

Computer Supported Modeling and Reasoning

Jan-Georg Smaus

CSMR in Time and Space

Organizational Matters

Instructor: Dr. Jan-Georg Smaus

Lecture: Wednesday 9:15 – 11:00, with break, Gebäude 101,
SR 00-010/14

Labs: Friday 9:15 – 11:00, Gebäude 52, SR 00-005 (Hiwi
room)

Language: English (questions may be asked in other lan-
guages)

History of this Course

In previous years, this course was given by Prof. Dr. David Basin and Dr. Burkhard Wolff.

In WS01/02 and WS02/03, Jan-Georg Smaus was in charge of the labs and maintaining the lecture slides.

As of 2003, David Basin moved to ETH Zürich. Some members of the Software Engineering group have followed him.

Jan is now in the group of Bernhard Nebel. Jan will give this course this Wintersemester and perhaps in subsequent semesters.

Some Former Students of this Course

Konrad Anton, Rafael Baumgartner, Diana Dragojević,
Michael Drescher, Harald Hiss, Steffen Kemmerer, Paul
Hankes Drielsma (now in Zürich), Angelika Kimmig, Florian
Pigorsch, Silvia Richter, Stefan Spinner
Ask them!

The Slides

The slides for this course will be made available at

<http://www.informatik.uni-freiburg.de/~ki/teaching/ws0405/csmr/>.

You might take notes of things written on the blackboard.

The slides are actually an **online course**. They are also available as lecture notes that can be printed out.

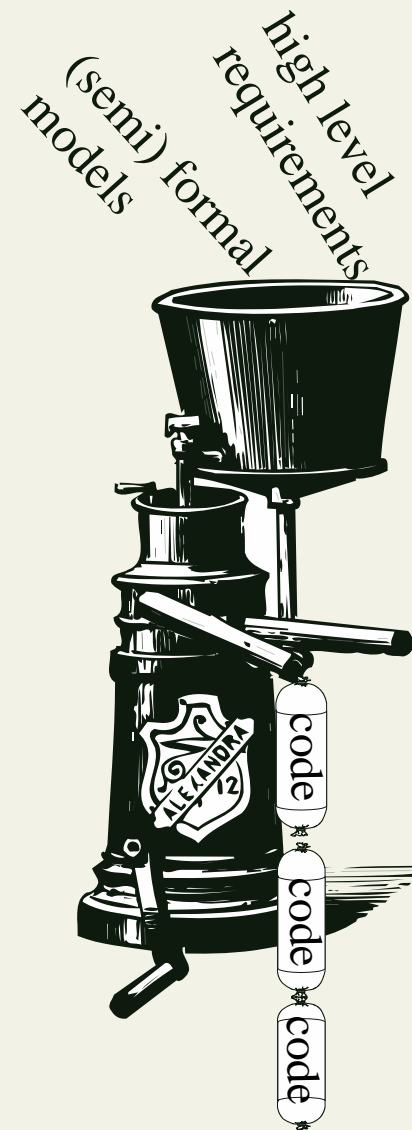
If you note mistakes or have suggestions, tell me!

The slides are around **1300**, contained in a single file. The lecture notes are around **700**, designed for being printed at a rate of four pages per sheet side. So please be mindful of resources when you print!

General Introduction

What this Course is about

- Mechanizing and using logic
 - program **verification**:
 - input**: theories, programs, properties
 - output**: correctness guarantees
 - Hilbert's program:
 - input**: arithmetic
 - output**: theorems
- Technically: **mechanization** and **application**
- Generally: making logic **come to life** and useful as a **general tool**.



Overview: Four Conceptual Parts

1. **Logics** (propositional, first-order, higher-order): appr. 6 units

Overview: Four Conceptual Parts

1. **Logics** (propositional, first-order, higher-order): appr. 6 units
2. **Metalogics** (Isabelle): appr. 2 units

Overview: Four Conceptual Parts

1. **Logics** (propositional, first-order, higher-order): appr. 6 units
2. **Metalogics** (Isabelle): appr. 2 units
3. Modeling mathematics and computer science (programming languages) in higher-order logic: appr. 6 units

Overview: Four Conceptual Parts

1. **Logics** (propositional, first-order, higher-order): appr. 6 units
2. **Metalogics** (Isabelle): appr. 2 units
3. Modeling mathematics and computer science (programming languages) in higher-order logic: appr. 6 units
4. Some case study in formalizing **a theory** (functional or imperative programming, or the specification language Z): appr. 2 units

Presentation roughly follows conceptual structure.

Why this Course Matters

Academic motivation: deepen knowledge of logic and formal reasoning

Why this Course Matters

Academic motivation: deepen knowledge of logic and formal reasoning

Practical motivation: verification and formal methods

- The last decade has seen spectacular hardware and software failures and the birth of a new discipline: the **verification engineer**
- Exciting positions at companies like Intel, Gemplus, . . .

Why this Course Matters (2)

In general:

- Understanding formal reasoning improves understanding of how to build correct systems
- Mechanization provides formal guarantees

Want to see some Isabelle/HOL applications?

Relationship to Other Courses

Logic: deduction, foundations, and applications

Software engineering: specification, refinement, verification

Hardware: formalizing and reasoning about circuit models

Artificial Intelligence: knowledge representation, reasoning, deduction

In general, you will develop a deeper understanding of mathematical and logical reasoning, which is central to computer science.

Requirements

Some knowledge of **logic** is useful for this course.

Requirements

Some knowledge of **logic** is useful for this course.

We will try to accommodate different backgrounds, e.g. with pointers to additional material. Your feedback is essential!

You must be willing to participate in the labs and **get your hands dirty** using a proof development system:

- further develop course material
- present material on **pragmatics** of mechanized reasoning
- hands-on experience.

Requirements

Some knowledge of **logic** is useful for this course.

We will try to accommodate different backgrounds, e.g. with pointers to additional material. Your feedback is essential!

You must be willing to participate in the labs and **get your hands dirty** using a proof development system:

- further develop course material
- present material on **pragmatics** of mechanized reasoning
- hands-on experience.

Experience shows students end up in two **equivalence classes**. It is hard in the beginning but the rewards are large. ►

More Detailed Explanations

What is Verification?

Verification is the process of formally proving that a program has the desired properties. To this end, it is necessary to define a **specification language** in which the desired properties can be formulated, i.e. **specified**. One must define a **semantics** for this language as well as for the program. These semantics must be linked in such a way that it is meaningful to say: “Program X makes formula Φ true”.

What is Hilbert's Program?

In the 1920's, David Hilbert attempted a single rigorous formalization of all of mathematics, named **Hilbert's** program. He was concerned with the following three questions:

1. Is mathematics **complete** in the sense that every statement could be proved or disproved?
2. Is mathematics **consistent** in the sense that no statement could be proved both true and false?
3. Is mathematics **decidable** in the sense that there existed a definite method to determine the truth or falsity of any mathematical statement?

Hilbert believed that the answer to all three questions was 'yes'.

Thanks to the incompleteness theorem of Gödel (1931) and the

undecidability of first-order logic shown by Church and Turing (1936–37) we know now that his dream will never be realized completely. This makes it a never-ending task to find partial answers to Hilbert's questions.

For more details:

- Panel talk by Moshe Vardi
- Lecture by Michael J. O'Donnell
- Article by Stephen G. Simpson
- Original works **Über das Unendliche** and **Die Grundlagen der Mathematik** [vH67]
- Some quotations shedding light on Gödel's incompleteness theorem
- Eric Weisstein's world of mathematics explaining Gödel's incompleteness theorem

Mechanizing Logic

We will learn to make logic **run on a computer** by using the Isabelle system.

What is (a) Logic?

The word **logic** is used in a wider and a narrower sense.

In a wider sense, **logic** is the science of reasoning. In fact, it is the science that reasons about reasoning itself.

In a narrower sense, **a** logic is just a precisely defined language allowing to write down statements, together with a predefined meaning for some of the syntactic entities of this language. Propositional logic, first-order logic, and higher-order logic are three different logics.

What is a Metalogic?

A **metalogic** is a logic that allows us to express properties of another logic.

What is a Theory?

Intuitively, whenever you do computer-supported modeling and reasoning, you have to formalize a tiny portion of the “world”, the portion that your problem lives in. For example, rational numbers may or may not exist in this portion. A **theory** is such a formalization of a tiny portion of the “world”. A theory extends a logic by axioms that describe that portion of the “world”.

Theories will be considered in more detail **later**.

What we Neglect

We will introduce different logics and formal systems (so-called **calculi**) used to deduce formulas in a logic. We will neglect other aspects that are usually treated in classes or textbooks on logic, e.g.:

- semantics (interpretations) of logics; and
- correctness and completeness of calculi.

As an introduction we recommend [vD80].

Equivalence Classes

In this course it makes no sense to follow just a little bit. Our experience is that the committed students learn the material, whereas the others go away empty-handed.

Propositional Logic

Propositional Logic: Overview

- System for formalizing certain **valid patterns of reasoning**
- Expressions built by combining “atomic propositions” using **not**, **if...then...**, **and**, **or**, etc.
- **Validity** means: no counterexample. Validity independent of **content**. Depends on **form** of the expressions \Rightarrow can make patterns explicit by replacing words by symbols

From **if A then B** and **A** it follows that **B**.

Propositional Logic: Overview

- System for formalizing certain **valid patterns of reasoning**
- Expressions built by combining “atomic propositions” using **not**, **if...then...**, **and**, **or**, etc.
- **Validity** means: no counterexample. Validity independent of **content**. Depends on **form** of the expressions \Rightarrow can make patterns explicit by replacing words by symbols

$$\frac{A \rightarrow B \quad A}{B}$$

Propositional Logic: Overview

- System for formalizing certain **valid patterns of reasoning**
- Expressions built by combining “atomic propositions” using **not**, **if...then...**, **and**, **or**, etc.
- **Validity** means: no counterexample. Validity independent of **content**. Depends on **form** of the expressions \Rightarrow can make patterns explicit by replacing words by symbols

$$\frac{A \rightarrow B \quad A}{B}$$

- What about

From if A then B and B it follows that A?

More Examples

1. If it is Sunday, then I don't need to work.
It is Sunday.
Therefore I don't need to work.
2. It will rain or snow.
It will not snow.
Therefore it will rain.
3. The Butler is guilty or the Maid is guilty.
The Maid is guilty or the Cook is guilty.
Therefore either the Butler is guilty or the Cook is guilty.

