*.

In the presence of *overloaded* functions (functions with multiple types) this is not always possible.

User can help with *type annotations* inside the term.

Examples $f(x:\text{nat})$
 $g(A:\text{real set})$

Currying

Process of transforming a function that takes multiple arguments into:

- ▶ one that takes just a single argument, and
- ▶ returns another *function* if any arguments are still needed.

Currying

Process of transforming a function that takes multiple arguments into:

- ▶ one that takes just a single argument, and
- ▶ returns another *function* if any arguments are still needed.

Typing:

- ▶ Curried: $f :: \tau_1 \Rightarrow \tau_2 \Rightarrow \tau$
- ▶ Tupled: $f' :: \tau_1 \times \tau_2 \Rightarrow \tau$

Currying

Process of transforming a function that takes multiple arguments into:

- ▶ one that takes just a single argument, and
- ▶ returns another *function* if any arguments are still needed.

Typing:

- ▶ Curried: $f :: \tau_1 \Rightarrow \tau_2 \Rightarrow \tau$
- ▶ Tupled: $f' :: \tau_1 \times \tau_2 \Rightarrow \tau$

Advantage:

Currying allows *partial application*

$$fa_1 :: \tau_2 \Rightarrow \tau \text{ where } a_1 :: \tau_1$$

Predefined syntactic sugar

- ▶ *Infix:* `+`, `-`, `*`, `#`, `@`, ...

Predefined syntactic sugar

- ▶ *Infix:* `+`, `-`, `*`, `#`, `@`, ...
- ▶ *Mixfix:* `if_then_else_`, `case_of`, ...

Predefined syntactic sugar

- ▶ Infix: `+`, `-`, `*`, `#`, `@`, ...
- ▶ Mixfix: `if_ then_ else_`, `case_ of`, ...

Prefix binds more strongly than infix:

$$! \quad fx + y \equiv (fx) + y \neq f(x + y) \quad !$$

Predefined syntactic sugar

- ▶ Infix: `+`, `-`, `*`, `#`, `@`, ...
- ▶ Mixfix: `if_ then_ else_`, `case_ of`, ...

Prefix binds more strongly than infix:

$$! \quad fx + y \equiv (fx) + y \neq f(x + y) \quad !$$

Enclose `if` and `case` in parentheses:

$$! \quad (if_ then_ else_) \quad !$$

Example: Type *bool*

```
datatype bool = True | False
```

Example: Type *bool*

```
datatype bool = True | False
```

Predefined functions:

$\wedge, \vee, \rightarrow, \dots :: \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool}$

Example: Type *bool*

```
datatype bool = True | False
```

Predefined functions:

$\wedge, \vee, \rightarrow, \dots :: \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool}$

A *formula* is a term of type *bool*

Example: Type *bool*

```
datatype bool = True | False
```

Predefined functions:

$\wedge, \vee, \rightarrow, \dots :: bool \Rightarrow bool \Rightarrow bool$

A *formula* is a term of type *bool*

if-and-only-if: =

Example: Type *nat*

```
datatype nat = 0 | Suc nat
```

Example: Type *nat*

```
datatype nat = 0 | Suc nat
```

Values of type *nat*: 0, Suc 0, Suc(Suc 0), ...

Example: Type *nat*

datatype *nat* = 0 | Suc *nat*

Values of type *nat*: 0, Suc 0, Suc(Suc 0), ...

Predefined functions: +, *, ... :: *nat* \Rightarrow *nat* \Rightarrow *nat*

Example: Type *nat*

```
datatype nat = 0 | Suc nat
```

Values of type *nat*: 0, Suc 0, Suc(Suc 0), ...

Predefined functions: +, *, ... :: nat \Rightarrow nat \Rightarrow nat

! Numbers and arithmetic operations are overloaded:

0, 1, 2, ... :: 'a, + :: 'a \Rightarrow 'a \Rightarrow 'a

Example: Type *nat*

```
datatype nat = 0 | Suc nat
```

Values of type *nat*: 0, Suc 0, Suc(Suc 0), ...

Predefined functions: +, *, ... :: nat \Rightarrow nat \Rightarrow nat

! Numbers and arithmetic operations are overloaded:

0, 1, 2, ... :: 'a, + :: 'a \Rightarrow 'a \Rightarrow 'a

You need type annotations: 1 :: nat, x + (y::nat)

Example: Type *nat*

```
datatype nat = 0 | Suc nat
```

Values of type *nat*: 0, Suc 0, Suc(Suc 0), ...

Predefined functions: +, *, ... :: nat \Rightarrow nat \Rightarrow nat

! Numbers and arithmetic operations are overloaded:

0, 1, 2, ... :: 'a, + :: 'a \Rightarrow 'a \Rightarrow 'a

You need type annotations: 1 :: nat, x + (y::nat)
unless the context is unambiguous: Suc z

More on Isabelle/HOL

If you are really keen, look at the chapter “Higher-Order Logic” in the “logics” document in the Isabelle documentation.

Or the file `src/HOL/HOL.thy` in the Isabelle installation.

Exercise (only if you are interested!): why can't Russell's paradox happen in HOL?

Summary

- ▶ General introduction to Higher-Order Logic
- ▶ Types and Terms in Isabelle/HOL

Automated Reasoning

Lecture 8: Representation II Locales in Isabelle/HOL

Jacques Fleuriot
`jdf@inf.ed.ac.uk`

Axiomatic Extensions Considered Harmful

As we saw already, **definitional** extension is favoured over **axiomatic** extension in Isabelle/HOL.

Axiomatic Extensions Considered Harmful

As we saw already, **definitional** extension is favoured over **axiomatic** extension in Isabelle/HOL.

- ▶ Axiomatization can introduce an inconsistency.

Axiomatic Extensions Considered Harmful

As we saw already, **definitional** extension is favoured over **axiomatic** extension in Isabelle/HOL.

- ▶ Axiomatization can introduce an inconsistency.
- ▶ Example: After declaring the existence of a new type *SET* in Isabelle, it is possible to add a new axiom:

axiomatization

Member :: $SET \Rightarrow SET \Rightarrow bool$

where

comprehension : $\exists y. \forall x. Member\,x\,y \longleftrightarrow P\,x$

Axiomatic Extensions Considered Harmful

As we saw already, **definitional** extension is favoured over **axiomatic** extension in Isabelle/HOL.

- ▶ Axiomatization can introduce an inconsistency.
- ▶ Example: After declaring the existence of a new type *SET* in Isabelle, it is possible to add a new axiom:

axiomatization

Member :: $SET \Rightarrow SET \Rightarrow bool$

where

comprehension : $\exists y. \forall x. Member\,x\,y \longleftrightarrow P\,x$

which enables a "proof" of the paradoxical lemma:

`lemma member_if_not_member : $\exists y. Member\,y\,y \longleftrightarrow \neg Member\,y\,y$`

from which *False* can be derived.

Axiomatic Extensions Considered Harmful

As we saw already, **definitional** extension is favoured over **axiomatic** extension in Isabelle/HOL.

- ▶ Axiomatization can introduce an inconsistency.
- ▶ Example: After declaring the existence of a new type *SET* in Isabelle, it is possible to add a new axiom:

axiomatization

Member :: $SET \Rightarrow SET \Rightarrow bool$

where

comprehension : $\exists y. \forall x. Member\,x\,y \longleftrightarrow P\,x$

which enables a "proof" of the paradoxical lemma:

```
lemma member_if_not_member :  $\exists y. Member\,y\,y \longleftrightarrow \neg Member\,y\,y$ 
```

from which *False* can be derived.

- ▶ Yet, axiomatic reasoning is part of mathematics. We want to be able to carry it out safely in Isabelle.

Local axiomatic reasoning in Isabelle/HOL

Fortunately, we can reason from axioms *locally* in a sound way. For example, to prove results about groups, rings or vector spaces.

Local axiomatic reasoning in Isabelle/HOL

Fortunately, we can reason from axioms *locally* in a sound way. For example, to prove results about groups, rings or vector spaces.

We later *instantiate* the axioms with actual groups, rings, vector spaces.

Local axiomatic reasoning in Isabelle/HOL

Fortunately, we can reason from axioms *locally* in a sound way. For example, to prove results about groups, rings or vector spaces.

We later *instantiate* the axioms with actual groups, rings, vector spaces.

Isabelle provides a facility for doing this called **locales**.

```
locale group =
  fixes mult :: 'a ⇒ 'a ⇒ 'a  and  unit :: 'a
  assumes left_unit      : mult unit x = x
    and associativity : mult x (mult y z) = mult (mult x y) z
    and left_inverse   : ∃y. mult y x = unit
```

Isabelle Locales

- ▶ Named, encapsulated contexts, highly suitable for formalising abstract mathematics.

Isabelle Locales

- ▶ Named, encapsulated contexts, highly suitable for formalising abstract mathematics.
 - ▶ Context as a formula:

$$\bigwedge \overbrace{x_1 \dots x_n}^{parameters}. \overbrace{[A_1; \dots A_m]}^{assumptions} \Rightarrow \overbrace{C}^{theorem}$$

Isabelle Locales

- ▶ Named, encapsulated contexts, highly suitable for formalising abstract mathematics.
 - ▶ Context as a formula:

$$\bigwedge \overbrace{x_1 \dots x_n}^{parameters}. \llbracket \overbrace{A_1; \dots A_m}^{assumptions} \rrbracket \Rightarrow \overbrace{C}^{theorem}$$

- ▶ Locales usually have

Isabelle Locales

- ▶ Named, encapsulated contexts, highly suitable for formalising abstract mathematics.
 - ▶ Context as a formula:

$$\bigwedge \overbrace{x_1 \dots x_n}^{parameters}. \llbracket \overbrace{A_1; \dots A_m}^{assumptions} \rrbracket \Rightarrow \overbrace{C}^{theorem}$$

- ▶ Locales usually have
 - ▶ parameters, declared using `fixes`

Isabelle Locales

- ▶ Named, encapsulated contexts, highly suitable for formalising abstract mathematics.
 - ▶ Context as a formula:

$$\bigwedge \overbrace{x_1 \dots x_n}^{parameters}. \llbracket \overbrace{A_1; \dots A_m}^{assumptions} \rrbracket \Rightarrow \overbrace{C}^{theorem}$$

- ▶ Locales usually have
 - ▶ parameters, declared using `fixes`
 - ▶ assumptions, declared using `assumes`

Isabelle Locales

- ▶ Named, encapsulated contexts, highly suitable for formalising abstract mathematics.
 - ▶ Context as a formula:

$$\bigwedge \overbrace{x_1 \dots x_n}^{parameters}. \llbracket \overbrace{A_1; \dots A_m}^{assumptions} \rrbracket \Rightarrow \overbrace{C}^{theorem}$$

- ▶ Locales usually have
 - ▶ parameters, declared using `fixes`
 - ▶ assumptions, declared using `assumes`
- ▶ Inside a locale, definitions can be made and theorems proven based on the parameters and assumptions.

Isabelle Locales

- ▶ Named, encapsulated contexts, highly suitable for formalising abstract mathematics.
 - ▶ Context as a formula:

$$\bigwedge \overbrace{x_1 \dots x_n}^{parameters}. \llbracket \overbrace{A_1; \dots A_m}^{assumptions} \rrbracket \Rightarrow \overbrace{C}^{theorem}$$

- ▶ Locales usually have
 - ▶ parameters, declared using `fixes`
 - ▶ assumptions, declared using `assumes`
- ▶ Inside a locale, definitions can be made and theorems proven based on the parameters and assumptions.
- ▶ A locale can import/extend other locales.

Locale Example: Finite Graphs

```
locale finitegraph =
  fixes edges :: "('a × 'a) set" and vertices :: "'a set"
  assumes finite_vertex_set : finite vertices
    and is_graph : (u, v) ∈ edges ⇒ u ∈ vertices ∧ v ∈ vertices
begin
  inductive walk :: "'a list ⇒ bool" where
    Nil : walk []
    | Singleton : v ∈ vertices ⇒ walk [v]
    | Cons : (v, w) ∈ edges ⇒ walk (w#vs) ⇒ walk (v#w#vs)
  lemma walk_edge : (v, w) ∈ edges ⇒ walk [v, w]
  ...
end
```

- ▶ # is the list cons operator in Isabelle.

Locale Example: Finite Graphs

```
locale finitegraph =
  fixes edges :: "('a × 'a) set  and  vertices :: 'a set
  assumes finite_vertex_set : finite vertices
        and is_graph          : (u, v) ∈ edges ⇒ u ∈ vertices ∧ v ∈ vertices

begin

  inductive walk :: 'a list ⇒ bool  where
    Nil           : walk []
    | Singleton   : v ∈ vertices ⇒ walk [v]
    | Cons         : (v, w) ∈ edges ⇒ walk (w#vs) ⇒ walk (v#w#vs)

  lemma walk_edge : (v, w) ∈ edges ⇒ walk [v, w]
  ...

end
```

- ▶ # is the list cons operator in Isabelle.
- ▶ The definition of this locale can be inspected by typing `thm finitegraph_def` in Isabelle:

```
finitegraph ?edges ?vertices ≡
finite ?vertices ∧
(∀uv.(u, v) ∈ ?edges → u ∈ ?vertices ∧ v ∈ ?vertices)
```

Adding Theorems to a Locale

Aside from proving a lemma within the locale definition, e.g. `walk_edge` on the previous slide, we can also state lemmas that are "in" some locale:

```
lemma (in group) associativity_bw :  
  "mult (mult x y) z = mult x (mult y z)"  
apply (subst associativity)  
apply (rule refl)  
done
```

Adding Theorems to a Locale

Aside from proving a lemma within the locale definition, e.g. `walk_edge` on the previous slide, we can also state lemmas that are "in" some locale:

```
lemma (in group) associativity_bw :  
  "mult (mult x y) z = mult x (mult y z)"  
apply (subst associativity)  
apply (rule refl)  
done
```

Alternatively, we can enter a locale at the theory level using the `context` keyword and formalize new definitions and theorems:

```
context group  
begin  
  lemma associativity_bw :  
    "mult (mult x y) z = mult x (mult y z)"  
  apply (subst associativity)  
  apply (rule refl)  
  done  
end
```

Locale Extension

- ▶ New locales can extend existing ones by adding more parameter, assumptions and definitions. This is also known as an *import*.

Locale Extension

- ▶ New locales can extend existing ones by adding more parameter, assumptions and definitions. This is also known as an *import*.
- ▶ The context of the imported locale i.e. all its assumptions, theorems etc. are available in the extended locale.

Locale Extension

- ▶ New locales can extend existing ones by adding more parameter, assumptions and definitions. This is also known as an *import*.
- ▶ The context of the imported locale i.e. all its assumptions, theorems etc. are available in the extended locale.

```
locale weighted_finigraph = finigraph +
  fixes weight :: ('a × 'a) ⇒ nat
  assumes edges_weighted : ∀e ∈ edges. ∃w. weight e = w
```

Locale Extension

- ▶ New locales can extend existing ones by adding more parameter, assumptions and definitions. This is also known as an *import*.
- ▶ The context of the imported locale i.e. all its assumptions, theorems etc. are available in the extended locale.

```
locale weighted_finigraph = finigraph +
  fixes weight :: ('a × 'a) ⇒ nat
  assumes edges_weighted : ∀e ∈ edges. ∃w. weight e = w
```

Viewed in terms of the imported *finigraph* locale (and the weighted edges axiom), we have:

```
weighted_finigraph ?edges ?vertices ?weight ≡
finigraph ?edges ?vertices ∧ (∀e ∈ ?edges. ∃w. ?weight e = w)
```

Instantiating Locales

- ▶ *Concrete* examples may be proven to be instances of a locale.

Instantiating Locales

- ▶ *Concrete* examples may be proven to be instances of a locale.
- ▶ interpretation `interpretation_name : locale_name args` generates the proof obligation that the locale predicate holds of the `args`.

Instantiating Locales

- ▶ *Concrete* examples may be proven to be instances of a locale.
- ▶ interpretation `interpretation_name : locale_name args` generates the proof obligation that the locale predicate holds of the `args`.
- ▶ Example: A graph with one vertex and single edge from that vertex to itself is a concrete instance of the locale `finite_graph`.

Instantiating Locales

- ▶ *Concrete* examples may be proven to be instances of a locale.
- ▶ interpretation `interpretation_name : locale_name args` generates the proof obligation that the locale predicate holds of the `args`.
- ▶ Example: A graph with one vertex and single edge from that vertex to itself is a concrete instance of the locale *finite_graph*.

```
interpretation singleton_finitegraph : finitegraph "{(1, 1)}" "{1}"
proof
  show "finite {1}" by simp
  next fix u v
  assume "(u, v) ∈ {(1, 1)}" then show "u ∈ {1} ∧ v ∈ {1}" by blast
qed
```

Instantiating Locales

- ▶ *Concrete* examples may be proven to be instances of a locale.
- ▶ interpretation `interpretation_name : locale_name args` generates the proof obligation that the locale predicate holds of the `args`.
- ▶ Example: A graph with one vertex and single edge from that vertex to itself is a concrete instance of the locale *finite_graph*.

```
interpretation singleton_finitegraph : finitegraph "{(1, 1)}" "{1}"
proof
  show "finite {1}" by simp
  next fix u v
  assume "(u, v) ∈ {(1, 1)}" then show "u ∈ {1} ∧ v ∈ {1}" by blast
qed
```

- ▶ We can prove that *singleton_finitegraph* is an instance of a finite weighted graph locale by providing a weight function as an additional argument:

Instantiating Locales

- ▶ *Concrete* examples may be proven to be instances of a locale.
- ▶ interpretation `interpretation_name : locale_name args` generates the proof obligation that the locale predicate holds of the `args`.
- ▶ Example: A graph with one vertex and single edge from that vertex to itself is a concrete instance of the locale *finite_graph*.

```
interpretation singleton_finitegraph : finitegraph "{(1, 1)}" "{1}"
proof
  show "finite {1}" by simp
  next fix u v
  assume "(u, v) ∈ {(1, 1)}" then show "u ∈ {1} ∧ v ∈ {1}" by blast
qed
```

- ▶ We can prove that *singleton_finitegraph* is an instance of a finite weighted graph locale by providing a weight function as an additional argument:

```
interpretation
  singleton_finitegraph : weighted_finitegraph "{(1, 1)}" "{1}" "λ(u, v). 1"
  by (unfold_locales) simp
```

Summary

- ▶ Axiomatization at the Isabelle theory level (i.e. as an extension of Isabelle/HOL) is not favoured as it can be unsound (see the additional exercise on the AR web page).
- ▶ Locales provide a sound way of reasoning locally about axiomatic theories.
- ▶ This was an introduction to locale declarations, extensions and interpretations.
 - ▶ There are many other features involving representation and reasoning using locales in Isabelle.
 - ▶ Reading: Tutorial to Locales and Locale Interpretation (on the AR web page).

Automated Reasoning

Lecture 10: Isar – A Language for Structured Proofs

Jacques Fleuriot

jdf@inf.ed.ac.uk

Acknowledgement: Tobias Nipkow kindly provided the slides for this lecture

Apply scripts

- ▶ unreadable
- ▶ hard to maintain
- ▶ do not scale

No structure!

Apply scripts versus Isar proofs

Apply script = assembly language program

Isar proof = structured program with comments

But: **apply** still useful for proof exploration

A typical Isar proof

proof

assume $formula_0$

have $formula_1$ **by** *simp*

⋮

have $formula_n$ **by** *blast*

show $formula_{n+1}$ **by** ...

qed

proves $formula_0 \Rightarrow formula_{n+1}$

Isar core syntax

proof = **proof** [method] step* **qed**
| **by** method

method = (*simp* ...) | (*blast* ...) | (*induction* ...) | ...

step = **fix** variables (\wedge)
| **assume** prop (\Rightarrow)
| [**from** fact⁺] (**have** | **show**) prop proof

prop = [name:] "formula"

fact = name | ...

Example: Cantor's theorem

lemma $\neg \text{surj}(f :: 'a \Rightarrow 'a \text{ set})$

proof default proof: assume surj , show False

assume $a : \text{surj } f$

from a **have** $b : \forall A. \exists a. A = fa$

by (*simp add: surj_def*)

from b **have** $c : \exists a. \{x. x \notin fx\} = fa$

by *blast*

from c **show** False

by *blast*

qed

Abbreviations

- this* = the previous proposition proved or assumed
- then* = **from this**
- thus* = **then show**
- hence* = **then have**

using and with

(**have|show**) prop **using** facts
=

from facts (**have|show**) prop

with facts
=

from facts *this*

Structured lemma statement

lemma

fixes $f :: "a \Rightarrow 'a\ set"$

assumes $s: \text{surj } f$

shows "False"

proof - no automatic proof step

have " $\exists a. \{x. x \notin fx\} = fa$ " **using** s

by(*auto simp: surj_def*)

thus "False" **by** *blast*

qed

Proves $\text{surj } f \Rightarrow \text{False}$

but $\text{surj } f$ becomes local fact s in proof.

The essence of structured proofs

Assumptions and intermediate facts
can be named and referred to explicitly and selectively

Structured lemma statements

fixes $x :: \tau_1$ **and** $y :: \tau_2$...

assumes $a: P$ **and** $b: Q$...

shows R

- ▶ **fixes** and **assumes** sections optional
- ▶ **shows** optional if no **fixes** and **assumes**

Proof patterns: Case distinction

show "R"	have "P ∨ Q" ...
proof cases	then show "R"
assume "P"	proof
:	assume "P"
show "R" ...	:
next	show "R" ...
assume " $\neg P$ "	next
:	assume "Q"
show "R" ...	:
qed	show "R" ...
	qed

Proof patterns: Contradiction

```
show " $\neg P$ "  
proof  
  assume " $P$ "  
  :  
  show "False" ...  
qed
```

```
show " $P$ "  
proof (rule ccontr)  
  assume " $\neg P$ "  
  :  
  show "False" ...  
qed
```

Proof patterns: \longleftrightarrow

```
show "P  $\longleftrightarrow$  Q"
proof
  assume "P"
  :
  show "Q" ...
next
  assume "Q"
  :
  show "P" ...
qed
```

Proof patterns: \forall and \exists introduction

show " $\forall x. P(x)$ "

proof

fix x local fixed variable

show " $P(x)$ " ...

qed

show " $\exists x. P(x)$ "

proof

:

show " $P(\text{witness})$ " ...

qed

Proof patterns: \exists elimination: obtain

have $\exists x. P(x)$

then obtain x **where** $p: P(x)$ **by** *blast*

: x fixed local variable

Works for one or more x

obtain example

lemma $\neg \text{surj}(f :: 'a \Rightarrow 'a \text{ set})$

proof

assume $\text{surj } f$

hence $\exists a. \{x. x \notin fx\} = fa$ **by** (auto simp: surj_def)

then obtain a **where** $\{x. x \notin fx\} = fa$ **by** blast

hence $a \notin fa \longleftrightarrow a \in fa$ **by** blast

thus False **by** blast

qed

Proof patterns: Set equality and subset

```
show "A = B"  
proof  
  show "A ⊆ B" ...  
next  
  show "B ⊆ A" ...  
qed
```

```
show "A ⊆ B"  
proof  
  fix x  
  assume "x ∈ A"  
  :  
  show "x ∈ B" ...  
qed
```

Example: pattern matching

```
show formula1  $\longleftrightarrow$  formula2 (is ?L  $\longleftrightarrow$  ?R)
```

```
proof
```

```
assume ?L
```

```
:
```

```
show ?R ...
```

```
next
```

```
assume ?R
```

```
:
```

```
show ?L ...
```

```
qed
```

?thesis

show formula (is ?thesis)

proof -

:

show ?thesis ...

qed

Every show implicitly defines ?thesis

let

Introducing local abbreviations in proofs:

let *?t* = "some-big-term "

:

have "... *?t* ..."

Quoting facts by value

By name:

```
have x0: "x > 0" ...
```

```
:
```

```
from x0 ...
```

By value:

```
have "x > 0" ...
```

```
:
```

```
from 'x>0' ...
```



back quotes

Example

lemma

$(\exists ys \ zs. xs = ys @ zs \wedge \text{length } ys = \text{length } zs) \vee$
 $(\exists ys \ zs. xs = ys @ zs \wedge \text{length } ys = \text{length } zs + 1)$ "

proof ???

When automation fails

Split proof up into smaller steps.

Or explore by **apply**:

have ... using ...

apply - to make incoming facts
part of proof state

apply auto or whatever

apply ...

At the end:

- ▶ **done**
- ▶ Better: convert to structured proof

moreover—ultimately

have " P_1 " ...

moreover

have " P_2 " ...

moreover

:

moreover

have " P_n " ...

ultimately

have " P " ...

\approx

have lab_1 : " P_1 " ...

have lab_2 : " P_2 " ...

:

have lab_n : " P_n " ...

from lab_1 lab_2 ...

have " P " ...

With names

Raw proof blocks

```
{ fix x1 ... xn
  assume A1 ... Am
  :
  have B
}
```

proves $\llbracket A_1; \dots ; A_m \rrbracket \Rightarrow B$
where all x_i have been replaced by $\textcolor{blue}{?x}_i$.

Proof state and Isar text

In general: **proof** *method*

Applies *method* and generates subgoal(s):

$$\lambda x_1 \dots x_n [[A_1; \dots ; A_m]] \Rightarrow B$$

How to prove each subgoal:

fix $x_1 \dots x_n$

assume $A_1 \dots A_m$

\vdots

show B

Separated by **next**

Datatype case analysis

datatype $t = C_1 \vec{\tau} \mid \dots$

```
proof (cases "term")
  case ( $C_1 x_1 \dots x_k$ )
    ...
     $x_j$  ...
  next
  :
  qed
```

where **case** ($C_i x_1 \dots x_k$) \equiv

fix $x_1 \dots x_k$
assume $\underbrace{C_i:}_{\text{label}} \underbrace{\text{term} = (C_i x_1 \dots x_k)}_{\text{formula}}$

Structural induction for *nat*

```
show P(n)
proof (induction n)
  case 0           ≡ let ?case = P(0)
  ...
  show ?case
next
  case (Suc n)    ≡ fix n assume Suc: P(n)
  ...
  let ?case = P(Suc n)
  ...
  show ?case
qed
```

Structural induction with \Rightarrow

```
show A(n)  $\Rightarrow$  P(n)
proof (induction n)
  case 0           ≡  assume 0: A(0)
  ...
  let ?case = P(0)

  show ?case
next
  case (Suc n)    ≡  fix n
  ...
  assume Suc: A(n)  $\Rightarrow$  P(n)
            A(Suc n)
  ...
  let ?case = P(Suc n)

  show ?case
qed
```

Named assumptions

In a proof of

$$A_1 \Rightarrow \dots \Rightarrow A_n \Rightarrow B$$

by structural induction:

In the context of

case *C*

we have

C.IH the induction hypotheses

C.prems the premises A_i

C $C.IH + C.prems$

A remark on style

- ▶ **case** ($Suc\ n$) ...**show** $?case$
is easy to write and maintain
- ▶ **fix** n **assume** $formula$...**show** $formula'$
is easier to read:
 - ▶ all information is shown locally
 - ▶ no contextual references (e.g. $?case$)

Rule induction

```
inductive I ::  $\tau \Rightarrow \sigma \Rightarrow \text{bool}$ 
where
rule1: ...
:
rulen: ...
```

```
show  $I x y \Rightarrow P x y$ 
proof (induction rule: I.induct)
  case rule1
  ...
  show ?case
next
:
next
  case rulen
  ...
  show ?case
qed
```

Fixing your own variable names

case ($\text{rule}_i\ x_1 \dots x_k$)

Renames the first k variables in rule_i (from left to right) to $x_1 \dots x_k$.

Named assumptions

In a proof of

$$I \dots \Rightarrow A_1 \Rightarrow \dots \Rightarrow A_n \Rightarrow B$$

by rule induction on $I \dots$:

In the context of

case R

we have

$R.IH$ the induction hypotheses

$R.hyps$ the assumptions of rule R

$R.prem$ s the premises A_i

$$R \quad R.IH + R.hyps + R.prem$$

Rule inversion

```
inductive ev :: "nat ⇒ bool" where
  ev0: "ev 0" |
  evSS: "ev n ⇒ ev(Suc(Suc n))"
```

What can we deduce from $ev\ n$?