More Examples (Which are Valid?)

1. If it is Sunday, then I don't need to work.
It is Sunday.
Therefore I don't need to work.
2. It will rain or snow.
It will not snow.
Therefore it will rain.
3. The Butler is guilty or the Maid is guilty.
The Maid is guilty or the Cook is guilty.
Therefore either the Butler is guilty or the Cook is guilty.

History

- Propositional logic was developed to make this all precise.
- Laws for valid reasoning were known to the Stoic philosophers (about 300 BC).
- The formal system is often attributed to George Boole (1815-1864).

More Formal Examples

Formalization allows us to “turn the crank”.

More Formal Examples

Formalization allows us to “turn the crank”.

Phrases like “from . . . it follows” or “therefore” are formalized as **derivation rules**, e.g.

$$\frac{A \rightarrow B \quad A}{B} \rightarrow\text{-}E$$

More Formal Examples

Formalization allows us to “turn the crank”.

Phrases like “from . . . it follows” or “therefore” are formalized as **derivation rules**, e.g.

$$\frac{A \rightarrow B \quad A}{B} \rightarrow\text{-}E$$

Rules are grafted together to build trees called **derivations**.
This defines a **proof system** in the style of **natural deduction**.

Formalizing Propositional Logic

- We must formalize
 - (a) Language
 - (b) Deductive system

Formalizing Propositional Logic

- We must formalize
 - (a) Language
 - (b) Deductive system
- Here we will focus on formalizing the deductive machinery and implicitly assume semantics and metatheorems (soundness and completeness).

Formalizing Propositional Logic

- We must formalize
 - (a) Language
 - (b) Deductive system
- Here we will focus on formalizing the deductive machinery and implicitly assume semantics and metatheorems (soundness and completeness).
- For labs we will carry out proofs using the Isabelle System. Isabelle supports a Natural Deduction deductive system.

Propositional Logic: Language

Propositions are built from a collection of (propositional) variables and closed under disjunction, conjunction, implication, . . .

Propositional Logic: Language (2)

More formally: Let a set V of variables be given. L_P , the language of propositional logic, is the smallest set where:

- X in L_P if X in V .
- \perp in L_P .
- $(A \wedge B)$ in L_P if A in L_P and B in L_P .
- $(A \vee B)$ in L_P if A in L_P and B in L_P .
- $(A \rightarrow B)$ in L_P if A in L_P and B in L_P .
- $(\neg A)$ in L_P if A in L_P .

The elements of L_P are called (propositional) formulas.

We omit unnecessary brackets.

Deductive System: Natural Deduction

Developed by Gentzen [Gen35] and Prawitz [Pra65].

Designed to support ‘natural’ logical arguments:

- we make (temporary) **assumptions**;
- we **derive** new formulas by applying **rules**;
- there is also a mechanism for “getting rid of” assumptions.

Natural Deduction (2)

Derivations are trees

$$\begin{array}{c}
 \frac{A \rightarrow (B \rightarrow C) \quad A}{B \rightarrow C} \rightarrow\text{-}E \\
 \frac{\quad B}{C} \rightarrow\text{-}E
 \end{array}$$

where the leaves are called assumptions.

Natural Deduction (2)

Derivations are trees

$$\frac{\frac{A \rightarrow (B \rightarrow C) \quad A}{B \rightarrow C} \rightarrow\text{-}E \quad B}{C} \rightarrow\text{-}E$$

where the leaves are called **assumptions**.

Write $A_1, \dots, A_n \vdash A$ if there exists a derivation of A with assumptions A_1, \dots, A_n , e.g. $A \rightarrow (B \rightarrow C), A, B \vdash C$.

Natural Deduction (2)

Derivations are trees

$$\frac{\frac{A \rightarrow (B \rightarrow C) \quad A}{B \rightarrow C} \rightarrow\text{-}E \quad B}{C} \rightarrow\text{-}E$$

where the leaves are called **assumptions**.

Write $A_1, \dots, A_n \vdash A$ if there exists a derivation of A with assumptions A_1, \dots, A_n , e.g. $A \rightarrow (B \rightarrow C), A, B \vdash C$.

A **proof** is a derivation where we “got rid” of all assumptions.

Natural Deduction: an Abstract Example

- Language $\mathcal{L} = \{\heartsuit, \clubsuit, \spadesuit, \diamondsuit\}$.

Natural Deduction: an Abstract Example

- Language $\mathcal{L} = \{\heartsuit, \clubsuit, \spadesuit, \diamondsuit\}$.
- Deductive system given by **rules of proof**:

$$\frac{\diamondsuit}{\clubsuit} \alpha \quad \frac{\diamondsuit}{\spadesuit} \beta \quad \frac{\clubsuit \quad \spadesuit}{\heartsuit} \gamma$$

How do you read these rules?

Natural Deduction: an Abstract Example

- Language $\mathcal{L} = \{\heartsuit, \clubsuit, \spadesuit, \diamondsuit\}$.
- Deductive system given by **rules of proof**:

$$\begin{array}{cccc}
 \frac{\diamondsuit}{\clubsuit} \alpha & \frac{\diamondsuit}{\spadesuit} \beta & \frac{\clubsuit \quad \spadesuit}{\heartsuit} \gamma & \frac{[\diamondsuit] \quad \vdots \quad \heartsuit}{\heartsuit} \delta
 \end{array}$$

How about this one?

Natural Deduction: an Abstract Example

- Language $\mathcal{L} = \{\heartsuit, \clubsuit, \spadesuit, \diamondsuit\}$.
- Deductive system given by **rules of proof**:

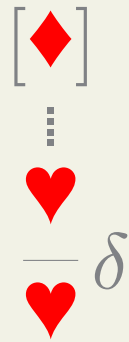
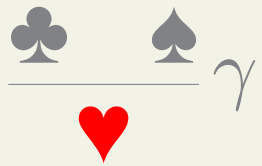
$$\begin{array}{cccc}
 \frac{\diamondsuit}{\clubsuit} \alpha & \frac{\diamondsuit}{\spadesuit} \beta & \frac{\clubsuit \quad \spadesuit}{\heartsuit} \gamma & \frac{[\diamondsuit] \quad \vdots \quad \heartsuit}{\heartsuit} \delta
 \end{array}$$

How about this one?

$\alpha, \beta, \gamma, \delta$ are just **names** for the rules.

Proof of ♥

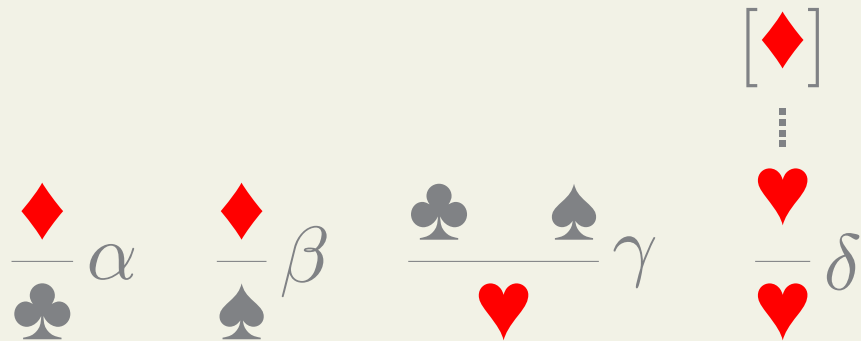
The rules:



The proof:

Proof of ♥

The rules:



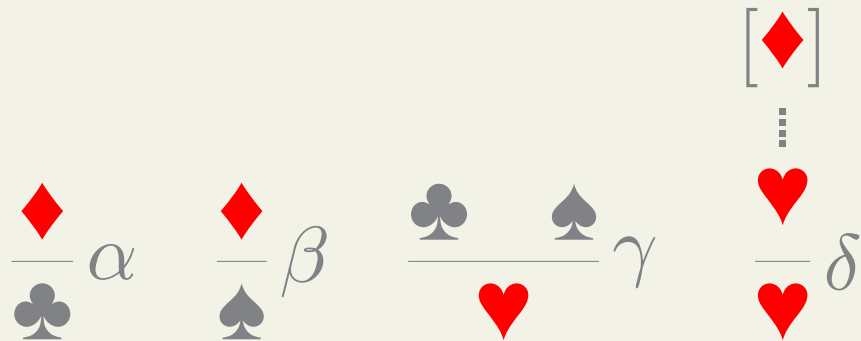
The proof:



We make an assumption. The assumption is now open.

Proof of ♥

The rules:



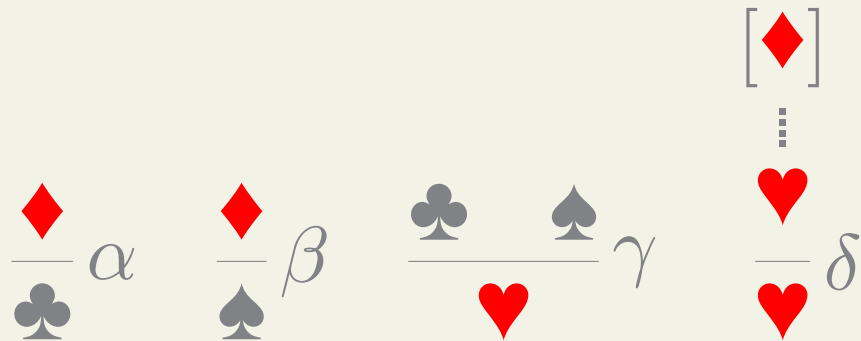
The proof:



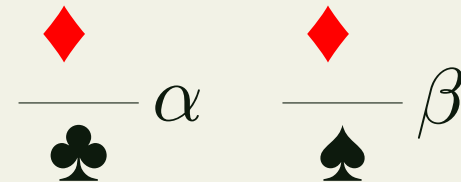
We apply α .

Proof of ♥

The rules:



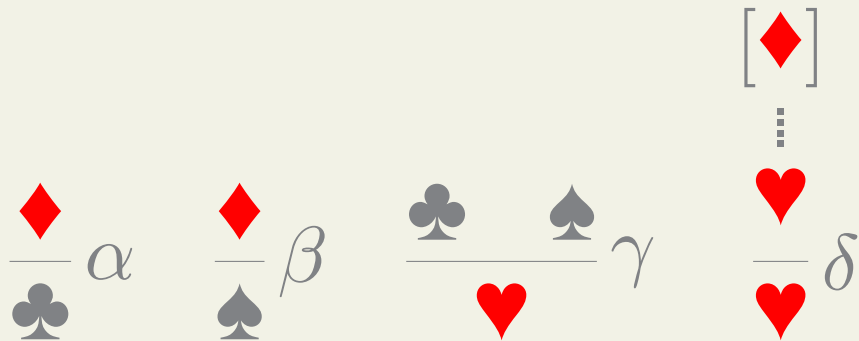
The proof:



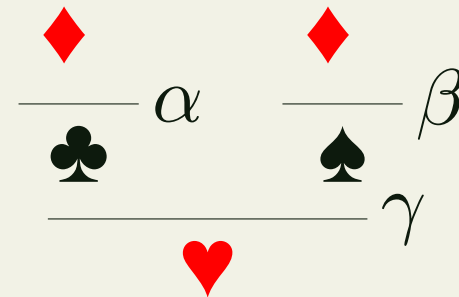
Similarly with β .

Proof of ♥

The rules:



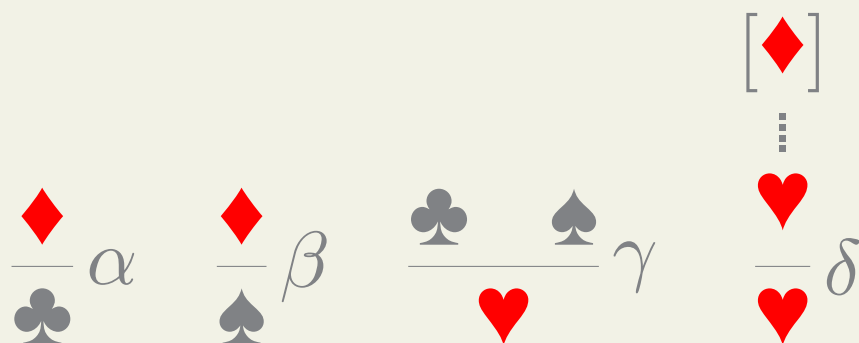
The proof:



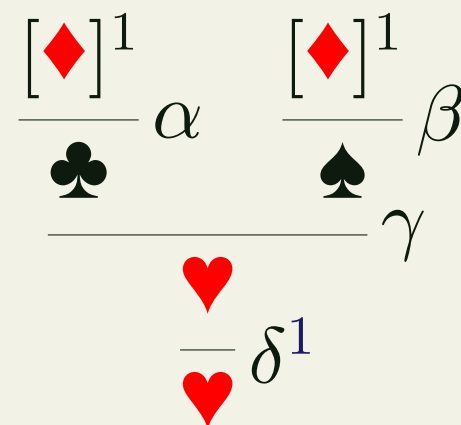
We apply γ .

Proof of ♥

The rules:



The proof:



We apply δ , discharging two occurrences of ♥. We mark the brackets and the rule with a label so that it is clear which assumption is discharged in which step. The derivation is now a **proof**: it has no **open assumptions** (all discharged).

Deductive System: Rules of Propositional Logic

We have rules for conjunction, implication, disjunction, falsity and negation.

Some rules **introduce**, others **eliminate** connectives.

Rules of Propositional Logic: Conjunction

- Rules of two kinds: introduce and eliminate connectives

$$\frac{A \quad B}{A \wedge B} \wedge\text{-}I \quad \frac{A \wedge B}{A} \wedge\text{-}EL \quad \frac{A \wedge B}{B} \wedge\text{-}ER$$

Rules of Propositional Logic: Conjunction

- Rules of two kinds: **introduce** and **eliminate** connectives

$$\frac{A \quad B}{A \wedge B} \wedge\text{-}I \quad \frac{A \wedge B}{A} \wedge\text{-}EL \quad \frac{A \wedge B}{B} \wedge\text{-}ER$$

- Rules are **schematic**.
- Why **valid**? If all assumptions are true, then so is conclusion

$$\mathcal{A} \models A \wedge B \text{ iff } \mathcal{A} \models A \text{ and } \mathcal{A} \models B$$

Example Derivation with Conjunction

The rules:

$$\frac{A \quad B}{A \wedge B} \wedge-I$$

$$\frac{A \wedge B}{A} \wedge-EL$$

$$\frac{A \wedge B}{B} \wedge-ER$$

Example Derivation with Conjunction

The rules:

$$\frac{A \quad B}{A \wedge B} \wedge\text{-}I$$

$$\frac{A \wedge B}{A} \wedge\text{-}EL$$

$$\frac{A \wedge B}{B} \wedge\text{-}ER$$

$$\frac{A \wedge (B \wedge C)}{A} \wedge\text{-}EL$$

Example Derivation with Conjunction

The rules:

$$\frac{A \quad B}{A \wedge B} \wedge\text{-}I$$

$$\frac{A \wedge B}{A} \wedge\text{-}EL$$

$$\frac{A \wedge B}{B} \wedge\text{-}ER$$

$$\frac{A \wedge (B \wedge C)}{A} \wedge\text{-}EL$$

$$\frac{A \wedge (B \wedge C)}{B \wedge C} \wedge\text{-}ER$$

Example Derivation with Conjunction

The rules:

$$\frac{A \quad B}{A \wedge B} \wedge\text{-}I$$

$$\frac{A \wedge B}{A} \wedge\text{-}EL$$

$$\frac{A \wedge B}{B} \wedge\text{-}ER$$

$$\frac{A \wedge (B \wedge C)}{A} \wedge\text{-}EL$$

$$\frac{A \wedge (B \wedge C)}{B \wedge C} \wedge\text{-}ER$$

$$\frac{B \wedge C}{C} \wedge\text{-}ER$$

Example Derivation with Conjunction

The rules:

$$\frac{A \quad B}{A \wedge B} \wedge-I$$

$$\frac{A \wedge B}{A} \wedge-EL$$

$$\frac{A \wedge B}{B} \wedge-ER$$

$$\frac{\frac{A \wedge (B \wedge C)}{A} \wedge-EL \quad \frac{\frac{A \wedge (B \wedge C)}{B \wedge C} \wedge-ER \quad \frac{B \wedge C}{C} \wedge-ER}{A \wedge C} \wedge-I$$

Example Derivation with Conjunction

The rules:

$$\frac{A \quad B}{A \wedge B} \wedge-I$$

$$\frac{A \wedge B}{A} \wedge-EL$$

$$\frac{A \wedge B}{B} \wedge-ER$$

$$\frac{\frac{A \wedge (B \wedge C)}{A} \wedge-EL \quad \frac{\frac{A \wedge (B \wedge C)}{B \wedge C} \wedge-ER \quad \frac{B \wedge C}{C} \wedge-ER}{A \wedge C} \wedge-I$$

Can we **prove** anything with just these three rules?

Rules of Propositional Logic: Implication

- Rules

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B} \rightarrow -I \quad \frac{A \rightarrow B \quad A}{B} \rightarrow -E$$

Rules of Propositional Logic: Implication

- Rules

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B} \rightarrow -I \quad \frac{A \rightarrow B \quad A}{B} \rightarrow -E$$

- $\rightarrow -E$ is also called **modus ponens**.

Rules of Propositional Logic: Implication

- Rules

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B} \rightarrow -I \quad \frac{A \rightarrow B \quad A}{B} \rightarrow -E$$

- $\rightarrow -E$ is also called **modus ponens**.

- $\rightarrow -I$ formalizes strategy:

To derive $A \rightarrow B$, derive B under the additional assumption A .

A very Simple Proof

The simplest proof we can think of is the proof of $P \rightarrow P$.

P

A very Simple Proof

The simplest proof we can think of is the proof of $P \rightarrow P$.

$$\frac{[P]^1}{P \rightarrow P} \rightarrow\text{-I}^1$$

Do you find this strange?

Examples with Conjunction and Implication

1. $A \rightarrow B \rightarrow A$

2. $A \wedge (B \wedge C) \rightarrow A \wedge C$

3. $(A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$

Are these object or metavariables here?

Disjunction

- Rules

$$\frac{A}{A \vee B} \vee\text{-}IL$$

$$\frac{B}{A \vee B} \vee\text{-}IR$$

$$\frac{A \vee B \quad \begin{array}{c} [A] \\ \vdots \\ C \end{array} \quad \begin{array}{c} [B] \\ \vdots \\ C \end{array}}{C} \vee\text{-}E$$

Disjunction

- Rules

$$\frac{A}{A \vee B} \vee\text{-}IL \qquad \frac{B}{A \vee B} \vee\text{-}IR \qquad \frac{A \vee B \quad \begin{array}{c} [A] \\ \vdots \\ C \end{array} \quad \begin{array}{c} [B] \\ \vdots \\ C \end{array}}{C} \vee\text{-}E$$

- Formalizes case-split strategy for using $A \vee B$.

Disjunction: Example

- Rules

$$\begin{array}{c}
 \frac{A}{A \vee B} \vee\text{-}IL \qquad \frac{B}{A \vee B} \vee\text{-}IR \qquad \frac{A \vee B \quad \begin{array}{c} [A] \\ \vdots \\ C \end{array} \quad \begin{array}{c} [B] \\ \vdots \\ C \end{array}}{C} \vee\text{-}E
 \end{array}$$

- Example: formalize and prove

When it rains then I wear my jacket.

When it snows then I wear my jacket.

It is raining or snowing.

Therefore I wear my jacket.

Falsity and Negation

- Falsity

$$\frac{\perp}{A} \perp\text{-}E$$

No introduction rule!

Falsity and Negation

- Falsity

$$\frac{\perp}{A} \perp\text{-}E$$

No introduction rule!

- Negation: define $\neg A$ as $A \rightarrow \perp$. Rules for \neg just special cases of rules for \rightarrow . Convenient to have

$$\frac{\neg A \quad A}{B} \neg\text{-}E \quad \text{derived by} \quad \frac{\frac{\neg A \quad A}{\perp} \rightarrow\text{-}E}{B} \perp\text{-}E$$

Intuitionistic versus Classical Logic

- Peirce's Law: $((A \rightarrow B) \rightarrow A) \rightarrow A$.
Is this **valid**? **Provable**?