That it was proved by either $ev0$ or $evSS$!

$$ev\ n \Rightarrow n = 0 \vee (\exists k. n = Suc(Suc k) \wedge ev\ k)$$

Rule inversion = case distinction over rules

Rule inversion template

```
from `ev n` have "P"
proof cases
  case ev0                               n = 0
  ...
  show ?thesis ...
next
  case (evSS k)                         n = Suc (Suc k), ev k
  ...
  show ?thesis ...
qed
```

Impossible cases disappear automatically

Summary

- ▶ Introduction to Isar and to some common proof patterns e.g. case distinction, contradiction, etc.
- ▶ Structured proofs are becoming the norm for Isabelle as they are more readable and easier to maintain.
- ▶ Mastering structured proof takes practice and it is usually better to have a clear proof plan beforehand.
- ▶ Useful resource: Isar quick reference manual (see AR web page).
- ▶ Reading: N&K (Concrete Semantics), Chapter 5.

Automated Reasoning

Lecture 11: Unification

Jacques Fleuriot
`jdf@inf.ed.ac.uk`

Recap

- ▶ This lecture:
 - ▶ Solving equations by Unification
 - ▶ Matching and Unification algorithms
 - ▶ Building-in axioms: E -Unification

Motivation

Unification: finding a common instance of two terms

Informally: we want to make two terms **identical** by finding the **most general substitution** of terms for variables.

Why?

- ▶ Applying rules in Isabelle: working out what $?P$, $?Q$, $?x$ are
- ▶ Heavily used in automated first-order theorem proving to postpone decisions during proof search: PROLOG, tableau provers, resolution provers
- ▶ Also used in most type inference algorithms (Haskell, OCaml, SML, Scala, ...)

A First Look at Unification

Unification: finding a common instance of two terms

Informally: we want to make two terms **identical** by finding the **most general substitution** of terms for variables.

Example

Can we make these pairs of terms equal by finding a common instance (assuming X, Y are variables and a, b are constants)?

$f(X, b)$ and $f(a, Y)$	Yes: $[a/X, b/Y]$	instance: $f(a, b)$
$f(X, X)$ and $f(a, b)$	No	
$f(X, X)$ and $f(Y, g(Y))$	No	

Only (meta-)variables (X, Y, Z, \dots) can be replaced by other terms.

Matching

Problem

Given **pattern** and **target** find a **substitution** such that:

$$\text{pattern}[\text{substitution}] \equiv \text{target}$$

where \equiv means that the terms are identical.

Example

$$(s(X) + Y)[0/X, s(0)/Y] \equiv (s(0) + s(0))$$

How we do find an adequate substitution?

We view matching as equation solving.

Matching (continued)

Discover a substitution by decomposing the equation to be solved along the term trees:

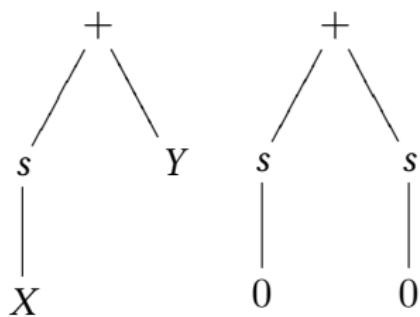
$$(s(X) + Y) \equiv (s(0) + s(0))$$



$$(s(X) \equiv s(0)) \wedge (Y \equiv s(0))$$



$$(X \equiv 0) \wedge (Y \equiv s(0))$$



Some Abbreviations

Term	Meaning
\vec{t}	$t_1, \dots, t_n \quad (t \geq 1)$
$\bigwedge_i t_i$	$t_1 \wedge \dots \wedge t_n$
$\text{vars}(t)$	the set of free variables in t
Vars	the set of (all) free variables

$$\text{vars}(f(X, Y, g(a, Z, X))) = \{X, Y, Z\}$$

$$\text{vars}(f(a, b, c)) = \{\}$$

Matching as Equation Solving

Start with the *pattern* and *target* standardised apart:

$$\text{vars}(\textit{pattern}) \cap \text{vars}(\textit{target}) = \{\}$$

Goal is to solve for $\text{vars}(\textit{pattern})$ in equation $\textit{pattern} \equiv \textit{target}$.

Strategy is to use transformation rules:

$$\begin{array}{c} \textit{pattern} \equiv \textit{target} \\ \downarrow \\ \vdots \\ \downarrow \\ X_1 \equiv t_1 \wedge \dots \wedge X_n \equiv t_n \end{array}$$

Resulting substitution is $[t_1/X_1, \dots, t_n/X_n]$.

Transformations end in failure if no match is possible.

Transformation Rules for Matching (Examples)

$$s(X) + Y \equiv s(0) + s(0)$$

Decompose



$$s(X) \equiv s(0) \wedge Y \equiv s(0)$$

$$s(X) + y \equiv s(0)$$

Conflict



fail

Cannot match: $s \not\equiv +$

$$(X + Y \equiv s(0) + 0) \wedge (Y \equiv 0)$$

Eliminate



$$(X + 0 \equiv s(0) + 0) \wedge (Y \equiv 0)$$

$$X \equiv 0 \wedge (s(0) + 0 \equiv s(0) + 0)$$

Delete



$$X \equiv 0$$

Transformation Rules for Matching

Assumptions: s and t are arbitrary terms and are standardised apart.

Name	Before	After	Condition
Decompose	$P \wedge f(\vec{s}) \equiv f(\vec{t})$	$P \wedge \bigwedge_i s_i \equiv t_i$	
Conflict	$P \wedge f(\vec{s}) \equiv g(\vec{t})$	fail	$f \neq g$
Eliminate	$P \wedge X \equiv t$	$P[t/X] \wedge X \equiv t$	$X \in \text{vars}(P)$
Delete	$P \wedge t \equiv t$	P	

Algorithm terminates when no further rules apply and fail has not occurred.

The algorithm terminates with a match iff there is one.

The algorithm may terminate without a match: e.g., $X \equiv a \wedge b \equiv Y$

Unification

Unification is two-way matching (there is no distinction between pattern and target).

$$\text{term}_1[\text{substitution}] \equiv \text{term}_2[\text{substitution}]$$

Example

What substitution makes $(s(X) + s(0))$ and $(s(0) + Y)$ identical?

$$\theta = [0/X, s(0)/Y]$$

We need to add **extra** rules to the matching algorithm:

$$\begin{array}{c} (s(X) + s(0)) \equiv (s(0) + Y) \\ \downarrow \qquad \qquad \qquad \text{Decompose} \\ s(X) \equiv s(0) \wedge s(0) \equiv Y \\ \downarrow \qquad \qquad \qquad \text{Decompose} \\ X \equiv 0 \wedge s(0) \equiv Y \\ \downarrow \qquad \qquad \qquad \text{Switch} \\ X \equiv 0 \wedge Y \equiv s(0) \end{array}$$

New Transformation Rules

Switch

$$t \equiv X$$

↓

$$X \equiv t$$

Switch rule applies only if
lhs is not originally a
variable

Example

$$f(X, X) \equiv f(Y, Y + 1)$$

↓ Decompose

$$X \equiv Y \wedge X \equiv Y + 1$$

↓ Coalesce

$$X \equiv Y \wedge Y \equiv Y + 1$$

↓ Occurs check

fail

Coalesce

$$X \equiv Y + 1 \wedge Y \equiv X$$

↓

$$X \equiv X + 1 \wedge Y \equiv X$$

Similar to Eliminate, except
both *lhs* and *rhs* are variables

Occurs Check

$$X \equiv X + 1$$

↓

fail

lhs cannot occur in *rhs*

$$p(X) \wedge X \equiv X + 1$$

↓ Eliminate

$$p(X + 1) \wedge X \equiv X + 1$$

↓ Eliminate

$$p((X + 1) + 1) \wedge X \equiv X + 1$$

↓ Eliminate

...

Non-termination can result without the occurs check.

Unification Algorithm

Assumptions: s and t are arbitrary terms and $Vars = vars(s) \cup vars(t)$.

Name	Before	After	Condition
Decompose	$P \wedge f(\vec{s}) \equiv f(\vec{t})$	$P \wedge \bigwedge_i s_i \equiv t_i$	
Conflict	$P \wedge f(\vec{s}) \equiv g(\vec{t})$	fail	$f \not\equiv g$
Switch	$P \wedge s \equiv X$	$P \wedge X \equiv s$	$X \in Vars$ $s \notin Vars$
Delete	$P \wedge s \equiv s$	P	
Eliminate	$P \wedge X \equiv s$	$P[s/X] \wedge X \equiv s$	$X \in vars(P)$ $X \notin vars(s)$ $s \notin Vars$
Occurs Check	$P \wedge X \equiv s$	fail	$X \in vars(s)$ $s \notin Vars$
Coalesce	$P \wedge X \equiv Y$	$P[Y/X] \wedge X \equiv Y$	$X, Y \in vars(P)$ $X \not\equiv Y$

- ▶ Conditions ensure that at most one rule applies to each conjunct
- ▶ Algorithm terminates with success when no further rules apply.

Composition of Unifiers (Substitutions)

Definition

If ϕ and θ are substitutions then their *composition* $\phi \circ \theta$ is also a substitution which, for any term t , satisfies the following property:

$$t[\phi \circ \theta] \equiv (t[\phi])[\theta]$$

Composition of Unifiers (Substitutions)

Definition

If ϕ and θ are substitutions then their *composition* $\phi \circ \theta$ is also a substitution which, for any term t , satisfies the following property:

$$t[\phi \circ \theta] \equiv (t[\phi])[\theta]$$

Examples:

$$[a/x] \circ [b/y] = [a/x, b/y]$$

Composition of Unifiers (Substitutions)

Definition

If ϕ and θ are substitutions then their *composition* $\phi \circ \theta$ is also a substitution which, for any term t , satisfies the following property:

$$t[\phi \circ \theta] \equiv (t[\phi])[\theta]$$

Examples:

$$[a/x] \circ [b/y] = [a/x, b/y]$$

$$[g(y)/x] \circ [b/y] = [g(b)/x, b/y]$$

Composition of Unifiers (Substitutions)

Definition

If ϕ and θ are substitutions then their *composition* $\phi \circ \theta$ is also a substitution which, for any term t , satisfies the following property:

$$t[\phi \circ \theta] \equiv (t[\phi])[\theta]$$

Examples:

$$[a/x] \circ [b/y] = [a/x, b/y]$$

$$[g(y)/x] \circ [b/y] = [g(b)/x, b/y]$$

$$[a/x] \circ [b/x] = [a/x]$$

Composition of Unifiers (Substitutions)

Definition

If ϕ and θ are substitutions then their *composition* $\phi \circ \theta$ is also a substitution which, for any term t , satisfies the following property:

$$t[\phi \circ \theta] \equiv (t[\phi])[\theta]$$

Examples:

$$[a/x] \circ [b/y] = [a/x, b/y]$$

$$[g(y)/x] \circ [b/y] = [g(b)/x, b/y]$$

$$[a/x] \circ [b/x] = [a/x]$$

- ▶ Equality of substitutions: $\phi = \theta$ if $x[\phi] = x[\theta]$ for any variable x .
- ▶ Properties: $(\phi \circ \theta) \circ \sigma = \phi \circ (\theta \circ \sigma)$, $\phi \circ [] = \phi$ and $[] \circ \phi = \phi$.
- ▶ Composition is needed to define the notion of a *most general unifier*.

Properties of the Unification Algorithm

- ▶ The algorithm will find a unifier, if it exists.
- ▶ It returns the **most general unifier** (mgu) θ .

Definition

Given any two terms s and t , θ is their mgu if:

$$s[\theta] \equiv t[\theta] \wedge \forall \phi. s[\phi] \equiv t[\phi] \rightarrow \exists \psi. \phi = \theta \circ \psi.$$

Consider $g(g(X))$ and $g(Y)$. Is $[g(3)/Y, 3/X]$ a unifier? Is it the mgu?

- ▶ mgu is **unique** up to alphabetic variance;
- ▶ the algorithm can easily be extended to simultaneous unification on n expressions.

Building-in Axioms

General Scheme:

$$(Ax_1 \cup \textcolor{blue}{Ax}_2) + \textit{unif} \implies Ax_1 + \textit{unif}_{Ax_2}.$$

Some axioms of the theory become built into unification.

Example

Commutative-Unification

$$X + 2 = Y + 3$$

↓

We no longer use \equiv but $=$

$$Y = 2 \wedge X = 3$$

How do we deal with this?

We can add a new transformation rule (**Mutate rule**).

Unification Algorithm for Commutativity

Name	Before	After	Condition
Decompose	$P \wedge f(\vec{s}) = f(\vec{t})$	$P \wedge \bigwedge_i s_i = t_i$	
Conflict	$P \wedge f(\vec{s}) = g(\vec{t})$	fail	$f \neq g$
Switch	$P \wedge s = X$	$P \wedge X = s$	$X \in \text{Vars}$ $s \notin \text{Vars}$
Delete	$P \wedge s = s$	P	
Eliminate	$P \wedge X = s$	$P[s/X] \wedge X = s$	$X \in \text{vars}(P)$ $X \notin \text{vars}(s)$ $s \notin \text{Vars}$
Check	$P \wedge X = s$	fail	$X \in \text{vars}(s)$ $s \notin \text{Vars}$
Coalesce	$P \wedge X = Y$	$P[Y/X] \wedge X = Y$	$X, Y \in \text{vars}(P)$ $X \neq Y$
Mutate	$P \wedge f(s_1, t_1) = f(s_2, t_2)$	$P \wedge s_1 = t_2 \wedge t_1 = s_2$	f is commutative

Decompose and Mutate rules overlap.

Most General Unifiers

For ordinary unification, the mgu is unique, but what happens when new rules are built-into the unification algorithm?

Multiple mgus: Commutative unification

$$X + Y = a + b \longrightarrow \begin{cases} X = a \wedge Y = b \\ X = b \wedge Y = a \end{cases} \quad \text{Both are equally general.}$$

Infinitely many mgus: Associative unification $X + (Y + Z) = (X + Y) + Z$.

$$X + a = a + X \longrightarrow \begin{cases} X = a \\ X = a + a \\ X = a + a + a \\ \dots \end{cases} \quad \begin{array}{ll} \text{All independent} & \\ (\text{not unifiable}). & \end{array}$$

No mgus: Build in $f(0, X) = X$ and $g(f(X, Y)) = g(Y)$:

$$g(X) = g(a) \longrightarrow \begin{cases} X = a \\ X = f(Y_1, a) \\ X = f(Y_1, f(Y_2, a)) \end{cases} \quad \begin{array}{l} \text{Many unifiers} \\ \text{but no mgu.} \end{array}$$

Types of Unification

Unitary A single unique mgu, or none (predicate logic).