Intuitionistic versus Classical Logic

- Peirce's Law: $((A \rightarrow B) \rightarrow A) \rightarrow A$.
Is this **valid**? **Provable**?
- It is provable in **classical logic**, obtained by adding

$$A \vee \neg A \text{ or } \frac{[\neg A] \vdots \perp}{A} RAA \text{ or } \frac{[\neg A] \vdots A}{A} \textit{classical} .$$

Example of Classical Reasoning

There exist irrational numbers a and b such that a^b is rational.

Proof: Let b be $\sqrt{2}$ and consider whether or not b^b is rational.

Case 1: If rational, let $a = b = \sqrt{2}$

Case 2: If irrational, let $a = \sqrt{2}^{\sqrt{2}}$, and then

$$a^b = \sqrt{2}^{\sqrt{2}^{\sqrt{2}}} = \sqrt{2}^{(\sqrt{2} * \sqrt{2})} = \sqrt{2}^2 = 2$$

Overview of Rules

$$\frac{A \quad B}{A \wedge B} \wedge\text{-}I \qquad \frac{A \wedge B}{A} \wedge\text{-}EL \qquad \frac{A \wedge B}{B} \wedge\text{-}ER$$

$$\frac{A}{A \vee B} \vee\text{-}IL \qquad \frac{B}{A \vee B} \vee\text{-}IR \qquad \frac{A \vee B \quad \begin{array}{c} [A] \\ \vdots \\ C \end{array} \quad \begin{array}{c} [B] \\ \vdots \\ C \end{array}}{C} \vee\text{-}E$$

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B} \rightarrow\text{-}I \qquad \frac{A \rightarrow B \quad A}{B} \rightarrow\text{-}E \qquad \frac{\perp}{A} \perp\text{-}E$$

Deductive System: Derived Rules

Using the **basic** rules, we can derive new rules.

Example: Resolution rule.

Deductive System: Derived Rules

Using the **basic** rules, we can derive new rules.

Example: Resolution rule.

$$\frac{R \vee S \quad \neg S}{R}$$

It looks like this.

Deductive System: Derived Rules

Using the **basic** rules, we can derive new rules.

Example: Resolution rule.

$$\neg S$$

$$\frac{R \vee S \quad \neg S}{R}$$

$$\frac{R \vee S}{R}$$

We build a fragment of a derivation by writing the conclusion R and the assumptions $R \vee S$ and $\neg S$.

Deductive System: Derived Rules

Using the **basic** rules, we can derive new rules.

Example: Resolution rule.

$$\neg S$$

$$\frac{R \vee S \quad \neg S}{R} \quad \frac{R \vee S \quad R}{R} \quad \vee\text{-}E$$

Since we have assumption $R \vee S$, using $\vee\text{-}E$ seems a good idea. So we should make assumptions R and S . First R . But that is a derivation of R from R !

Deductive System: Derived Rules

Using the **basic** rules, we can derive new rules.

Example: Resolution rule.

$$\neg S \quad S$$

$$\frac{R \vee S \quad \neg S}{R}$$

$$\frac{R \vee S \quad R}{R} \quad \vee\text{-}E$$

So now S .

Deductive System: Derived Rules

Using the **basic** rules, we can derive new rules.

Example: Resolution rule.

$$\begin{array}{c}
 \frac{\neg S \quad S}{\perp} \rightarrow -E \\
 \\
 \frac{R \vee S \quad \neg S}{R} \quad \frac{R \vee S \quad R}{R} \vee -E
 \end{array}$$

$\neg S$ and S allow us to apply $\rightarrow -E$.

Deductive System: Derived Rules

Using the **basic** rules, we can derive new rules.

Example: Resolution rule.

$$\begin{array}{c}
 \frac{R \vee S \quad \neg S}{R} \\
 \frac{R \vee S \quad R}{R} \\
 \frac{\frac{\frac{\neg S \quad S}{\perp} \rightarrow -E}{R} \perp -E}{R} \vee -E
 \end{array}$$

To apply $\vee -E$ in the end, we need to derive R . But that's easy using $\perp -E$!

Deductive System: Derived Rules

Using the **basic** rules, we can derive new rules.

Example: Resolution rule.

$$\begin{array}{c}
 \frac{R \vee S \quad \neg S}{R} \qquad \frac{R \vee S \quad [R]^1}{R} \qquad \frac{\frac{\frac{\neg S \quad [S]^1}{\perp} \rightarrow -E}{R} \perp -E}{\vee - E^1}
 \end{array}$$

Finally, we can apply $\vee - E$. The derivation with open assumptions is a new rule that can be used like any other rule.

Alternative Deductive System Using Sequent Notation

One can base the deductive system around the **derivability judgement**, i.e., reason about $\Gamma \vdash A$ where $\Gamma \equiv A_1, \dots, A_n$ instead of individual formulae.

Sequent Rules (for \rightarrow / \wedge Fragment)

Rules for assumptions and weakening:

$$\Gamma \vdash A \quad (\text{where } A \in \Gamma) \quad \frac{\Gamma \vdash B}{A, \Gamma \vdash B} \textit{weaken}$$

Sequent Rules (for \rightarrow / \wedge Fragment)

Rules for assumptions and weakening:

$$\Gamma \vdash A \quad (\text{where } A \in \Gamma) \quad \frac{\Gamma \vdash B}{A, \Gamma \vdash B} \text{weaken}$$

Rules for \wedge and \rightarrow :

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge-I \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge-EL \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge-ER$$

$$\frac{A, \Gamma \vdash B}{\Gamma \vdash A \rightarrow B} \rightarrow-I \quad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \rightarrow-E$$

More rules can be derived.

Example: Refinement Style with Metavariables

$$\frac{}{\vdash A \wedge (B \wedge C) \rightarrow A \wedge C} \rightarrow\text{-I}$$

We want to show that $A \wedge (B \wedge C) \rightarrow A \wedge C$ is a tautology, i.e., that it is derivable without any assumptions.

Example: Refinement Style with Metavariables

$$\frac{\frac{}{A \wedge (B \wedge C) \vdash A \wedge C} \wedge-I}{\vdash A \wedge (B \wedge C) \rightarrow A \wedge C} \rightarrow-I$$

The topmost connective of the formula is \rightarrow , so the best rule to choose is $\rightarrow-I$.

Example: Refinement Style with Metavariables

$$\begin{array}{c}
 \frac{}{A \wedge (B \wedge C) \vdash A} \wedge\text{-}EL \qquad \frac{}{A \wedge (B \wedge C) \vdash C} \wedge\text{-}ER \\
 \hline
 \frac{}{A \wedge (B \wedge C) \vdash A \wedge C} \wedge\text{-}I \\
 \hline
 \frac{}{\vdash A \wedge (B \wedge C) \rightarrow A \wedge C} \rightarrow\text{-}I
 \end{array}$$

The topmost connective of the formula is \wedge , so the best rule to choose is $\wedge\text{-}I$.

Example: Refinement Style with Metavariables

$$\begin{array}{c}
 \frac{A \wedge (B \wedge C) \vdash A \wedge \textcolor{red}{?X}}{A \wedge (B \wedge C) \vdash A} \wedge\text{-}EL \qquad \frac{}{A \wedge (B \wedge C) \vdash C} \wedge\text{-}ER \\
 \hline
 \frac{A \wedge (B \wedge C) \vdash A \wedge C}{\vdash A \wedge (B \wedge C) \rightarrow A \wedge C} \wedge\text{-}I \rightarrow\text{-}I
 \end{array}$$

Things are becoming less obvious. To know that $\wedge\text{-}EL$ is the best rule for the r.h.s., you need to inspect the assumption $A \wedge (B \wedge C)$.

Example: Refinement Style with Metavariables

$$\begin{array}{c}
 \frac{A \wedge (B \wedge C) \vdash A \wedge ?X}{A \wedge (B \wedge C) \vdash A} \wedge\text{-}EL \qquad \frac{\frac{\frac{A \wedge (B \wedge C) \vdash (?Y \wedge C)}{A \wedge (B \wedge C) \vdash C} \wedge\text{-}ER}{A \wedge (B \wedge C) \vdash A \wedge C} \wedge\text{-}I}{\vdash A \wedge (B \wedge C) \rightarrow A \wedge C} \rightarrow\text{-}I
 \end{array}$$

Now it's becoming even more difficult. To know that $\wedge\text{-}ER$ is the best rule for the l.h.s., you need to look deep into the assumption $A \wedge (B \wedge C)$.

Example: Refinement Style with Metavariables

$$\begin{array}{c}
 \frac{A \wedge (B \wedge C) \vdash A \wedge \textcolor{red}{?X}}{A \wedge (B \wedge C) \vdash A} \wedge\text{-}EL \qquad \frac{A \wedge (B \wedge C) \vdash \textcolor{red}{?Z} \wedge (\textcolor{red}{?Y} \wedge C)}{A \wedge (B \wedge C) \vdash (\textcolor{red}{?Y} \wedge C)} \wedge\text{-}ER \\
 \frac{\qquad \frac{A \wedge (B \wedge C) \vdash C}{A \wedge (B \wedge C) \vdash A \wedge C} \wedge\text{-}I}{\vdash A \wedge (B \wedge C) \rightarrow A \wedge C} \rightarrow\text{-}I
 \end{array}$$

Again you need to look at both sides of the \vdash to decide what to do.

Example: Refinement Style with Metavariables

$$\begin{array}{c}
 \frac{A \wedge (B \wedge C) \vdash A \wedge ?X}{A \wedge (B \wedge C) \vdash A} \wedge\text{-}EL \qquad \frac{A \wedge (B \wedge C) \vdash ?Z \wedge (?Y \wedge C)}{A \wedge (B \wedge C) \vdash (?Y \wedge C)} \wedge\text{-}ER \\
 \frac{A \wedge (B \wedge C) \vdash (?Y \wedge C)}{A \wedge (B \wedge C) \vdash C} \wedge\text{-}ER \\
 \frac{A \wedge (B \wedge C) \vdash A \wedge C}{\vdash A \wedge (B \wedge C) \rightarrow A \wedge C} \wedge\text{-}I \rightarrow\text{-}I
 \end{array}$$

Solution for $?Z = A$, $?Y = B$ and $?X = (B \wedge C)$.

Comments about Refinement

This crazy way of carrying out proofs is the (standard) Isabelle-way!

- Refinement style means we work from **goals to axioms**
- metavariables used to delay commitments

Isabelle allows **other refinements**/alternatives too (see labs).

Outlook

- Computer Supported Modeling and Reasoning is about turning logic into a useful tool and bringing it to life.
- We will cover:
 - deductive aspects of logic (their proof systems)
 - metatheoretic aspects (their representation)
 - pragmatics (their use), and
 - case studies
- This is an active, hands-on course
 - The labs are as important as (if not more than!) the lectures
 - Individual projects are possible. Individual initiative desired! ►

More Detailed Explanations

What is Validity (of a Pattern of Reasoning)?

A and B are symbols whose meaning is **not** “hard-wired” into propositional logic.

From if A then B and A it follows that B

is **valid** because it is true regardless of what A and B “mean”, and in particular, regardless of whether A and B stand for true or false propositions.

An Invalid Pattern

From if A then B and B it follows that A

is invalid because there is a counterexample:

Let A be “Kim is a man” and B be “Kim is a person”.

More Examples

1. If it is Sunday, then I don't need to work.
It is Sunday.
Therefore I don't need to work. VALID
2. It will rain or snow.
It is too warm for snow.
Therefore it will rain. VALID
3. The Butler is guilty or the Maid is guilty.
The Maid is guilty or the Cook is guilty.
Therefore either the Butler is guilty or the Cook is guilty. NOT VALID

More Examples (Which are Valid?)

1. If it is Sunday, then I don't need to work.
It is Sunday.
Therefore I don't need to work. VALID
2. It will rain or snow.
It is too warm for snow.
Therefore it will rain. VALID
3. The Butler is guilty or the Maid is guilty.
The Maid is guilty or the Cook is guilty.
Therefore either the Butler is guilty or the Cook is guilty. NOT VALID

Turning the Crank

By formalizing patterns of reasoning, we make it possible for such reasoning to be checked or even carried out by a computer.

From known patterns of reasoning new patterns of reasoning can be constructed.

What does Formalization Mean?

At this stage, we are content with a formalization that builds on geometrical notions like “above” or “to the right of”. In other words, our formalization consists of geometrical objects like trees.

We study formalization in more detail [later](#).

Proof Systems

A **proof system** or **deductive system** is characterized by a particular set of rules plus the general principles of how rules are grafted together to trees in natural deduction. We will see this shortly, but note that natural deduction is just one style of proof systems.

We call the rules in that particular set **basic** rules. Later we will see one can also **derive** rules.

Soundness and Completeness

A proof system is **sound** if only **valid** propositions can be derived in it.

A proof system is **complete** if all **valid** propositions can be derived in it.

What is a Metatheorem?

A metatheorem is a theorem **about** a proof system, as opposed to a theorem derived within the proof system. The statement “proof system XYZ is sound” is a metatheorem.

What is a Language?

By **language** we mean the language of formulae. We can also say that we define the (object) logic. Here “logic” is used in the **narrower sense**.

What is Semantics?

As we mentioned *earlier*, we will not say much about semantics in this lecture.

The standard semantics for propositional logic builds on the semantic domain of truth values $\{True, False\}$, i.e., each formula is *interpreted* as either *True* or *False*. To interpret a formula, we need an *assignment* of the propositional variables. Each variable must be assigned one of the values *True* or *False*. For each syntactic construct of propositional logic, it is specified how it must be interpreted as a function from truth values to truth values.

We go into more detail *later*.

Two formulae are *equivalent* if they yield the same truth value for any assignment of the propositional variables.

What does **Open** Mean?

For example, all assumptions in

$$\frac{A \rightarrow (B \rightarrow C) \quad A}{B \rightarrow C} \rightarrow\text{-}E \quad \frac{B \rightarrow C \quad B}{C} \rightarrow\text{-}E$$

are open. For the moment, it suffices to know that when an assumption is made, it is initially an **open** assumption.

What is \vdash ?

For the moment, the way to understand it is as follows: by writing $A \rightarrow (B \rightarrow C), A, B \vdash C$, we assert that C can be derived in this proof system under the assumptions $A \rightarrow (B \rightarrow C), A, B$.

We will say more about the \vdash notation [later](#).




Why is this Example Abstract?




Natural deduction is not just about propositional logic! We explain here the **general principles** of natural deduction, not just the application to propositional logic.

In order to emphasize that applying natural deduction is a completely mechanical process, we give an example that is void of any intuition.

It is important that you understand this process. Applying rules mechanically is one thing. Understanding why this process is semantically justified is another.

How to Read these Rules

The first rule reads: if at some root of a tree in the forest you have constructed so far, there is a , then you are allowed to draw a line underneath that  and write  underneath that line.

The third rule reads: if the forest you have constructed so far contains two neighboring trees, where the left tree has root  and the right tree has root , then you are allowed to draw a line underneath those two roots and write  underneath that line.

How to Read these Rules (2)

The last rule reads: if at some root of a tree in the forest you have constructed so far, there is a ♥, then you are allowed to draw a line underneath that ♥ and write ♦ underneath that line. Moreover you are allowed to **discharge** (eliminate, close) 0 or more occurrences of ♦ at the leaves of the tree.

Discharging is marked by writing [] around the discharged formula.

Note that generally, the tree may contain assumptions other than ♦ at the leaves. However, these must not be discharged in this rule application. They will remain open until they might be discharged by some other rule application later.

Making Assumptions

In everyday language, “making an assumption” has a connotation of “claiming”. This is not the case here. By making an assumption, we are not claiming anything.

When interpreting a derivation tree, we must always consider the open assumptions. We must say: under the assumptions . . . , we derived It is thus unproblematic to “make” assumptions.

Propositional Variables

In mathematics, logic and computer science, there are various notions of **variable**. In propositional logic, a variable stands for a **proposition**, i.e., a variable can be interpreted as *True* or *False*.

This will be different in logics that we will learn about **later**.

What is a Formula?

In logic, the word “formula” has a specific meaning. **Formulae** are a syntactic category, namely the expressions that stand for a statement. So formulas are syntactic expressions that are interpreted (on the semantic level) as *True* or *False*.

We will **later** learn about another syntactic category, that of **terms**.

Associativity and Precedences

To save brackets, we use standard associativity and precedences. All **binary connectives** are right-associative:

$$A \circ B \circ C \equiv A \circ (B \circ C)$$

The precedences are \neg before \wedge before \vee before \rightarrow . So for example

$$A \rightarrow B \wedge \neg C \vee D \equiv A \rightarrow ((B \wedge (\neg C)) \vee D)$$

The Language of Propositional Logic

Strictly speaking, the definition of L_P depends on V . A different choice of variables leads to a different language of propositional logic, and so we should not speak of **the** language of propositional logic, but rather of **a** language of propositional logic. However, for propositional logic, one usually does not care much about the names of the variables, or about the fact that their number could be insufficient to write down a certain formula of interest. We usually assume that there are countably infinitely many variables.

Later, we will be more fussy about this point.

Why Smallest Set?

The language of propositional logic is a **set** of formulae, defined by **induction**. Note the following points about the definition, which are important characteristics of any inductive definition:

- By the second item in the definition, L_P is non-empty (also, one would usually have that V is non-empty, since otherwise L_P is not very interesting);
- L_P is required to be the **smallest** set meeting the above conditions. Otherwise, **anything** (a number, a dog, the pope) could be a propositional formula.
- All conditions (or rules) defining L_P have the form: if ψ_1 and \dots and ψ_n are in L_P , then some formula built from ψ_1 and \dots and ψ_n is in L_P .

It is crucial that no negation is involved here. If for example, there was a rule stating: if A is in L_P then A is **not** in L_P , then there could be no L_P fulfilling such a rule.

More detail on inductive definitions can be found in an article by Aczel [[Acz77](#)].

Introduction and Elimination

It is typical that the **basic** rules of a proof system can be classified as introduction or elimination rules for a particular connective.

This classification provides obvious names for the rules and may guide the search for proofs.

The rules for conjunction are pronounced **and-introduction**, **and-elimination-left**, and **and-elimination-right**.

Apart from the **basic** rules, we will later see that there are also **derived** rules.

Validity Revisited

A rule is **valid** if for any **assignment** under which the assumptions of the formula are true, the conclusion is true as well.

The notation $\mathcal{A} \models A \wedge B$ stands for: under the assignment \mathcal{A} , the formula $A \wedge B$ is interpreted as *True*.

This is consistent with the **earlier intuitive explanation** of validity of a formula. Details can be found in any textbook on logic [vD80].

Note that while the notation $\mathcal{A} \models \dots$ will be used again **later**, there \mathcal{A} will not stand for an assignment, but rather for a construct having an assignment as one constituent. This is because we will generalize, and in the new setting we need something more complex than just an assignment. But in spirit $\mathcal{A} \models \dots$ will still mean the same thing.

Schematic Rules

The letters A and B in the rules are not propositional variables. Instead, they can stand for arbitrary propositional formulas. One can also say that A and B are **metavariables**, i.e., they are variables of the proof system as opposed to **object variables**, i.e., variables of the language that we reason about (here: propositional logic).

When a rule is applied, the metavariables of it must be replaced with actual formulae. We say that a rule is being **instantiated**.

We will see more about the use of metavariables **later**.

Can we **Prove** Anything . . . ?

All three rules have a non-empty sequence of assumptions. Thus to build a tree using these rules, we must first make some assumptions.

None of the rules involves **discharging** an assumption.

We have said **earlier** that a **proof** is a derivation with no open assumptions.

Consequently, the answer is **no**.

Object vs. Meta

In these examples, you may regard A, B, C as propositional variables. On the other hand, the proofs are schematic, i.e., they go through for any formula replacing A, B , and C .

So you Find this Strange!

When we make the assumption P , we obtain a **forest** consisting of one tree. In this tree, P is at the same time a leaf and the root. Thus the tree P is a degenerate example of the schema

$$\begin{array}{c} [A] \\ \vdots \\ B \end{array}$$

where both A and B are replaced with P .