Finitary Finite number of mgus (predicate logic with commutativity).

Infinitary Possibly infinite number of mgus (predicate logic with associativity).

Nullary No mgus exist, although unifiers may exist.

Undecidable Unification not decidable – no algorithm.

Types of Unification

Axioms	Type	Decidable
nil	unitary	yes
commutative	finitary	yes
associative	infinitary	yes
assoc. + dist.	infinitary	yes
lambda calculus	infinitary	no
λ -calculus pattern fragment	unitary	yes

Summary

- ▶ Unification (Bundy Ch. 17.1 - 17.4)
 - ▶ Algorithms for matching and unification.
 - ▶ Unification as equation solving.
 - ▶ Transformation rules for equation solving.
 - ▶ Building-in axioms.(E-Unification/Semantic Unification)
 - ▶ Most general unifiers and classification.
- ▶ Next time: Proof by rewriting

Automated Reasoning

Lecture 12: Rewriting I

Jacques Fleuriot
`jdf@inf.ed.ac.uk`

Recap

- ▶ Previously:
 - ▶ Unification
- ▶ This time: Rewriting
 - ▶ Sets of rewrite rules
 - ▶ Termination
 - ▶ Rewriting in Isabelle

Term Rewriting

Rewriting is a technique for replacing terms in an expression with equivalent terms.

For example, the rules:

$$x * 0 \Rightarrow 0$$

$$x + 0 \Rightarrow x$$

can be used to simplify an expression:

$$x + (\underline{x * 0}) \longrightarrow \underline{x + 0} \longrightarrow x$$

We use the notation $L \Rightarrow R$ to define a rewrite rule that replaces the term L with the term R in an expression and $s \longrightarrow t$ to denote a rewrite rule *application*, where expression s gets rewritten to an expression t .

In general, rewrite rules contain (meta-)variables (e.g., $X + 0 \Rightarrow X$), and are instantiated using **matching** (one-way unification).

The Power of Rewrites

$$0 + N \Rightarrow N \quad (1)$$

$$(0 \leq N) \Rightarrow \text{True} \quad (2)$$

$$s(M) + N \Rightarrow s(M + N) \quad (3)$$

$$s(M) \leq s(N) \Rightarrow M \leq N \quad (4)$$

Given this set of rules:

We can prove this statement:

$$\begin{aligned} & 0 + s(0) \leq s(0) + x \\ \rightarrow & \underline{s(0) \leq s(0) + x} \quad \text{by (1)} \\ \rightarrow & \underline{s(0) \leq s(0 + x)} \quad \text{by (3)} \\ \rightarrow & \underline{0 \leq 0 + x} \quad \text{by (4)} \\ \rightarrow & \text{True} \quad \text{by (2)} \end{aligned}$$

Symbolic Computation

$$0 + N \Rightarrow N \quad (1)$$

$$s(M) + N \Rightarrow s(M + N) \quad (2)$$

$$0 * N \Rightarrow 0 \quad (3)$$

$$s(M) * N \Rightarrow (M * N) + N \quad (4)$$

Given this set of rules:

($s(x)$ means “successor of x ”, i.e. $1 + x$)

We can rewrite $2 * x$ to $x + x$:

$$\begin{aligned} & s(s(0)) * x \\ \longrightarrow & (s(0) * x) + x \quad \text{by (4)} \\ \longrightarrow & ((0 * x) + x) + x \quad \text{by (4)} \\ \longrightarrow & (0 + x) + x \quad \text{by (3)} \\ \longrightarrow & x + x \quad \text{by (1)} \end{aligned}$$

Rewrite Rule of Inference

$$\frac{P\{t\} \quad L \Rightarrow R \quad L[\theta] \equiv t}{P\{R[\theta]\}}$$

where $P\{t\}$ means that P contains t somewhere inside it.

Note: rewriting uses **matching**, not unification (the substitution θ is not applied to t).

Example

Given an expression $(s(a) + s(0)) + s(b)$
and a rewrite rule $s(X) + Y \Rightarrow s(X + Y)$
we can find $t = s(a) + s(0)$
and $\theta = [a/X, s(0)/Y]$

to yield $s(a + s(0)) + s(b)$

Restrictions

A rewrite rule $\alpha \Rightarrow \beta$ must satisfy the following restrictions:

- ▶ α is not a variable.

For example, $x \Rightarrow x + 0$ is not allowed. If the LHS can match anything, then it's very hard to control.

- ▶ $\text{vars}(\beta) \subseteq \text{vars}(\alpha)$.

This rules out $0 \Rightarrow 0 \times x$ for example. This ensures that if we start with a ground term, we will always have a ground term.

More on Notation

- ▶ Rewrite rules: $L \Rightarrow R$, as we've seen already.
- ▶ Rewrite rule applications: $s \rightarrow t$
e.g., $s(s(0)) * x \rightarrow (s(0) * x) + x$
- ▶ Multiple (*zero or more*) rewrite rule applications: $s \rightarrow^* t$
e.g., $s(s(0)) * x \rightarrow^* x + x$
e.g., $0 \rightarrow^* 0$
- ▶ Back-and-forth:
 - ▶ $s \leftrightarrow t$ for $s \rightarrow t$ or $t \rightarrow s$
 - ▶ $s \leftrightarrow^* t$ for a chain of zero or more u_i such that
 $s \leftrightarrow u_1 \leftrightarrow \dots \leftrightarrow u_n \leftrightarrow t$

Logical Interpretation

A rewrite rule $L \Rightarrow R$ on its own is just a “replace” instruction.

To be useful, it must have some logical meaning attached to it.

Most commonly, a rewrite $L \Rightarrow R$ means that $L = R$;

- ▶ Rewrites can instead be based on implications and other formulas (e.g., $a = b \text{ mod } n$), but care is needed to make sure that rewriting corresponds to logically valid steps.

e.g., if $A \rightarrow B$ means A implies B , then it is safe to rewrite A to B in $A \wedge C$, but not in $\neg A \wedge C$. Why?

How to choose rewrite rules?

There are often many equalities to choose from:

$$X + Y = Y + X \quad X + (Y + Z) = (X + Y) + Z \quad X + 0 = X$$

$$0 + X = X \quad 0 + (X + Y) = Y + X \quad \dots$$

Could all be valid rewrite rules.

But: *Not everything that can be rewrite rule should be a rewrite rule!*

- ▶ Ideally, a set of rewrite rules should be **terminating**
- ▶ Ideally, they should rewrite to a **canonical normal form**

An Example: Algebraic Simplification

Rules:

$$x * 0 \Rightarrow 0 \quad (1)$$

$$1 * x \Rightarrow x \quad (2)$$

$$x^0 \Rightarrow 1 \quad (3)$$

$$x + 0 \Rightarrow x \quad (4)$$

Example:

$$\underline{a^{2*0}} * 5 + b * 0$$

$$\longrightarrow \underline{a^0} * 5 + b * 0 \quad \text{by (1)}$$

$$\longrightarrow \underline{1 * 5} + b * 0 \quad \text{by (3)}$$

$$\longrightarrow 5 + \underline{b * 0} \quad \text{by (2)}$$

$$\longrightarrow 5 + \underline{0} \quad \text{by (1)}$$

$$\longrightarrow \frac{5}{5} \quad \text{by (4)}$$

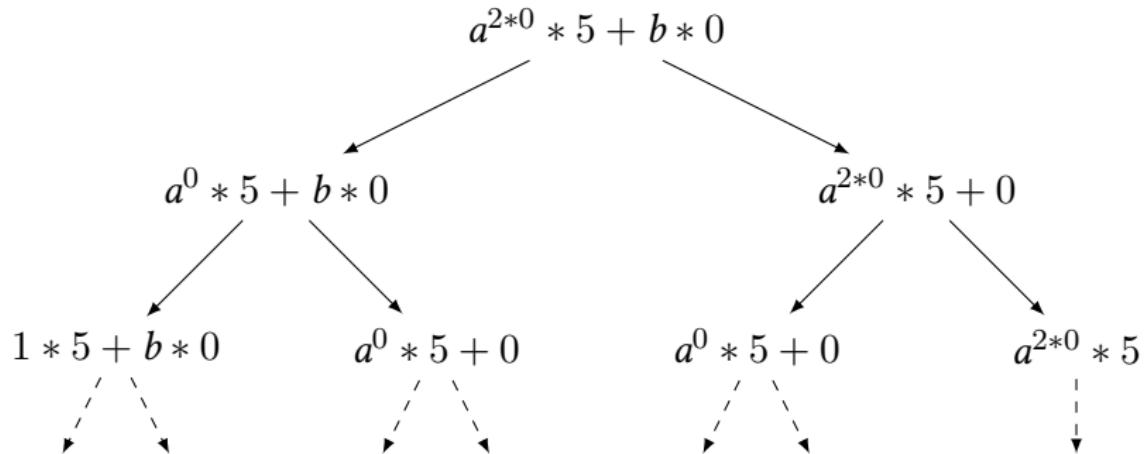
Any subexpression that can be rewritten (i.e. matches the LHS of a rewrite rule) is called a **redex** (*reducible expression*).

The redexes used (but *not* all redexes) have been underlined above.

Choices: Which redex to choose? Which rule to choose?

The Rewrite Search Tree

In general, get a tree of possible rewrites:



Common strategies:

- ▶ Innermost (inside-out) leftmost redex
- ▶ Outermost (outside-in) leftmost redex

Important questions:

- ▶ Is the tree finite? (does the rewriting always terminate?)
- ▶ Does it matter which path we take? (is every leaf the same?)

Termination

We say that a set of rewrite rules is **terminating** if:

starting with any expression, successively applying rewrite rules eventually brings us to a state where no rule applies.

Also called (*strongly*) *normalizing* or *noetherian*.

All the rewrite sets so far in this lecture are terminating

Examples of rules that *may* cause non-termination:

- ▶ Reflexive rules: e.g. $0 \Rightarrow 0$
- ▶ Self-commuting rewrites: e.g. $X * Y \Rightarrow Y * X$, but not with a lexicographical measure.
- ▶ Commuting pairs of rewrites: e.g.:
$$X + (Y + Z) \Rightarrow (X + Y) + Z \text{ and } (X + Y) + Z \Rightarrow X + (Y + Z)$$

An expression to which no rewrite rules apply is called a **normal form** (with respect to that set of rewrite rules).

Proving Termination

Termination can be shown in some cases by:

1. defining a natural number **measure** on expressions
2. such that each rewrite rule decreases the measure

Measure cannot go below zero, so any sequence will terminate.

Example:

$$x * 0 \Rightarrow 0 \quad (1)$$

$$1 * x \Rightarrow x \quad (2)$$

$$x^0 \Rightarrow 1 \quad (3)$$

$$x + 0 \Rightarrow x \quad (4)$$

For these rules, define the **measure** of an expression as the number of binary operations (+, -, *) it contains.

Every rule removes a binary operation, so each rule application will reduce the overall measure of an expression.

In general: look for a **well-founded termination order** (e.g., lexicographical path ordering (LPO))

Examples (from Past Exams)

- ▶ Consider the following rewrite rule:

$$f(f(x)) \Rightarrow f(g(f(x)))$$

Is it terminating? If so, why?

- ▶ How about:

$$-(x + y) \Rightarrow (- - x + y) + y$$

where x and y are variables? Can you show that it is non-terminating?

Interlude: Rewriting in Isabelle

Isabelle has two rules for primitive rewriting (useful with `erule`):

$$\begin{aligned}\text{subst} & : \llbracket ?s = ?t; ?P ?s \rrbracket \implies ?P ?t \\ \text{ssubst} & : \llbracket ?t = ?s; ?P ?s \rrbracket \implies ?P ?t\end{aligned}$$

The $?P$ is matched against the term using *higher-order unification*.

There is also a tactic that rewrites using a theorem:

<code>apply (subst theorem)</code>	: rewrites goal using <i>theorem</i>
<code>apply (subst (asm) theorem)</code>	: rewrites assumptions using <i>theorem</i>
<code>apply (subst (i₁ i₂...) theorem)</code>	: rewrites goal at positions i_1, i_2, \dots
<code>apply (subst (asm) (i₁ i₂...) theorem)</code>	: rewrites assumptions at positions i_1, i_2, \dots

Working out what the right positions are is essentially just trial and error, and can be quite brittle.

The Isabelle Simplifier

The methods (tactics) `simp` and `auto`:

- ▶ `simp` does automatic rewriting on the first subgoal, using a database of rules also known as a *simpset*.
- ▶ `auto` simplifies *all* subgoals, not just the first one.
- ▶ `auto` also applies all obvious logical (Natural Deduction) steps:
 - ▶ splitting conjunctive goals and disjunctive assumptions
 - ▶ quantifier removals – which ones?

Adding `[simp]` after a lemma (or theorem) name when *declaring* it adds that lemma to the simplifier's database/simpset.

- ▶ If it is not an equality, then it is treated as $P = \text{True}$.
- ▶ Many rules are already added to the *default* simpset – so the simplifier often appears quite magical.

The Isabelle Simplifier

Variations on `simp` and `auto` enable *control* over the rules used:

- ▶ `simp add: ... del: : ...`
- ▶ `simp only: : ...`
- ▶ `simp (no_asm)` – ignore assumptions
- ▶ `simp (no_asm_simp)` – use assumptions, but do not rewrite them
- ▶ `simp (no_asm_use)` – rewrite assumptions, don't use them
- ▶ `auto simp add: ... del: ...`

A few specialised simpsets (for arithmetic reasoning):

- ▶ `add_ac` and `mult_ac`: associative/commutative properties of addition and multiplication
- ▶ `algebra_simps`: useful for multiplying out polynomials
- ▶ `field_simps`: useful for multiplying out denominators when proving inequalities e.g. `auto simp add: field_simps`

Note Every definition `defn` in Isabelle generates an associated rewrite rule `defn_def`.

The Isabelle Simplifier

The Isabelle simplifier also has more bells and whistles:

1. Conditional rewriting: Apply $\llbracket P_1; \dots; P_n \rrbracket \Rightarrow s = t$ if
 - ▶ the lhs s matches some expression and
 - ▶ Isabelle can *recursively* prove P_1, \dots, P_n by rewriting.