Therefore we may apply rule $\rightarrow -I$, similarly as in our **abstract example**.

$$A \rightarrow B \rightarrow A$$

The rule(s):

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B} \rightarrow -I$$

The proof:

$$\frac{\frac{[A]^1}{B \rightarrow A} \rightarrow -I}{A \rightarrow B \rightarrow A} \rightarrow -I^1$$

$$(A \wedge (B \wedge C)) \rightarrow (A \wedge C)$$

The rules:

$$\frac{A \quad B}{A \wedge B} \wedge\text{-}I$$

$$\frac{A \wedge B}{A} \wedge\text{-}EL$$

$$\frac{A \wedge B}{B} \wedge\text{-}ER$$

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B} \rightarrow\text{-}I$$

The proof:

$$\frac{\frac{\frac{[A \wedge (B \wedge C)]^1}{A} \wedge\text{-}EL \quad \frac{\frac{\frac{[A \wedge (B \wedge C)]^1}{B \wedge C} \wedge\text{-}ER}{C} \wedge\text{-}ER}{A \wedge C} \wedge\text{-}I}{(A \wedge (B \wedge C)) \rightarrow (A \wedge C)} \rightarrow\text{-}I^1$$

$$(A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$$

The rules:

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B} \rightarrow -I$$

$$\frac{A \rightarrow B \quad A}{B} \rightarrow -E$$

The proof:

$$\frac{\frac{[(A \rightarrow B \rightarrow C)]^1 \quad [A]^3}{B \rightarrow C} \rightarrow -E \quad \frac{[(A \rightarrow B)]^2 \quad [A]^3}{B} \rightarrow -E}{C} \rightarrow -E$$

$$\frac{C}{A \rightarrow C} \rightarrow -I^3$$

$$\frac{(A \rightarrow B) \rightarrow A \rightarrow C \rightarrow -I^2}{(A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C} \rightarrow -I^1$$

Falsity

The symbol \perp stands for “false”.

No Introduction Rule for \perp

The symbol \perp stands for “false”.

It should be intuitively clear that since the purpose of a proof system is to derive **true** formulae, there is no introduction rule for falsity. One may wonder: what is the role of \perp then? We will see this soon. The main role is linked to negation. We quote from [And02, p. 152]:

\perp plays the role of a contradiction in indirect proofs.

Connectives

The **connectives** are called **conjunction** (\wedge), **disjunction** (\vee), **implication** (\rightarrow) and **negation** (\neg).

The connectives $\wedge, \vee, \rightarrow$ are **binary** since they connect two formulas, the connective \neg is **unary** (most of the time, one only uses the word **connective** for binary connective).

Negation

“Officially”, negation does not exist in our language and proof system. Negation is only used as a **shorthand**, or **syntactic sugar**, for reasons of convenience. In paper-and-pencil proofs, we are allowed to erase any occurrence of $\neg P$ and replace it with $P \rightarrow \perp$, or vice versa, at any time. However, we shall see that when proofs are automated, this process must be made explicit.

The Rules for \neg

The rule

$$\frac{\neg A \quad A}{\perp}$$

is simply an instance of $\rightarrow\text{-}E$ (since $\neg A$ is shorthand for $A \rightarrow \perp$).

Likewise, the rule

$$\frac{\begin{array}{c} [A] \\ \vdots \\ \perp \end{array}}{\neg A}$$

is simply an instance of $\rightarrow\text{-}I$. Therefore, we will not introduce these as special rules. But there is a special rule $\neg\text{-}E$.

The Rule \neg -E

For negation, it is common to have a rule

$$\frac{\neg A \quad A}{B} \neg\text{-}E$$

We have seen how this rule can be derived. The concept of deriving rules will be explained more systematically [later](#).

This rule is also called **ex falso quod libet** (from the false whatever you like).

Peirce's Law Valid?

Yes, simply check the truth table:

A	B	$((A \rightarrow B) \rightarrow A) \rightarrow A$
<i>True</i>	<i>True</i>	<i>True</i>
<i>True</i>	<i>False</i>	<i>True</i>
<i>False</i>	<i>True</i>	<i>True</i>
<i>False</i>	<i>False</i>	<i>True</i>

Peirce's Law Provable?

In the proof system given so far, this is not provable. To prove that it is not provable requires an analysis of so-called normal forms of proofs. However, we do not do this here.

Intuitionistic versus Classical Logic

The proof system we have given so far is a proof system for **intuitionistic logic**. The main point about intuitionistic logic is that one cannot claim that every statement is either true or false, but rather, evidence must be given for every statement.

In classical reasoning, the law of the excluded middle holds.

One also says that proofs in intuitionistic logic are **constructive** whereas proofs in classical logic are not necessarily constructive.

We quote the first sentence from [Min00]:

Intuitionistic logic is studied here as part of familiar classical logic which allows an effective interpretation and mechanical extraction of programs from proofs.

The difference between intuitionistic and classical logic has been the

topic of a fundamental discourse in the literature on logic [PM68]. Often proofs contain case distinctions, assuming that for any statement ψ , either ψ or $\neg\psi$ holds. This reasoning is classical; it does not apply in intuitionistic logic.

Axiom of the Excluded Middle

$A \vee \neg A$ is called axiom of the excluded middle.

Reductio ad absurdum

The rule

$$\frac{[\neg A] \quad \vdots \quad \perp}{A} RAA$$

is called **reduction ad absurdum**.

The classical rule in Isabelle

The rule

$$\frac{[\neg A] \quad \vdots \quad A}{A} \text{classical}$$

corresponds to the formulation in Isabelle.

Sequent Notation

An object like $A \rightarrow (B \rightarrow C), A, B \vdash C$ is called a **derivability judgement**. We explained it **earlier** as simply asserting the fact that there exists a derivation tree with C at its root and open assumptions $A \rightarrow (B \rightarrow C), A, B$.

However, it is also possible to make such judgements the central objects of the deductive system, i.e., have rules involving such objects.

The notation $\Gamma \vdash A$ is called **sequent notation**. However, this should not be confused with the **sequent calculus** (we will consider it **later**). The sequent calculus is based on **sequents**, which are syntactic entities of the form $A_1, \dots, A_n \vdash B_1, \dots, B_m$, where the $A_1, \dots, A_n, B_1, \dots, B_m$ are all formulae. You see that this definition is more general than the derivability judgements we consider here.

What we are about to present is a kind of hybrid between natural

deduction and the sequent calculus, which we might call **natural deduction using a sequent notation**.

Axioms vs. Rules

An **axiom** is a rule without premises. We call a rule with premises **proper**. One can write an axiom A as

$$\overline{A}$$

to emphasise that it is a rule with an empty set of premises.

Note that the **natural deduction rules** for propositional logic contain no axioms. In the **sequent style** formalization, having the assumption rule (axiom) is essential for being able to prove anything, but in the natural deduction style we learned first, we can construct proofs **without having any axioms**.

Assumptions

The special rule for assumptions takes the role in this [sequent style](#) notation that the process of making and discharging assumptions had in [natural deduction based on trees](#).

It is not so obvious that the two ways of writing proofs are equivalent, but we shall become familiar with this in the exercises by doing proofs on paper as well as in Isabelle.

Weakening

The rule *weaken* is

$$\frac{\Gamma \vdash B}{A, \Gamma \vdash B} \text{weaken}$$

Intuitively, the soundness of rule *weaken* should be clear: having an additional assumption in the context cannot hurt since there is no proof rule that requires the **absence** of some assumption.

We will see an application of that rule **later**.

Deriving \wedge -E

As an example, consider

$$\frac{A, B, \Gamma \vdash C \quad \Gamma \vdash A \wedge B}{\Gamma \vdash C} \wedge\text{-E}$$

This rule can be derived as follows:

$$\frac{\frac{\frac{A, B, \Gamma \vdash C}{A, \Gamma \vdash B \rightarrow C} \rightarrow\text{-I} \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge\text{-EL} \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge\text{-ER}}{\frac{\Gamma \vdash A \rightarrow B \rightarrow C}{\Gamma \vdash B \rightarrow C} \rightarrow\text{-I} \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge\text{-ER}} \rightarrow\text{-E} \quad \frac{\Gamma \vdash B \rightarrow C \quad \Gamma \vdash A \wedge B}{\Gamma \vdash C} \rightarrow\text{-E}$$

Which Rule to Choose?

In general, statements about which rule to choose when building a proof are **heuristics**, i.e., they are not guaranteed to work. Building a proof means **searching** for a proof. However, there are situations where the choice is clear. E.g., when the topmost connective of a formula is \rightarrow , then \rightarrow -I is usually the right rule to apply.

The question will be addressed more systematically **later**.

Goals to Axioms

As you saw in our animation, we worked from the root of the tree to the leaves.

Working on Assumptions

One aspect you might have noted in the proof is that the steps at the top, where \wedge -*EL* and \wedge -*ER* were used, required non-obvious choices, and those choices were based on the assumptions in the current derivability judgement.

In Isabelle, we will apply other rules and proof techniques that allow us to manipulate assumptions explicitly. These techniques make the process of finding a proof more deterministic.

But that is just one aspect. We will give a more theoretic account of the way Isabelle constructs proofs [later](#).

Natural Deduction: Review

Overview

- Short review: ND Systems and proofs
- First-Order Logic
 - Overview
 - Syntax
 - Semantics
 - Deduction, some derived rules, and examples

How Are ND Proofs Built?

ND proofs build derivations under (possibly temporary) assumptions.

ND: Example for \rightarrow / \wedge Fragment

Rules:

$$\begin{array}{c}
 \frac{A \quad B}{A \wedge B} \wedge\text{-I} \quad \frac{A \wedge B}{A} \wedge\text{-EL} \\
 \\
 \frac{A \wedge B}{B} \wedge\text{-ER} \quad \frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B} \rightarrow\text{-I} \\
 \\
 \frac{A \rightarrow B \quad A}{B} \rightarrow\text{-E}
 \end{array}$$

Proof:

$$\begin{array}{c}
 \frac{[A \wedge B]^1}{B} \wedge\text{-EL} \quad \frac{[A \wedge B]^1}{A} \wedge\text{-ER} \\
 \hline
 \frac{B \quad A}{B \wedge A} \wedge\text{-I} \\
 \hline
 \frac{A \wedge B \rightarrow B \wedge A}{A \wedge B \rightarrow B \wedge A} \rightarrow\text{-I}^1
 \end{array}$$

Alternative Formalization Using Sequents

Rules (for \rightarrow / \wedge fragment). Here, Γ is a set of formulae.

$$\Gamma \vdash A \quad (\text{where } A \in \Gamma)$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge-I \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge-EL \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge-ER$$

$$\frac{A, \Gamma \vdash B}{\Gamma \vdash A \rightarrow B} \rightarrow-I \quad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \rightarrow-E$$

Two representations equivalent. Sequent notation seems simpler in practice.