Example: $\overbrace{[a \neq 0; b \neq 0]}^{\text{prove}} \Rightarrow \overbrace{b/(a * b)}^{\text{match}} = 1/a$

2. (Termination of) Ordered rewriting: a lexicographical (dictionary) ordering is used to *prevent* (some) loops like:

$$a + b \longrightarrow b + a \longrightarrow a + b \longrightarrow \dots$$

Using $x + y = y + x$ as a rewrite rule is actually okay in Isabelle.

3. Case splitting:

$$\begin{aligned} & ?P (\text{case } ?x \text{ of True } \Rightarrow ?f_1 | \text{False } \Rightarrow ?f_2) \\ &= ((?x = \text{True} \longrightarrow ?P ?f_1) \wedge (?x = \text{False} \longrightarrow ?P ?f_2)) \end{aligned}$$

Applies when there is an explicit case split in the goal

Summary

- ▶ Rewriting (Bundy Ch. 9)
 - ▶ Rewriting expressions using rules
 - ▶ Termination (by strictly decreasing measure)
- ▶ Rewriting in Isabelle (Isabelle Tutorial, Section 3.1)
- ▶ Next time: More on Rewriting

Automated Reasoning

Lecture 13: Rewriting II

Jacques Fleuriot
`jdf@inf.ed.ac.uk`

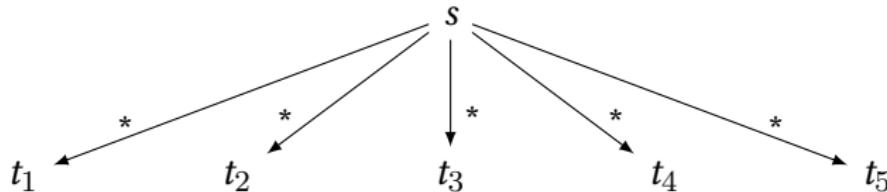
Recap

- ▶ Previously: Rewriting
 - ▶ Definition of Rewrite Rule of Inference
 - ▶ Termination
 - ▶ Rewriting in Isabelle
- ▶ This time: More of the same!
 - ▶ Canonical normal forms
 - ▶ Confluence
 - ▶ Critical Pairs
 - ▶ Knuth-Bendix Completion

Canonical Normal Form

For some rewrite rule sets, order of application might affect result.

We might have:



where all of t_1, t_2, t_3, t_4, t_5 are in normal form after multiple (zero or more) rewrite rule applications.

If all the normal forms are identical we can say we have a **canonical** normal form for s .

This is a very nice property!

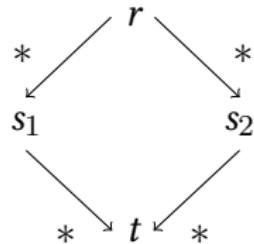
- ▶ Means that order of rewrite rule application doesn't matter
- ▶ In general, means our rewrites are simplifying the expression in a canonical (safe) way.

Confluence and Church-Rosser

How do we know when a set of rules yields canonical normal forms?

A set of rewrite rules is **confluent** if for all terms r , s_1 , s_2 such that $r \rightarrow^* s_1$ and $r \rightarrow^* s_2$ there exists a term t such that $s_1 \rightarrow^* t$ and $s_2 \rightarrow^* t$.

A set of rewrite rules is **Church-Rosser** if for all terms s_1 and s_2 such that $s_1 \leftrightarrow^* s_2$, there exists a term t such that $s_1 \rightarrow^* t$ and $s_2 \rightarrow^* t$.



Theorem

Church-Rosser is equivalent to confluence.

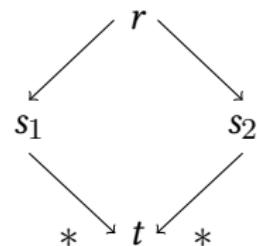
Theorem

For terminating rewrite sets, these properties mean that any expression will rewrite to a canonical normal form.

Local Confluence

The properties of Church-Rosser and confluence can be difficult to prove. A weaker definition is useful:

A set of rewrite rules is **locally confluent** if for all terms r, s_1, s_2 such that $r \rightarrow s_1$ and $r \rightarrow s_2$ there exists a term t such that $s_1 \rightarrow^* t$ and $s_2 \rightarrow^* t$.



Theorem (Newman's Lemma)

$$\text{local confluence} + \text{termination} = \text{confluence}$$

Also: local confluence is decidable (due to Knuth and Bendix)

Both theorem and the decision procedure use idea of **critical pairs**

Choices in Rewriting

How can choices arise in rewriting?

- ▶ Multiple rules apply to a single redex: **order might matter**
- ▶ Rules apply to multiple redexes:
 - ▶ if they are separate: **order does not matter**
 - ▶ if one contains the other: **order might matter**

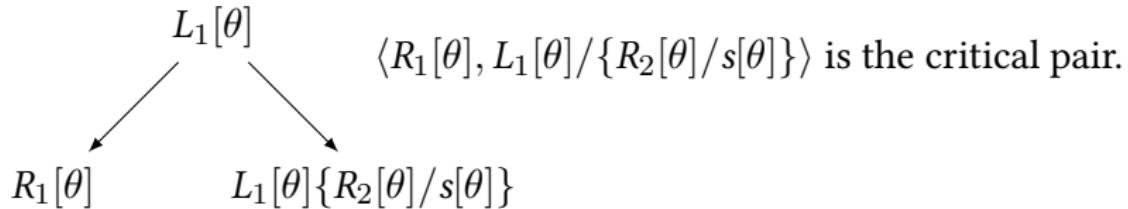
We are interested in cases where the order matters:

Rules	Rewrites	Critical Pair
$X^0 \Rightarrow 1$	0^0 rewrites to 0 and to 1	$\langle 0, 1 \rangle$
$0^Y \Rightarrow 0$		
$X \cdot e \Rightarrow X$	$(x \cdot e) \cdot z$ rewrites to	$\langle x \cdot z, x \cdot (e \cdot z) \rangle$
$(X \cdot Y) \cdot Z \Rightarrow X \cdot (Y \cdot Z)$	$x \cdot z$ and $x \cdot (e \cdot z)$	

Critical Pairs

Given two rules $L_1 \Rightarrow R_1$ and $L_2 \Rightarrow R_2$, we are concerned with the case when there exists a *non-variable* sub-term s of L_1 such that $s[\theta] = L_2[\theta]$, with most general unifier θ .

Applying these rules in different orders gives rise to a **critical pair**, where $L_1[\theta]\{R_2[\theta]/s[\theta]\}$ denotes replacing $s[\theta]$ by $R_2[\theta]$ in $L_1[\theta]$.



Note: the variables in the two rules should be *renamed* so they do **not** share any variable names.

Note: A rewrite rule may have critical pairs with itself e.g. consider the rule $f(f(x)) \Rightarrow g(x)$.

With $W \cdot e \Rightarrow W$ and $(X \cdot Y) \cdot Z \Rightarrow X \cdot (Y \cdot Z)$, where X , Y and Z are variables, we can have $\theta = [W/X, e/Y]$, **any other?**

Critical Pairs: Example

Consider the rewrite rules:

$$\begin{array}{ccc} \overbrace{f(f(x, y), z)}^{L_1} & \Rightarrow & \overbrace{f(x, f(y, z))}^{R_1} \\ s & & \\ \overbrace{f(i(x_1), x_1)}^{L_2} & \Rightarrow & \overbrace{e}^{R_2} \end{array}$$

The mgu θ , given our choice of non-variable subterm s of L_1 , is given by $\theta = \{i(x_1)/x, x_1/y\}$ and by considering:

$$\begin{array}{ccc} f(f(i(x_1), x_1), z) & & \\ \searrow & & \searrow \\ f(i(x_1), f(x_1, z)) & & f(e, z) \end{array}$$

We get the critical pair $\langle f(i(x_1), f(x_1, z)), f(e, z) \rangle$.

Testing for Local Confluence

If we can **conflate** (join) all the critical pairs, then have **local confluence**.

Conflation for a critical pair $\langle s_1, s_2 \rangle$ is when there is a t such that $s_1 \longrightarrow^* t$ and $s_2 \longrightarrow^* t$.

An algorithm to test for local confluence (assuming termination):

1. Find all the critical pairs in set of rewrite rules R
2. For each critical pair $\langle s_1, s_2 \rangle$:
 - 2.1 Find a normal form s'_1 of s_1 ;
 - 2.2 Find a normal form s'_2 of s_2 ;
 - 2.3 Check $s'_1 = s'_2$, if not then fail.

Establishing Local Confluence

Sometimes a set of rules is not locally confluent

$X \cdot e \Rightarrow X$
 $f \cdot X \Rightarrow X$ is not locally confluent: $\langle f, e \rangle$ does not conflate.

We can add the rule $f \Rightarrow e$ to make this critical pair joinable.

However, adding new rules requires **care**:

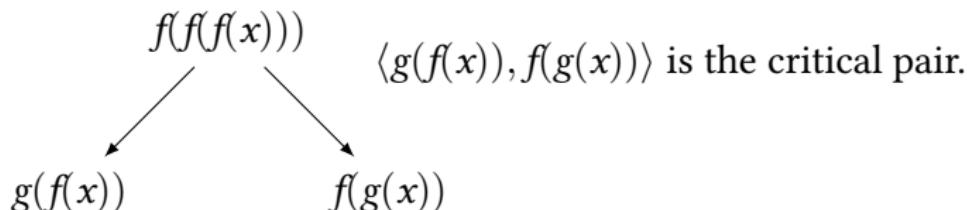
- ▶ Must preserve termination
- ▶ Might give rise to *new* critical pairs and so we may need to check local confluence again.

Establishing Local Confluence: Example

Consider the set R consisting of just one rewrite rule, with x a variable:

$$f(f(x)) \Rightarrow g(x)$$

which has exactly one critical pair (CP) when it is overlapped with a *renamed* copy of itself $f(f(y)) \Rightarrow g(y)$. The lhs $f(f(x))$ unifies with the subterm $f(y)$ of the renamed lhs to produce the mgu $\{f(x)/y\}$:



- ▶ This CP is not joinable, so R is not locally confluent.
- ▶ Adding the rule $f(g(x)) \Rightarrow g(f(x))$ to R makes the pair joinable.
- ▶ The enlarged R is terminating (how?), but
- ▶ (After renaming) new CP: $\langle g(g(z)), f(g(f(z))) \rangle$ arises (how?);
- ▶ LC test: it is joinable, $f(g(f(z))) \rightarrow g(f(f(z))) \rightarrow g(g(z))$.

Knuth-Bendix (KB) Completion Algorithm

Start with a set R of terminating rewrite rules

While there are non-conflatable critical pairs in R :

1. Take a critical pair $\langle s_1, s_2 \rangle$ in R
2. Normalise s_1 to s'_1 and s_2 to s'_2 (and we know $s'_1 \neq s'_2$)
3. if $R \cup \{s'_1 \Rightarrow s'_2\}$ is terminating then

$$R := R \cup \{s'_1 \Rightarrow s'_2\}$$

else if $R \cup \{s'_2 \Rightarrow s'_1\}$ is terminating then

$$R := R \cup \{s'_2 \Rightarrow s'_1\}$$

else Fail

- ▶ If KB succeeds then we have a locally confluent and terminating (and hence confluent) rewrite set (KB may run forever!)
- ▶ Depends on the termination check: define a measure and use that to test for termination.

Summary

- ▶ Rewriting (Bundy Ch. 9)
 - ▶ Local confluence
 - ▶ Local confluence + Termination = Confluence
 - ▶ Canonical Normal Forms
 - ▶ Critical Pairs and Knuth-Bendix Completion

Automated Reasoning

Lecture 14: Inductive Proof (in Isabelle)

Jacques Fleuriot
`jdf@inf.ed.ac.uk`

Recap

- ▶ Previously:
 - ▶ Unification and Rewriting
- ▶ This time: Proof by Induction (in Isabelle)
 - ▶ Proof by Mathematical Induction
 - ▶ Structural Recursion and Induction
 - ▶ Challenges in Inductive Proof Automation

A Summation Problem

What is

$$1 + 2 + 3 + \dots + 999 + 1000 \quad ?$$

Is there a general formula for any n ?

A Summation Problem

What is

$$1 + 2 + 3 + \dots + 999 + 1000 \quad ?$$

Is there a general formula for any n ?

Gauss's solution:

$$1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

A Summation Problem

What is

$$1 + 2 + 3 + \dots + 999 + 1000 \quad ?$$

Is there a general formula for any n ?

Gauss's solution:

$$1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

How can we prove this? (Automatically?)

- ▶ First-order proof search is (generally) unable to prove this

Proof by Induction

To prove $\forall n. P n$:

- { (base) prove $P 0$
- { (step) for all n , assume $P n$ and prove $P (n + 1)$

Proof by Induction

To prove $\forall n. P n$:

- { (base) prove $P 0$
- { (step) for all n , assume $P n$ and prove $P (n + 1)$

To prove $\forall n. 1 + 2 + \dots + n = \frac{n(n+1)}{2}$:

Proof by Induction

To prove $\forall n. P n$:

- { (base) prove $P 0$
- { (step) for all n , assume $P n$ and prove $P (n + 1)$

To prove $\forall n. 1 + 2 + \dots + n = \frac{n(n+1)}{2}$:

(base): $0 = \frac{0*1}{2}$, by computation.

Proof by Induction

To prove $\forall n. P n$:

- { (base) prove $P 0$
- { (step) for all n , assume $P n$ and prove $P (n + 1)$

To prove $\forall n. 1 + 2 + \dots + n = \frac{n(n+1)}{2}$:

(base): $0 = \frac{0*1}{2}$, by computation.

(step): assume the formula holds for n , and:

$$\begin{aligned} & 1 + 2 + \dots + n + (n + 1) \\ = & (1 + 2 + \dots + n) + (n + 1) \\ = & \frac{n(n+1)}{2} + (n + 1) \quad (\text{apply induction hypothesis}) \\ = & \dots \\ = & \frac{(n+1)(n+2)}{2} \end{aligned}$$

as required.

Inductively Defined Data

Induction is especially useful for dealing with **Inductive Datatypes**

Inductive Datatypes are *freely generated* by some constructors.

Free datatypes are those for which terms are only equal if they are syntactically identical e.g. $\text{Succ}(\text{Succ Zero}) \neq \text{Succ Zero}$.

datatype $nat = \text{Zero} \mid \text{Succ } nat$

datatype $'a\ list = \text{Nil} \mid \text{Cons } "'a" \ "'a\ list"$

Some values:
$$\begin{cases} \text{Succ}(\text{Succ Zero}) & \text{i.e. "2"} \\ \text{Cons Zero}(\text{Cons Zero Nil}) & \text{i.e. "[0, 0]"} \end{cases}$$

Non-freely generated datatypes. Contrast the above with the integers, for example, defined with the constructors Zero , Succ and Pred , where Zero and Succ are as for the natural numbers but Pred is the predecessor function.

In this case, $\text{Pred}(\text{Succ } n) = \text{Suc}(\text{Pred } n) = n$, for instance.

datatype — the general case

$$\begin{array}{lll} \textbf{datatype } (\alpha_1, \dots, \alpha_n)t & = & C_1 \ \tau_{1,1} \dots \tau_{1,n_1} \\ & | & \dots \\ & | & C_k \ \tau_{k,1} \dots \tau_{k,n_k} \end{array}$$

datatype — the general case

$$\begin{array}{ccl} \textbf{datatype } (\alpha_1, \dots, \alpha_n)t & = & C_1 \ \tau_{1,1} \dots \tau_{1,n_1} \\ & | & \dots \\ & | & C_k \ \tau_{k,1} \dots \tau_{k,n_k} \end{array}$$

- ▶ *Types:* $C_i :: \tau_{i,1} \Rightarrow \dots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \dots, \alpha_n)t$

datatype — the general case

$$\begin{array}{ccl} \textbf{datatype } (\alpha_1, \dots, \alpha_n)t & = & C_1 \ \tau_{1,1} \dots \tau_{1,n_1} \\ & | & \dots \\ & | & C_k \ \tau_{k,1} \dots \tau_{k,n_k} \end{array}$$

- ▶ *Types:* $C_i :: \tau_{i,1} \Rightarrow \dots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \dots, \alpha_n)t$
- ▶ *Distinctness:* $C_i \dots \neq C_j \dots$ if $i \neq j$

datatype — the general case

$$\begin{array}{ccl} \textbf{datatype } (\alpha_1, \dots, \alpha_n)t & = & C_1 \ \tau_{1,1} \dots \tau_{1,n_1} \\ & | & \dots \\ & | & C_k \ \tau_{k,1} \dots \tau_{k,n_k} \end{array}$$

- ▶ *Types:* $C_i :: \tau_{i,1} \Rightarrow \dots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \dots, \alpha_n)t$
- ▶ *Distinctness:* $C_i \dots \neq C_j \dots$ if $i \neq j$
- ▶ *Injectivity:* $(C_i \ x_1 \dots x_{n_i} = C_i \ y_1 \dots y_{n_i}) = (x_1 = y_1 \wedge \dots \wedge x_{n_i} = y_{n_i})$

datatype — the general case

$$\begin{array}{rcl} \textbf{datatype } (\alpha_1, \dots, \alpha_n)t & = & C_1 \ \tau_{1,1} \dots \tau_{1,n_1} \\ & | & \dots \\ & | & C_k \ \tau_{k,1} \dots \tau_{k,n_k} \end{array}$$

- ▶ *Types:* $C_i :: \tau_{i,1} \Rightarrow \dots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \dots, \alpha_n)t$
- ▶ *Distinctness:* $C_i \dots \neq C_j \dots$ if $i \neq j$
- ▶ *Injectivity:* $(C_i \ x_1 \dots x_{n_i} = C_i \ y_1 \dots y_{n_i}) = (x_1 = y_1 \wedge \dots \wedge x_{n_i} = y_{n_i})$

Distinctness and injectivity are applied automatically
Induction must be applied explicitly

Recursive Functions on Inductively Defined Data

Functions can be defined by recursion on "structurally smaller" data.

`primrec length :: "'a list ⇒ nat"`

`where`

`"length Nil = Zero" |`

`"length (Cons x xs) = Succ (length xs)"`

Recursive Functions on Inductively Defined Data

Functions can be defined by recursion on "structurally smaller" data.

primrec length :: "*'a list* \Rightarrow *nat*"

where

"length Nil = Zero" |

"length (Cons x xs) = Succ (length xs)"

primrec append :: "*'a list* \Rightarrow *'a list* \Rightarrow *'a list*"

where

"append Nil ys = ys" |

"append (Cons x xs) ys = Cons x (append xs ys)"

Recursive Functions on Inductively Defined Data

Functions can be defined by recursion on "structurally smaller" data.

primrec length :: "'a list \Rightarrow nat"

where

"length Nil = Zero" |

"length (Cons x xs) = Succ (length xs)"

primrec append :: "'a list \Rightarrow 'a list \Rightarrow 'a list"

where

"append Nil ys = ys" |

"append (Cons x xs) ys = Cons x (append xs ys)"

primrec reverse :: "'a list \Rightarrow 'a list"

where

"reverse Nil = Nil" |

"reverse (Cons x xs) = append (reverse xs) (Cons x Nil)"

Proof by Structural Induction

Properties of structurally recursive functions can be proved by **structural induction**.

To show $\forall xs. P xs$:

{ prove $P \text{ Nil}$
for all x, xs , assume $P xs$ to prove $P (\text{Cons } x xs)$

Proof by Structural Induction

Properties of structurally recursive functions can be proved by **structural induction**.

To show $\forall xs. P xs$:

$\begin{cases} \text{prove } P \text{ Nil} \\ \text{for all } x, xs, \text{ assume } P xs \text{ to prove } P (\text{Cons } x xs) \end{cases}$

To prove: $\text{append } xs (\text{append } ys zs) = \text{append } (\text{append } xs ys) zs$:

Proof by Structural Induction

Properties of structurally recursive functions can be proved by **structural induction**.

To show $\forall xs. P xs$:

$\begin{cases} \text{prove } P \text{ Nil} \\ \text{for all } x, xs, \text{ assume } P xs \text{ to prove } P (\text{Cons } x xs) \end{cases}$

To prove: $\text{append } xs (\text{append } ys zs) = \text{append } (\text{append } xs ys) zs$:

(base)

$$\begin{aligned} \text{append Nil } (\text{append } ys zs) &= \text{append } ys zs \\ &= \text{append } (\text{append Nil } ys) zs \end{aligned}$$

Proof by Structural Induction

Properties of structurally recursive functions can be proved by **structural induction**.

To show $\forall xs. P xs$:

$\begin{cases} \text{prove } P \text{ Nil} \\ \text{for all } x, xs, \text{ assume } P xs \text{ to prove } P (\text{Cons } x xs) \end{cases}$

To prove: $\text{append } xs (\text{append } ys zs) = \text{append } (\text{append } xs ys) zs$:

(base)

$$\begin{aligned} \text{append Nil } (\text{append } ys zs) &= \text{append } ys zs \\ &= \text{append } (\text{append Nil } ys) zs \end{aligned}$$

(step)

$$\begin{aligned} &\text{append } (\text{Cons } x xs) (\text{append } ys zs) \\ &= \text{Cons } x (\text{append } xs (\text{append } ys zs)) \\ &= \text{Cons } x (\text{append } (\text{append } xs ys) zs) \quad \text{by IH} \\ &= \text{append } (\text{Cons } x (\text{append } xs ys)) zs \\ &= \text{append } (\text{append } (\text{Cons } x xs) ys) zs \end{aligned}$$

Proof by Structural Induction

Properties of structurally recursive functions can be proved by **structural induction**.

To show $\forall xs. P xs$:

$\begin{cases} \text{prove } P \text{ Nil} \\ \text{for all } x, xs, \text{ assume } P xs \text{ to prove } P (\text{Cons } x xs) \end{cases}$

To prove: $\text{append } xs (\text{append } ys zs) = \text{append } (\text{append } xs ys) zs$:

(base)

$$\begin{aligned} \text{append Nil } (\text{append } ys zs) &= \text{append } ys zs \\ &= \text{append } (\text{append Nil } ys) zs \end{aligned}$$

(step)

$$\begin{aligned} &\text{append } (\text{Cons } x xs) (\text{append } ys zs) \\ &= \text{Cons } x (\text{append } xs (\text{append } ys zs)) \\ &= \text{Cons } x (\text{append } (\text{append } xs ys) zs) \quad \text{by IH} \\ &= \text{append } (\text{Cons } x (\text{append } xs ys)) zs \\ &= \text{append } (\text{append } (\text{Cons } x xs) ys) zs \end{aligned}$$

In practice: start with the equation to be proved as the goal, and rewrite both sides to be equal.

Structural induction for *list*

This is analogous to the one for natural numbers (see the lecture on Isar).

```
show P(xs)
proof (induction xs)
  case Nil
  :
  show ?case
next
  case (Cons x xs)
  :
  show ?case
qed
```

Well-Founded Induction

Let $<$ be an ordering on a set such that, for all x , there are no infinite downward chains:

Not allowed: $\dots < \dots < x_3 < x_2 < x_1 < x$

Such an ordering is called *well-founded* (or *noetherian*)

Well-Founded Induction

Let $<$ be an ordering on a set such that, for all x , there are no infinite downward chains:

Not allowed: $\dots < \dots < x_3 < x_2 < x_1 < x$

Such an ordering is called *well-founded* (or *noetherian*)

Then, to prove $\forall x. P x$, it suffices to prove:

$$\forall y. (\forall z. z < y \rightarrow P z) \rightarrow P y$$

Well-Founded Induction

Let $<$ be an ordering on a set such that, for all x , there are no infinite downward chains:

Not allowed:

$$\dots < \dots < x_3 < x_2 < x_1 < x$$

Such an ordering is called *well-founded* (or *noetherian*)

Then, to prove $\forall x. P x$, it suffices to prove:

$$\forall y. (\forall z. z < y \rightarrow P z) \rightarrow P y$$

Specialised to the natural numbers, with the usual less-than ordering, this is usually called **Complete Induction**.

Theoretical Limitations of Automated Inductive Proof

Recall L -systems, with left- and right-introduction rules:

$$\frac{\Gamma, P, Q \vdash R}{\Gamma, P \wedge Q \vdash R} \text{ (e conjE)}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \text{ (disjI1)}$$

$$\frac{\Gamma \vdash P \quad \Gamma, P \vdash Q}{\Gamma \vdash Q} \text{ (cut)}$$

This system has two nice properties:

1. *Cut elimination*: the cut rule is unnecessary
2. *Sub-formula property*: every cut-free proof only contains formulas which are sub-formulas of the original goal

($Q(t)$ is a sub-formula of $\forall x. Q(x)$ and $\exists x. Q(x)$, for any t)

So can do complete (but possibly non-terminating) proof search.

Theoretical Limitations of Automated Inductive Proof

Recall L -systems, with left- and right-introduction rules:

$$\frac{\Gamma, P, Q \vdash R}{\Gamma, P \wedge Q \vdash R} \text{ (e conjE)}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \text{ (disjI1)}$$

$$\frac{\Gamma \vdash P \quad \Gamma, P \vdash Q}{\Gamma \vdash Q} \text{ (cut)}$$

This system has two nice properties:

1. *Cut elimination*: the cut rule is unnecessary
2. *Sub-formula property*: every cut-free proof only contains formulas which are sub-formulas of the original goal

($Q(t)$ is a sub-formula of $\forall x. Q(x)$ and $\exists x. Q(x)$, for any t)

So can do complete (but possibly non-terminating) proof search.

If we add an induction rule:

$$\frac{\Gamma \vdash P(0) \quad \Gamma, P(n) \vdash P(n+1) \quad n \notin \text{fv}(\Gamma, P)}{\Gamma \vdash \forall n. P(n)}$$

Then Cut elimination fails!

There are variant rules that bring it back, but sub-formula property still fails

The Need for Intermediate Lemmas

Practically, the lack of a guarantee of a proof with the sub-formula property means that we need *creative generalisation* during proofs, or we need to *speculate new lemmas*.

The Need for Intermediate Lemmas

Practically, the lack of a guarantee of a proof with the sub-formula property means that we need *creative generalisation* during proofs, or we need to *speculate new lemmas*.

To prove: $\text{reverse}(\text{reverse } xs) = xs$

The Need for Intermediate Lemmas

Practically, the lack of a guarantee of a proof with the sub-formula property means that we need *creative generalisation* during proofs, or we need to *speculate new lemmas*.

To prove: $\text{reverse}(\text{reverse } xs) = xs$

(base) $\text{reverse}(\text{reverse Nil}) = \text{reverse Nil} = \text{Nil}$

The Need for Intermediate Lemmas

Practically, the lack of a guarantee of a proof with the sub-formula property means that we need *creative generalisation* during proofs, or we need to *speculate new lemmas*.

To prove: $\text{reverse}(\text{reverse } xs) = xs$

(base) $\text{reverse}(\text{reverse } \text{Nil}) = \text{reverse } \text{Nil} = \text{Nil}$

(step) IH: $\text{reverse}(\text{reverse } xs) = xs$

Attempt:

$$\begin{aligned} & \text{reverse}(\text{reverse}(\text{Cons } x xs)) \\ &= \text{reverse}(\text{append}(\text{reverse } xs)(\text{Cons } x \text{Nil})) \\ &\quad \text{????} \\ &= \text{Cons } x xs \end{aligned}$$

The Need for Intermediate Lemmas

Practically, the lack of a guarantee of a proof with the sub-formula property means that we need *creative generalisation* during proofs, or we need to *speculate new lemmas*.

To prove: $\text{reverse}(\text{reverse } xs) = xs$

(base) $\text{reverse}(\text{reverse } \text{Nil}) = \text{reverse } \text{Nil} = \text{Nil}$

(step) IH: $\text{reverse}(\text{reverse } xs) = xs$

Attempt:

$$\begin{aligned} & \text{reverse}(\text{reverse}(\text{Cons } x \text{ xs})) \\ &= \text{reverse}(\text{append}(\text{reverse } xs)(\text{Cons } x \text{ Nil})) \\ &\quad \text{????} \\ &= \text{Cons } x \text{ xs} \end{aligned}$$

We need to *speculate* a new lemma.