Example: Refinement Style with Metavariables

$$\begin{array}{c}
 \frac{A \wedge (B \wedge C) \vdash A \wedge \textcolor{red}{?X}}{A \wedge (B \wedge C) \vdash A} \qquad \frac{A \wedge (B \wedge C) \vdash \textcolor{red}{?Z} \wedge (\textcolor{red}{?Y} \wedge C)}{A \wedge (B \wedge C) \vdash (\textcolor{red}{?Y} \wedge C)} \\
 \hline
 \frac{A \wedge (B \wedge C) \vdash A \qquad A \wedge (B \wedge C) \vdash C}{A \wedge (B \wedge C) \vdash A \wedge C} \\
 \hline
 \vdash A \wedge (B \wedge C) \rightarrow A \wedge C
 \end{array}$$

Solution for $\textcolor{red}{?Z} = A$, $\textcolor{red}{?Y} = B$ and $\textcolor{red}{?X} = (B \wedge C)$.
 We went through this example in detail [last lecture](#).

Comments about Refinement

This crazy way of carrying out proofs is the (standard) Isabelle-way!

- Refinement style means we work from **goals to axioms**
- Metavariables used to delay commitments

Isabelle allows **other refinements**/alternatives too (see labs).



More Detailed Explanations

What are ND Systems and Proofs?

ND stands for **Natural Deduction**. It was explained in the previous lecture.

What is Sequent Notation ?

The judgement $(\Gamma \vdash \phi)$ means that we can derive ϕ from the assumptions in Γ using certain rules. As, explained [in the previous lecture](#), one can make such judgements the central objects of the deductive system.

Sequent Notation and Isabelle

In particular, the sequent style notation is more amenable to automation, and thus it is closer to what happens in Isabelle.

First-Order Logic

First-Order Logic: Overview

In propositional logic, formulae are **Boolean** combinations of **propositions**. A proposition is just a letter (**variable**).

First-Order Logic: Overview

In propositional logic, formulae are **Boolean** combinations of **propositions**. A proposition is just a letter (**variable**).

Can be used to model certain **finite** scenarios. E.g., we can model 10 time units with variables x_1, \dots, x_{10} . Then $x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4 \wedge x_5 \dots$ expresses “**alternating state**”.

First-Order Logic: Overview

In propositional logic, formulae are **Boolean** combinations of **propositions**. A proposition is just a letter (**variable**).

Can be used to model certain **finite** scenarios. E.g., we can model 10 time units with variables x_1, \dots, x_{10} . Then

$x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4 \wedge x_5 \dots$ expresses “**alternating state**”.

Cannot talk about **relations and functions**.

Cannot say something like “the state alternates over all time”.

First-Order Logic: Overview

In propositional logic, formulae are **Boolean** combinations of **propositions**. A proposition is just a letter (**variable**).

Can be used to model certain **finite** scenarios. E.g., we can model 10 time units with variables x_1, \dots, x_{10} . Then

$x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4 \wedge x_5 \dots$ expresses “**alternating state**”.

Cannot talk about **relations and functions**.

Cannot say something like “the state alternates over all time”.

Let us now extend propositional logic to first-order logic.

Variables: Intuition

In first-order logic, we talk about “things” that have certain “properties”.

A **variable** in first-order logic stands for a “thing”.

Variables: Intuition

In first-order logic, we talk about “things” that have certain “properties”.

A **variable** in first-order logic stands for a “thing”.

This is in contrast to **propositional logic**.

It is common to use letters x , y , z for variables.

Predicates: Intuition

A **predicate** denotes a property/relation.

$p(x) \equiv x$ is a prime number $d(x, y) \equiv x$ is divisible by y

Predicates: Intuition

A **predicate** denotes a property/relation.

$p(x) \equiv x$ is a prime number $d(x, y) \equiv x$ is divisible by y

Propositional connectives are used to build statements

- x is a prime and y or z is divisible by x

Predicates: Intuition

A **predicate** denotes a property/relation.

$p(x) \equiv x$ is a prime number $d(x, y) \equiv x$ is divisible by y

Propositional connectives are used to build statements

- x is a prime and y or z is divisible by x

$$p(x) \wedge (d(y, x) \vee d(z, x))$$

Predicates: Intuition

A **predicate** denotes a property/relation.

$p(x) \equiv x$ is a prime number $d(x, y) \equiv x$ is divisible by y

Propositional connectives are used to build statements

- x is a prime and y or z is divisible by x

$$p(x) \wedge (d(y, x) \vee d(z, x))$$

- x is a man and y is a woman and x loves y but not vice versa

Predicates: Intuition

A **predicate** denotes a property/relation.

$p(x) \equiv x$ is a prime number $d(x, y) \equiv x$ is divisible by y

Propositional connectives are used to build statements

- x is a prime and y or z is divisible by x

$$p(x) \wedge (d(y, x) \vee d(z, x))$$

- x is a man and y is a woman and x loves y but not vice versa

$$m(x) \wedge w(y) \wedge l(x, y) \wedge \neg l(y, x)$$

Predicates: Intuition (2)

We can represent only “abstractions” of these in propositional logic, e.g., $p \wedge (d_1 \vee d_2)$ could be an abstraction of $p(x) \wedge (d(y, x) \vee d(z, x))$.

Here p stands for “ x is a prime” and d_1 stands for “ y is divisible by x ”.

Functions: Intuition

- A **constant** stands for a “fixed thing” in a domain.

Functions: Intuition

- A **constant** stands for a “fixed thing” in a domain.
- More generally, a **function** of **arity** n expresses an n -ary operation over some domain, e.g.

Function arity expresses . . .

0

s

+

Functions: Intuition

- A **constant** stands for a “fixed thing” in a domain.
- More generally, a **function** of **arity** n expresses an n -ary operation over some domain, e.g.

Function	arity	expresses . . .
----------	-------	-----------------

0	nullary	
---	---------	--

s	unary	
-----	-------	--

$+$	binary	
-----	--------	--

Functions: Intuition

- A **constant** stands for a “fixed thing” in a domain.
- More generally, a **function** of **arity** n expresses an n -ary operation over some **domain**, e.g.

Function	arity	expresses . . .
----------	-------	-----------------

0	nullary	number “0”
---	---------	------------

s	unary	successor in \mathbb{N}
-----	-------	---------------------------

$+$	binary	function plus in \mathbb{N}
-----	--------	-------------------------------

Note special **notations**: **infix**, **prefix**, etc.

Quantifiers: Intuition

- A variable stands for “some thing” in a domain of discourse. Quantifiers \forall, \exists are used to speak about all or some members of this domain.

Quantifiers: Intuition

- A variable stands for “some thing” in a domain of discourse. Quantifiers \forall, \exists are used to speak about all or some members of this domain.
- Examples: Are they satisfiable? valid?

$$\forall x. \exists y. y * 2 = x$$

Quantifiers: Intuition

- A variable stands for “some thing” in a domain of discourse. Quantifiers \forall, \exists are used to speak about all or some members of this domain.
- Examples: Are they satisfiable? valid?

$$\forall x. \exists y. y * 2 = x \quad \text{true for rationals}$$

Quantifiers: Intuition

- A variable stands for “some thing” in a domain of discourse. Quantifiers \forall, \exists are used to speak about **all** or **some** members of this domain.
- Examples: Are they **satisfiable?** **valid?**

$$\forall x. \exists y. y * 2 = x \quad \text{true for rationals}$$

$$x < y \rightarrow \exists z. x < z \wedge z < y$$

Quantifiers: Intuition

- A variable stands for “some thing” in a domain of discourse. Quantifiers \forall, \exists are used to speak about all or some members of this domain.
- Examples: Are they satisfiable? valid?

$$\forall x. \exists y. y * 2 = x \quad \text{true for rationals}$$

$$x < y \rightarrow \exists z. x < z \wedge z < y \quad \text{true for any dense order}$$

Quantifiers: Intuition

- A variable stands for “some thing” in a domain of discourse. Quantifiers \forall, \exists are used to speak about all or some members of this domain.
- Examples: Are they satisfiable? valid?

$$\forall x. \exists y. y * 2 = x \quad \text{true for rationals}$$

$$x < y \rightarrow \exists z. x < z \wedge z < y \quad \text{true for any dense order}$$

$$\exists x. x \neq 0$$

Quantifiers: Intuition

- A variable stands for “some thing” in a domain of discourse. Quantifiers \forall, \exists are used to speak about all or some members of this domain.
- Examples: Are they satisfiable? valid?

$\forall x. \exists y. y * 2 = x$ true for rationals

$x < y \rightarrow \exists z. x < z \wedge z < y$ true for any dense order

$\exists x. x \neq 0$ true for domains with more than one element

Quantifiers: Intuition

- A variable stands for “some thing” in a domain of discourse. Quantifiers \forall, \exists are used to speak about all or some members of this domain.
- Examples: Are they satisfiable? valid?

$\forall x. \exists y. y * 2 = x$ true for rationals

$x < y \rightarrow \exists z. x < z \wedge z < y$ true for any dense order

$\exists x. x \neq 0$ true for domains with more than one element

$(\forall x. p(x, x)) \rightarrow p(a, a)$

Quantifiers: Intuition

- A variable stands for “some thing” in a domain of discourse. Quantifiers \forall, \exists are used to speak about all or some members of this domain.
- Examples: Are they satisfiable? valid?

$\forall x. \exists y. y * 2 = x$ true for rationals

$x < y \rightarrow \exists z. x < z \wedge z < y$ true for any dense order

$\exists x. x \neq 0$ true for domains with more than one element

$(\forall x. p(x, x)) \rightarrow p(a, a)$ valid

First-Order Logic: Syntax

- Two syntactic categories: **terms** and **formulae**
- A **first-order language** is characterized by giving a finite collection of function symbols \mathcal{F} and predicates \mathcal{P} as well as a set Var of variables.

First-Order Logic: Syntax

- Two syntactic categories: **terms** and **formulae**
- A **first-order language** is characterized by giving a finite collection of function symbols \mathcal{F} and predicates \mathcal{P} as well as a set Var of variables.
- Sometimes write f^i (or p^i) to indicate that function symbol f (predicate p) has arity $i \in \mathbb{N}$.

First-Order Logic: Syntax

- Two syntactic categories: **terms** and **formulae**
- A **first-order language** is characterized by giving a finite collection of function symbols \mathcal{F} and predicates \mathcal{P} as well as a set Var of variables.
- Sometimes write f^i (or p^i) to indicate that function symbol f (predicate p) has arity $i \in \mathbb{N}$.
- One often calls the pair $\langle \mathcal{F}, \mathcal{P} \rangle$ a **signature**.