A New Lemma

In this case, it turns out that we need:

$$\text{reverse}(\text{append } xs \ ys) = \text{append}(\text{reverse } ys)(\text{reverse } xs)$$

(which is proved by induction, and needs *another* lemma)

A New Lemma

In this case, it turns out that we need:

$$\text{reverse}(\text{append } xs \ ys) = \text{append}(\text{reverse } ys)(\text{reverse } xs)$$

(which is proved by induction, and needs *another* lemma)

Now we can proceed:

(step) IH: $\text{reverse}(\text{reverse } xs) = xs$

Attempt:

$$\begin{aligned} & \text{reverse}(\text{reverse}(\text{Cons } x \ xs)) \\ = & \text{reverse}(\text{append}(\text{reverse } xs)(\text{Cons } x \ \text{Nil})) \\ = & \text{append}(\text{Cons } x \ \text{Nil})(\text{reverse}(\text{reverse } xs)) \quad \text{by lemma} \\ = & \text{Cons } x (\text{append } \text{Nil} (\text{reverse}(\text{reverse } xs))) \\ = & \text{Cons } x (\text{reverse}(\text{reverse } xs)) \\ = & \text{Cons } x \ xs \qquad \qquad \qquad \text{by IH} \end{aligned}$$

Another approach

We got stuck trying to prove:

$$\text{reverse} (\text{append} (\text{reverse } xs) (\text{Cons } x \text{ Nil})) = \text{Cons } x \text{ xs}$$

under the assumption that $\text{reverse} (\text{reverse } xs) = xs$

Another approach

We got stuck trying to prove:

$$\text{reverse}(\text{append}(\text{reverse } xs)(\text{Cons } x \text{ Nil})) = \text{Cons } x \text{ xs}$$

under the assumption that $\text{reverse}(\text{reverse } xs) = xs$

What if we rewrite the RHS *backwards* by the IH, to get the new goal:

$$\text{reverse}(\text{append}(\text{reverse } xs)(\text{Cons } x \text{ Nil})) = \text{Cons } x (\text{reverse}(\text{reverse } xs))$$

Maybe this can be proved by induction?

Another approach

We got stuck trying to prove:

$$\text{reverse}(\text{append}(\text{reverse } xs)(\text{Cons } x \text{ Nil})) = \text{Cons } x \text{ xs}$$

under the assumption that $\text{reverse}(\text{reverse } xs) = xs$

What if we rewrite the RHS *backwards* by the IH, to get the new goal:

$$\text{reverse}(\text{append}(\text{reverse } xs)(\text{Cons } x \text{ Nil})) = \text{Cons } x (\text{reverse}(\text{reverse } xs))$$

Maybe this can be proved by induction?

Not quite (try it and see!); need to *generalise* and prove:

$$\text{reverse}(\text{append } xs(\text{Cons } x \text{ Nil})) = \text{Cons } x (\text{reverse } xs)$$

(A special case of the lemma speculated earlier)

Challenges in Automating Inductive Proofs

Theoretically, and practically, to do inductive proofs, we need:

- ▶ Lemma speculation
- ▶ Generalisation

Techniques (other than "Get the user to do it"):

- ▶ Boyer-Moore approach
 - roughly the approach described here (implemented in ACL2)
- ▶ Rippling, "Productive Use of Failure" (Bundy and Ireland, 1996)
- ▶ Up-front speculation:
 - e.g. "maybe this binary function is associative?"
- ▶ Cyclic proofs
 - (search for a circular proof, and afterwards prove it is well-founded)
- ▶ Only doing a few cases (0, 1, ..., 6)
- ▶ Special purpose techniques (e.g., generating functions)

Summary

- ▶ Proof by Induction (in Isabelle)
 - ▶ Natural number induction
 - ▶ Inductive Datatypes and Structural Induction (H&R 1.4.2)
 - ▶ The automation of Mathematical Induction by Bundy (see AR webpage).
 - ▶ The need for generalisation and lemma speculation



Program verification using Hoare Logic¹

Automated Reasoning - Guest Lecture

Petros Papapanagiotou
pe.p@ed.ac.uk

Part 1 of 2

¹Contains material from Mike Gordon's slides: <http://www.cl.cam.ac.uk/~mjcg/HL>

A simple “while” programming language

- ▶ Sequence: a ; b
- ▶ Skip (do nothing): SKIP
- ▶ Variable assignment: X := 0
- ▶ Conditional: IF cond THEN a ELSE b FI
- ▶ Loop: WHILE cond DO c OD

Example

Given some X

```
Y := 1 ;
Z := 0 ;
WHILE Z ≠ X DO
    Z := Z + 1 ;
    Y := Y × Z
OD
```

$$\{Y = X!\}$$

How do you know for sure?

Formal Methods

- ▶ *Formal Specification:*
 - ▶ Use mathematical notation to give a precise description of what a program should do
- ▶ *Formal Verification:*
 - ▶ Use logical rules to mathematically prove that a program satisfies a formal specification
- ▶ *Not a panacea:*
 - ▶ Formally verified programs may still not work!
 - ▶ Must be combined with testing

Modern use

- ▶ Some use cases:
 - ▶ Safety-critical systems (e.g. medical software, nuclear reactor controllers, autonomous vehicles)
 - ▶ Core system components (e.g. device drivers)
 - ▶ Security (e.g. ATM software, cryptographic algorithms)
 - ▶ Hardware verification (e.g. processors)

Formal Verification

Requires programming language *semantics*

*What does it mean to execute a command C?
How does it affect the State?*

(State = map of memory locations to values)

Formal Verification

- ▶ **Denotational** semantics: construct *mathematical objects* that describe the meaning
 - ▶ Programs = functions: $\llbracket C \rrbracket : State \rightarrow State$
- ▶ **Operational** semantics: describe the *steps of computation* during program execution
 - ▶ Small-step (only one transition): $\langle C, \sigma \rangle \rightarrow \langle C', \sigma' \rangle$
 - ▶ Big-step (entire transition to final value): $\langle C, \sigma \rangle \Downarrow \sigma'$
- ▶ **Axiomatic** semantics: define *axioms and rules of some logic* of programs
 - ▶ Hoare Logic $\{P\} C \{Q\}$

Floyd-Hoare Logic and Partial Correctness Specification

By Charles Antony (“Tony”) Richard Hoare with original ideas from
Robert Floyd - 1969

- ▶ **Specification:** Given a state that satisfies *preconditions* P , executing a *program* C (and assuming it terminates) results in a state that satisfies *postconditions* Q
- ▶ “Hoare triple”:

$$\{P\} \ C \ \{Q\}$$

e.g.:

$$\{X = 1\} \ X := X + 1 \ \{X = 2\}$$

Correctness

$$\{P\} \ C \ \{Q\}$$

Partial correctness + termination = *Total* correctness

Trivial Specifications

$$\{P\} \subset \{\mathbf{T}\}$$

$$\{\mathbf{F}\} \subset \{Q\}$$

Formal specification can be tricky!

- ▶ Specification for the maximum of two variables:

$$\{T\} \ C \ \{Y = \max(X, Y)\}$$

- ▶ C could be:

```
IF X >= Y THEN Y := X ELSE SKIP FI
```

- ▶ *But* C could also be:

```
IF X >= Y THEN X := Y ELSE SKIP FI
```

- ▶ Or even:

```
Y := X
```

- ▶ Better use “auxiliary” variables (i.e. *not* program variables) x and y :

$$\{X = x \wedge Y = y\} \ C \ \{Y = \max(x, y)\}$$

Hoare Logic

- ▶ A deductive proof system for Hoare triples $\{P\} C \{Q\}$
- ▶ Can be used for *verification* with forward or backward chaining
 - ▶ Conditions P and Q are described using FOL
 - ▶ *Verification Conditions (VCs)*: What needs to be proven so that $\{P\} C \{Q\}$ is *true*?
 - ▶ *Proof obligations* or simply *proof subgoals*: Working our way through proving the VCs

Hoare Logic Rules

- ▶ Similar to FOL inference rules
- ▶ One for each programming language construct:
 - ▶ Assignment
 - ▶ Sequence
 - ▶ Skip
 - ▶ Conditional
 - ▶ While
- ▶ Rules of *consequence*:
 - ▶ Precondition strengthening
 - ▶ Postcondition weakening

Assignment Axiom

$$\overline{\{Q[E/V]\} \ V := E \ \{Q\}}$$

- ▶ Example:

$$\{X + 1 = n + 1\} \ X := X + 1 \ \{X = n + 1\}$$

- ▶ Backwards!?

- ▶ Why not $\{P\} \ V := E \ \{P[V/E]\}$?
 - ▶ because then: $\{X = 0\} \ X := 1 \ \{X = 0\}$
- ▶ Why not $\{P\} \ V := E \ \{P[E/V]\}$?
 - ▶ because then: $\{X = 0\} \ X := 1 \ \{1 = 0\}$

Sequencing Rule

$$\frac{\{P\} C_1 \{Q\} \quad \{Q\} C_2 \{R\}}{\{P\} C_1 ; C_2 \{R\}}$$

► Example (Swap X Y): S := X ; X := Y ; Y := S

$$\overline{\{X = x \wedge Y = y\} S := x \{S = x \wedge Y = y\}} \quad (1)$$

$$\overline{\{S = x \wedge Y = y\} X := Y \{S = x \wedge X = y\}} \quad (2)$$

$$\overline{\{S = x \wedge X = y\} Y := S \{Y = x \wedge X = y\}} \quad (3)$$

$$\frac{\begin{array}{c} (1) \qquad \qquad (2) \\ \overline{\{X = x \wedge Y = y\} S := x ; X := Y \{S = x \wedge X = y\}} \end{array}}{\overline{\{X = x \wedge Y = y\} S := x ; X := Y ; Y := S \{Y = x \wedge X = y\}}} \quad (3)$$

Skip Axiom

$$\overline{\{P\} \text{ SKIP } \{P\}}$$

Conditional Rule

$$\frac{\{P \wedge S\} C_1 \{Q\} \quad \{P \wedge \neg S\} C_2 \{Q\}}{\{P\} \text{ IF } S \text{ THEN } C_1 \text{ ELSE } C_2 \text{ FI } \{Q\}}$$

- ▶ Example ($\text{Max } X \ Y$):

$$\frac{\{X \geq X \wedge X \geq Y\} \text{ MAX := } X \{MAX \geq X \wedge MAX \geq Y\} \\ \{T \wedge X \geq Y\} \text{ MAX := } X \{MAX \geq X \wedge MAX \geq Y\}}{(4)}$$

$$\frac{\{Y \geq X \wedge Y \geq Y\} \text{ MAX := } Y \{MAX \geq X \wedge MAX \geq Y\} \\ \{T \wedge \neg(X \geq Y)\} \text{ MAX := } Y \{MAX \geq X \wedge MAX \geq Y\}}{(5)}$$

$$\frac{\text{(4)} \quad \text{(5)}}{\{T\} \text{ IF } X \geq Y \text{ THEN MAX := } X \text{ ELSE MAX := } Y \text{ FI } \{MAX \geq X \wedge MAX \geq Y\}} \quad (6)$$

Summary

- ▶ *Formal Verification:* Use logical rules to mathematically prove that a program satisfies a formal specification
- ▶ Programming language semantics
 - ▶ denotational, operational, axiomatic
- ▶ Specification using *Hoare triples* $\{P\} C \{Q\}$
 - ▶ Preconditions P
 - ▶ Program C
 - ▶ Postconditions Q
- ▶ *Hoare Logic:* A deductive proof system for Hoare triples
- ▶ Logical Rules:
 - ▶ One for each program construct
- ▶ *Partial correctness + termination = Total correctness*

Next

- ▶ Precondition strengthening
- ▶ Postcondition weakening
- ▶ WHILE loops + invariants

To be continued...

Recommended reading

Theory:

- ▶ Mike Gordon, *Background Reading on Hoare Logic*,
<http://www.cl.cam.ac.uk/~mjcg/Teaching/2011/Hoare/Notes/Notes.pdf> (pp. 1-27, 37-48)
- ▶ Huth & Ryan, Sections 4.1-4.3 (pp. 256-292)
- ▶ Nipkow & Klein, Section 12.2.1 (pp. 191-199)

Practice:

- ▶ Isabelle's Hoare Logic library: <http://isabelle.in.tum.de/dist/library/HOL/HOL-Hoare>
- ▶ Tutorial exercise



THE UNIVERSITY of EDINBURGH
informatics

cisa

Centre for Intelligent Systems
and their Applications

Program verification using Hoare Logic¹

Automated Reasoning - Guest Lecture

Petros Papapanagiotou
pe.p@ed.ac.uk

Part 2 of 2

¹Contains material from Mike Gordon's slides: <http://www.cl.cam.ac.uk/~mjcg/HL>

Previously on Hoare Logic

A simple “while” language

- ▶ Sequence: a ; b
- ▶ Skip (do nothing): SKIP
- ▶ Variable assignment: X := 0
- ▶ Conditional:
IF cond THEN a ELSE b FI
- ▶ Loop: WHILE cond DO c OD

Hoare Logic

- ▶ $\{P\} \ C \ \{Q\}$
- ▶ Formal specification
- ▶ Axiomatic semantics
- ▶ Hoare Logic Rules and examples

Conditional Rule

$$\frac{\{P \wedge S\} C_1 \{Q\} \quad \{P \wedge \neg S\} C_2 \{Q\}}{\{P\} \text{ IF } S \text{ THEN } C_1 \text{ ELSE } C_2 \text{ FI } \{Q\}}$$

► Example ($\text{Max } X \ Y$):

$$\frac{\{X \geq X \wedge X \geq Y\} \text{ MAX := } X \{MAX \geq X \wedge MAX \geq Y\} \\ \{T \wedge X \geq Y\} \text{ MAX := } X \{MAX \geq X \wedge MAX \geq Y\}}{(1)}$$

$$\frac{\{Y \geq X \wedge Y \geq Y\} \text{ MAX := } Y \{MAX \geq X \wedge MAX \geq Y\} \\ \{T \wedge \neg(X \geq Y)\} \text{ MAX := } Y \{MAX \geq X \wedge MAX \geq Y\}}{(2)}$$

$$\frac{(1) \qquad (2)}{\{T\} \text{ IF } X \geq Y \text{ THEN } \text{MAX := } X \text{ ELSE } \text{MAX := } Y \text{ FI } \{MAX \geq X \wedge MAX \geq Y\}} (3)$$

What if?

$$\frac{\{P \wedge S\} C_1 \{Q\} \quad \{P \wedge \neg S\} C_2 \{Q\}}{\{P\} \text{ IF } S \text{ THEN } C_1 \text{ ELSE } C_2 \text{ FI } \{Q\}}$$

► Example (`Max X Y`):

$$\frac{\overline{\{X = \max(X, Y)\} \text{ MAX := } X \{MAX = \max(X, Y)\}}}{\overline{\{T \wedge X \geq Y\} \text{ MAX := } X \{MAX = \max(X, Y)\}}} \quad (4)$$

$$\frac{\overline{\{Y = \max(X, Y)\} \text{ MAX := } Y \{MAX = \max(X, Y)\}}}{\overline{\{T \wedge \neg(X \geq Y)\} \text{ MAX := } Y \{MAX = \max(X, Y)\}}} \quad (5)$$

$$\frac{(4) \quad (5)}{\overline{\{T\} \text{ IF } X \geq Y \text{ THEN } \text{MAX := } X \text{ ELSE } \text{MAX := } Y \text{ FI } \{MAX = \max(X, Y)\}}} \quad (6)$$

Precondition Strengthening

$$\frac{P \longrightarrow P' \quad \{P'\} \subset \{Q\}}{\{P\} \subset \{Q\}}$$

- ▶ Replace a *precondition* with a stronger condition
- ▶ Example:

$$\frac{X = n \longrightarrow X + 1 = n + 1 \quad \overline{\{X + 1 = n + 1\} \ X := X + 1 \ \{X = n + 1\}}}{\{X = n\} \ X := X + 1 \ \{X = n + 1\}}$$

Postcondition Weakening

$$\frac{\{P\} \; C \; \{Q'\} \quad Q' \rightarrow Q}{\{P\} \; C \; \{Q\}}$$

- ▶ Replace a *postcondition* with a weaker condition
- ▶ Example:

$$\frac{\{X = n\} \; X := X + 1 \; \{X = n + 1\} \quad X = n + 1 \longrightarrow X > n}{\{X = n\} \; X := X + 1 \; \{X > n\}}$$

Aha!

$$\frac{P \longrightarrow P' \quad \{P'\} C \{Q\}}{\{P\} C \{Q\}}$$

- ▶ Example (`Max X Y`):

$$\frac{\mathbf{T} \wedge X \geq Y \longrightarrow X = \max(X, Y) \quad \overline{\{X = \max(X, Y)\}} \text{ MAX := } X \{MAX = \max(X, Y)\}}{\{\mathbf{T} \wedge X \geq Y\} \text{ MAX := } X \{MAX = \max(X, Y)\}} \quad (7)$$

$$\frac{\mathbf{T} \wedge \neg(X \geq Y) \longrightarrow Y = \max(X, Y) \quad \overline{\{Y = \max(X, Y)\}} \text{ MAX := } Y \{MAX = \max(X, Y)\}}{\{\mathbf{T} \wedge \neg(X \geq Y)\} \text{ MAX := } Y \{MAX = \max(X, Y)\}} \quad (8)$$

$$\frac{(7) \qquad (8)}{\{\mathbf{T}\} \text{ IF } X \geq Y \text{ THEN MAX := } X \text{ ELSE MAX := } Y \text{ FI } \{MAX = \max(X, Y)\}} \quad (9)$$

Verification Conditions (VCs)

$\{T\} \text{ IF } X \geq Y \text{ THEN } \text{MAX} := X \text{ ELSE } \text{MAX} := Y \text{ FI } \{MAX = max(X, Y)\}$

- ▶ FOL VCs:

$$T \wedge X \geq Y \longrightarrow X = max(X, Y)$$

$$T \wedge \neg(X \geq Y) \longrightarrow Y = max(X, Y)$$

- ▶ Hoare Logic rules can be applied *automatically* to generate VCs
 - ▶ e.g. Isabelle's `vcg` tactic
- ▶ We need to provide proofs for the VCs / *proof obligations*
 - ▶ Reduced to FOL statements
 - ▶ From simple algebraic proofs to reasoning about inductive data types

WHILE Rule

$$\frac{\{P \wedge S\} \ C \ \{P\}}{\{P\} \text{ WHILE } S \text{ DO } C \text{ OD } \{P \wedge \neg S\}}$$

- ▶ P is an *invariant* for C whenever S holds
- ▶ *WHILE rule*: If executing C *once* preserves the truth of P , then executing C *any number of times* also preserves the truth of P
- ▶ If P is an invariant for C when S holds then P is an invariant of the *whole WHILE loop*, i.e. a *loop invariant*

WHILE Rule

$$\frac{\{P \wedge S\} \ C \ \{P\}}{\{P\} \text{ WHILE } S \text{ DO } C \text{ OD } \{P \wedge \neg S\}}$$

$\{Y = 1 \wedge Z = 0\}$
WHILE $Z \neq X$ DO
 $Z := Z + 1$;
 $Y := Y \times Z$

OD

$\{Y = X!\}$

vs.

$\{P\}$
WHILE $Z \neq X$ DO
 $Z := Z + 1$;
 $Y := Y \times Z$

OD

$\{P \wedge \neg Z \neq X\}$

- ▶ What is P?

WHILE Rule - How to find an invariant

$$\frac{\{P \wedge S\} \ C \ \{P\}}{\{P\} \text{ WHILE } S \text{ DO } C \text{ OD } \{P \wedge \neg S\}}$$

- ▶ The invariant P should:
 - ▶ Say what *has been done so far* together with what *remains to be done*
 - ▶ Hold *at each iteration* of the loop.
 - ▶ Give the *desired result* when the loop terminates

WHILE Rule - Invariant VCs

$$\frac{\{P \wedge S\} \ C \ \{P\}}{\{P\} \text{ WHILE } S \text{ DO } C \text{ OD } \{P \wedge \neg S\}}$$

$$\begin{aligned}\{Y = 1 \wedge Z = 0\} \text{ WHILE } Z \neq X \text{ DO } Z := Z + 1; Y := Y \times Z \text{ OD } \{Y = X!\} \\ \{P\} \text{ WHILE } Z \neq X \text{ DO } Z := Z + 1; Y := Y \times Z \text{ OD } \{P \wedge \neg Z \neq X\}\end{aligned}$$

► We need to find an invariant P such that:

- $\{P \wedge Z \neq X\} \ Z := Z + 1; Y := Y \times Z \ \{P\}$ (WHILE rule)
- $Y = 1 \wedge Z = 0 \longrightarrow P$ (precondition strengthening)
- $P \wedge \neg(Z \neq X) \longrightarrow Y = X!$ (postcondition weakening)

WHILE Rule - Loop invariant for factorial

$$\{P \wedge Z \neq X\} \quad Z := Z + 1 ; \quad Y := Y \times Z \quad \{P\}$$

$$Y = 1 \wedge Z = 0 \longrightarrow P$$

$$P \wedge \neg(Z \neq X) \longrightarrow Y = X!$$

► $Y = Z!$

► VCs:

► $\{Y = Z! \wedge Z \neq X\} \quad Z := Z + 1 ; \quad Y := Y \times Z \quad \{Y = Z!\}$

because: $Y = Z! \wedge Z \neq X \longrightarrow Y \times (Z + 1) = (Z + 1)!$ and (10)

► $Y = 1 \wedge Z = 0 \longrightarrow Y = Z!$

because: $0! = 1$

► $Y = Z! \wedge \neg(Z \neq X) \longrightarrow Y = X!$

because: $\neg(Z \neq X) \leftrightarrow Z = X$

$$\frac{\overline{\{Y \times (Z + 1) = (Z + 1)!\}} \quad Z := Z + 1 \quad \{Y \times Z = Z!\} \quad \overline{\{Y \times Z = Z!\}} \quad \overline{\{Y = Z!\}}}{\{Y \times (Z + 1) = (Z + 1)!\} \quad Z := Z + 1 ; \quad Y := Y \times Z \quad \{Y = Z!\}} \quad (10)$$

WHILE Rule - Complete factorial example

$$\begin{aligned}\{Y = 1 \wedge Z = 0\} \\ \{Y = Z!\}\end{aligned}$$

WHILE $Z \neq X$ DO

$$\begin{aligned}\{Y = Z! \wedge Z \neq X\} \\ \{Y \times (Z + 1) = (Z + 1)!\}\end{aligned}$$

$Z := Z + 1$;

$$\{Y \times Z = Z!\}$$

$Y := Y \times Z$

$$\{Y = Z!\}$$

OD

$$\begin{aligned}\{Y = Z! \wedge \neg Z \neq X\} \\ \{Y = X!\}\end{aligned}$$

Another example - Multiplication!

```
I := Y;  
Z := 0;  
WHILE I ≠ 0 DO  
    Z := Z + X ;  
    I := I - 1  
OD
```

$\{Y \geq 0\}$
 $\{0 = (I - I) \times X\}$
 $\{0 = (Y - I) \times X\}$
 $\{\mathbf{Z} = (\mathbf{Y} - \mathbf{I}) \times \mathbf{X}\}$
 $\{\mathbf{Z} = (\mathbf{Y} - \mathbf{I}) \times \mathbf{X} \wedge I \neq 0\}$
 $\{Z + X = (Y - (I - 1)) \times X\}$
 $\{Z = (Y - (I - 1)) \times X\}$
 $\{\mathbf{Z} = (\mathbf{Y} - \mathbf{I}) \times \mathbf{X}\}$
 $\{\mathbf{Z} = (\mathbf{Y} - \mathbf{I}) \times \mathbf{X} \wedge \neg I \neq 0\}$
 $\{\mathbf{Z} = \mathbf{X} \times \mathbf{Y}\}$

Isabelle

```
lemma Multipl: "VARS (z :: int) i
  {0 ≤ y}
  i := y;
  z := 0;
  WHILE i ≠ 0
    INV { z = (y - i) * x }
    DO
      z := z + x;
      i := i - 1
    OD
  {z = x * y}"
apply vcg
apply (auto simp add: algebra_simps)
```

Proof state Auto update Search: 100%

```
proof (prove)
goal (3 subgoals):
1. ⋀z i. 0 ≤ y ⟹ 0 = (y - i) * x
2. ⋀z i. z = (y - i) * x ∧ i ≠ 0 ⟹ z + x = (y - (i - 1)) * x
3. ⋀z i. z = (y - i) * x ∧ ~ i ≠ 0 ⟹ z = x * y
```

Isabelle

```
lemma Multipl: "VARS (z :: int) i
{0 ≤ y}
i := y;
z := 0;
WHILE i ≠ 0
INV { z = (y - i) * x }
DO
z := z + x;
i := i - 1
OD
{z = x * y}"
apply vcg
apply [auto simp add: algebra_simps]
```

Proof state Auto update Update Search: 100%

```
proof (prove)
goal:
No subgoals!
```

Specification and correctness

$\{T\}$

I := Y;

Z := 0;

WHILE I \neq 0 DO

Z := Z + X ;

I := I - 1

OD

$\{Z = X \times Y\}$

$\{Y \geq 0\}$

I := Y;

Z := 0;

WHILE I \neq 0 DO

Z := Z + X ;

I := I - 1

OD

$\{Z = X \times Y\}$

- What is the difference? - *Termination!*

Hoare Logic Rules (it does!)

$$\frac{P \longrightarrow P' \quad \{P'\} \ C \ \{Q\}}{\{P\} \ C \ \{Q\}} \ PS$$

$$\frac{\{P\} \ C \ \{Q'\} \quad Q' \longrightarrow Q}{\{P\} \ C \ \{Q\}} \ PW$$

$$\frac{}{\{Q[E/V]\} \ V := E \ \{Q\}} \ ASSIGN$$

$$\frac{}{\{P\} \ SKIP \ \{P\}} \ SKIP$$

$$\frac{\{P\} \ C_1 \ \{Q\} \quad \{Q\} \ C_2 \ \{R\}}{\{P\} \ C_1 ; \ C_2 \ \{R\}} \ SEQ$$

$$\frac{\{P \wedge S\} \ C_1 \ \{Q\} \quad \{P \wedge \neg S\} \ C_2 \ \{Q\}}{\{P\} \ \text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2 \text{ FI } \{Q\}} \ IF$$

$$\frac{\{P \wedge S\} \ C \ \{P\}}{\{P\} \ WHILE \ S \ DO \ C \ OD \ \{P \wedge \neg S\}} \ WHILE$$

Other topics

$$\{P\} \ C \ \{Q\}$$

- ▶ Weakest preconditions, strongest postconditions

Other topics

$$\{P\} \ C \ \{Q\}$$

- ▶ Meta-theory: Is Hoare logic...
 - ▶ ... *sound*? - Yes! Based on programming language semantics
(but what about more complex languages?)
 - ▶ ... *decidable*? - No! $\{T\} \ C \ \{F\}$ is the halting problem!
 - ▶ ... *complete*? - *Relatively* / only for simple languages

Other topics

$$\{P\} \ C \ \{Q\}$$

- ▶ Automatic Verification Condition Generation (VCG)
- ▶ Automatic generation/inference of loop invariants!
- ▶ More complex languages - e.g. Pointers = Separation logic
- ▶ Functional programming (recursion = induction)

Another example

```
j := 0; R := [ ]; {R = rev(take 0 A)}
```

WHILE j < length A DO { $R = \text{rev}(\text{take } j A) \wedge j < \text{length } A$ }

? ? ?

$\{A[j]\#R = \text{rev}(\text{take } (j + 1) A)\}$

```
R := A[j] # R ; {R = rev(take (j + 1) A)}
```

```
j := j + 1 {R = rev(take j A)}
```

OD { $R = \text{rev}(\text{take } j A) \wedge \neg j < \text{length } A$ }

$\{\mathbf{R} = \mathbf{rev} \mathbf{A}\}$

Summary

- ▶ Precondition strengthening
- ▶ Postcondition weakening
- ▶ Automated generation of *Verification Conditions* (VCs)
- ▶ WHILE rule: *Loop invariants!*
 - ▶ Properties that hold during *while* loops
 - ▶ Loop invariant generation is generally *undecidable*

Recommended reading

Theory:

- ▶ Mike Gordon, *Background Reading on Hoare Logic*,
<http://www.cl.cam.ac.uk/~mjcg/Teaching/2011/Hoare/Notes/Notes.pdf> (pp. 1-27, 37-48)
- ▶ Huth & Ryan, Sections 4.1-4.3 (pp. 256-292)
- ▶ Nipkow & Klein, Section 12.2.1 (pp. 191-199)

Practice:

- ▶ Isabelle's Hoare Logic library: <http://isabelle.in.tum.de/dist/library/HOL/HOL-Hoare>
- ▶ Tutorial exercise