Terms in First-Order Logic

Term, the set of **terms**, is the **smallest** set where

1. $x \in \text{Term}$ if $x \in \text{Var}$, and
2. $f^n(t_1, \dots, t_n) \in \text{Term}$ if $f^n \in \mathcal{F}$ and $t_j \in \text{Term}$, for all $1 \leq j \leq n$.

Formulae in First-Order Logic

Form, the set of formulae, is the smallest set where

1. $\perp \in Form$,
2. $p^n(t_1, \dots, t_n) \in Form$ if $p^n \in \mathcal{P}$ and $t_j \in Term$, for all $1 \leq j \leq n$,
3. $\neg\phi \in Form$ if $\phi \in Form$,
4. $(\phi \circ \psi) \in Form$ if $\phi \in Form$, $\psi \in Form$ and $\circ \in \{\wedge, \vee, \rightarrow\}$,
5. $Qx.\phi \in Form$ if $\phi \in Form$ and $x \in Var$ and $Q \in \{\forall, \exists\}$.

Formulae as in point 2 are called atoms.

Note quantifier scoping.

Variable Occurrences

- All occurrences of a variable in a formula are **bound** or **free** or **binding**.

A variable x in a formula ϕ is **bound** if x occurs within a subformula of ϕ of the form $\exists x.\psi$ or $\forall x.\psi$.

- Example:

$$(q(x) \vee \exists x. \forall y. p(f(x), z) \wedge q(y)) \vee \forall x. r(x, z, g(x))$$

Which are **bound**?

Variable Occurrences

- All occurrences of a variable in a formula are **bound** or **free** or **binding**.

A variable x in a formula ϕ is **bound** if x occurs within a subformula of ϕ of the form $\exists x.\psi$ or $\forall x.\psi$.

- Example:

$$(q(x) \vee \exists x. \forall y. p(f(\textcolor{red}{x}), z) \wedge q(\textcolor{red}{y})) \vee \forall x. r(\textcolor{red}{x}, z, g(\textcolor{red}{x}))$$

Which are **bound**?

Which are **free**?

Variable Occurrences

- All occurrences of a variable in a formula are **bound** or **free** or **binding**.

A variable x in a formula ϕ is **bound** if x occurs within a subformula of ϕ of the form $\exists x.\psi$ or $\forall x.\psi$.

- Example:

$$(q(x) \vee \exists x. \forall y. p(f(x), z) \wedge q(y)) \vee \forall x. r(x, z, g(x))$$

Which are **bound**?

Which are **free**?

Which are **binding**?

Variable Occurrences

- All occurrences of a variable in a formula are **bound** or **free** or **binding**.

A variable x in a formula ϕ is **bound** if x occurs within a subformula of ϕ of the form $\exists x.\psi$ or $\forall x.\psi$.

- Example:

$$(q(x) \vee \exists x. \forall y. p(f(x), z) \wedge q(y)) \vee \forall x. r(x, z, g(x))$$

Which are **bound**?

Which are **free**?

Which are **binding**?

First-Order Logic: Semantics

A **structure** is a pair $\mathcal{A} = \langle U_{\mathcal{A}}, I_{\mathcal{A}} \rangle$ where $U_{\mathcal{A}}$ is a nonempty set, the **universe**, and $I_{\mathcal{A}}$ is a mapping where

1. $I_{\mathcal{A}}(f^n)$ is an n -ary (total) function on $U_{\mathcal{A}}$, for $f^n \in \mathcal{F}$,
2. $I_{\mathcal{A}}(p^n)$ is an n -ary relation on $U_{\mathcal{A}}$, for $p^n \in \mathcal{P}$, and
3. $I_{\mathcal{A}}(x)$ is an element of $U_{\mathcal{A}}$, for each $x \in \text{Var}$.

As shorthand, write $p^{\mathcal{A}}$ for $I_{\mathcal{A}}(p)$, etc.

The Value of Terms

Let \mathcal{A} be a structure. We define the **value of a term t under \mathcal{A}** , written $\mathcal{A}(t)$, as

1. $\mathcal{A}(x) = x^{\mathcal{A}}$, for $x \in Var$, and
2. $\mathcal{A}(f(t_1, \dots, t_n)) = f^{\mathcal{A}}(\mathcal{A}(t_1), \dots, \mathcal{A}(t_n))$.

The Value of Formulae

We define the (truth-)value of the formula ϕ under \mathcal{A} , written $\mathcal{A}(\phi)$, as

$$\mathcal{A}(\perp) = 0$$

$$\mathcal{A}(p(t_1, \dots, t_n)) = \begin{cases} 1 & \text{if } (\mathcal{A}(t_1), \dots, \mathcal{A}(t_n)) \in p^{\mathcal{A}} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{A}(\neg\phi) = \begin{cases} 1 & \text{if } \mathcal{A}(\phi) = 0 \\ 0 & \text{otherwise} \end{cases}$$

⋮

The Value of Formulae (2)

We define the (truth-)value of the formula ϕ under \mathcal{A} , written $\mathcal{A}(\phi)$, as

$$\mathcal{A}(\forall x. \phi) = \begin{cases} 1 & \text{if for all } u \in U_{\mathcal{A}}, \mathcal{A}_{[x/u]}(\phi) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{A}(\exists x. \phi) = \begin{cases} 1 & \text{if for some } u \in U_{\mathcal{A}}, \mathcal{A}_{[x/u]}(\phi) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Models

- If $\mathcal{A}(\phi) = 1$, we write $\mathcal{A} \models \phi$ and say ϕ is true in \mathcal{A} or \mathcal{A} is a model of ϕ .

Models

- If $\mathcal{A}(\phi) = 1$, we write $\mathcal{A} \models \phi$ and say ϕ is true in \mathcal{A} or \mathcal{A} is a model of ϕ .
- If every suitable structure is a model, we write $\models \phi$ and say ϕ is valid or ϕ is a tautology.

Models

- If $\mathcal{A}(\phi) = 1$, we write $\mathcal{A} \models \phi$ and say ϕ is true in \mathcal{A} or \mathcal{A} is a model of ϕ .
- If every suitable structure is a model, we write $\models \phi$ and say ϕ is valid or ϕ is a tautology.
- If there is at least one model for ϕ , then ϕ is satisfiable.

Models

- If $\mathcal{A}(\phi) = 1$, we write $\mathcal{A} \models \phi$ and say ϕ is true in \mathcal{A} or \mathcal{A} is a model of ϕ .
- If every suitable structure is a model, we write $\models \phi$ and say ϕ is valid or ϕ is a tautology.
- If there is at least one model for ϕ , then ϕ is satisfiable.
- If there is no model for ϕ , then ϕ is contradictory.

Models

- If $\mathcal{A}(\phi) = 1$, we write $\mathcal{A} \models \phi$ and say ϕ is true in \mathcal{A} or \mathcal{A} is a model of ϕ .
- If every suitable structure is a model, we write $\models \phi$ and say ϕ is valid or ϕ is a tautology.
- If there is at least one model for ϕ , then ϕ is satisfiable.
- If there is no model for ϕ , then ϕ is contradictory.

There is also more differentiated terminology.

An Example

$$\forall x. p(x, s(x))$$

An Example

$$\forall x. p(x, s(x))$$

A model:

$$U_{\mathcal{A}} = \mathbb{N}$$

$$p^{\mathcal{A}} = \{(m, n) \mid m < n\}$$

$$s^{\mathcal{A}}(x) = x + 1$$

An Example

$$\forall x. p(x, s(x))$$

A model:

$$U_{\mathcal{A}} = \mathbb{N}$$

$$p^{\mathcal{A}} = \{(m, n) \mid m < n\}$$

$$s^{\mathcal{A}}(x) = x + 1$$

Not a model:

$$U_{\mathcal{A}} = \{a, b, c\}$$

$$p^{\mathcal{A}} = \{(a, b), (a, c)\}$$

$$s^{\mathcal{A}} = \text{“the identity function”}$$

Towards a Deductive System

Consider an “ordinary” mathematical proof of
if $x > 2$ then $x^2 > 4$.

Towards a Deductive System

Consider an “ordinary” mathematical proof of

$$\text{if } x > 2 \text{ then } x^2 > 4.$$

In natural language, quantifiers are often **implicit**.

Proof:

Towards a Deductive System

Consider an “ordinary” mathematical proof of

if $x > 2$ then $x^2 > 4$.

In natural language, quantifiers are often **implicit**.

Proof: Consider an arbitrary x where $x > 2$.

Then $x = 2 + y$ for some $y > 0$ and hence

$$x^2 = (2 + y)^2 = 4 + 4y + y^2 \geq 4 + 4 + 1 \geq 9 > 4.$$

Towards a Deductive System

Consider an “ordinary” mathematical proof of

$$\text{if } x > 2 \text{ then } x^2 > 4.$$

In natural language, quantifiers are often **implicit**.

Proof: Consider an arbitrary x (\forall -I) where $x > 2$ (\rightarrow -I).

Then $x = 2 + y$ for some $y > 0$ and hence

$$x^2 = (2 + y)^2 = 4 + 4y + y^2 \geq 4 + 4 + 1 \geq 9 > 4.$$

Some phrases used in this proof have a flavor of **introduction rules**.

Towards a Deductive System

Consider an “ordinary” mathematical proof of

$$\text{if } x > 2 \text{ then } x^2 > 4.$$

In natural language, quantifiers are often **implicit**.

Proof: Consider an arbitrary x (\forall -I) where $x > 2$ (\rightarrow -I).

Then $x = 2 + y$ for some $y > 0$ and hence

$$x^2 = (2 + y)^2 = 4 + 4y + y^2 \geq 4 + 4 + 1 \geq 9 > 4.$$

Note: Proof holds for natural numbers. **How** would you adapt for reals?

Weaker Statement

Even easier to prove the weaker statement

$$\exists x. x > 2 \rightarrow x^2 > 4.$$

Let $x = 0$ (indeed any number!). Statement follows as $0 > 2$ implies $0^2 > 4$.

Weaker Statement

Even easier to prove the weaker statement

$$\exists x. x > 2 \rightarrow x^2 > 4.$$

Let $x = 0$ (indeed any number!). Statement follows as $0 > 2$ implies $0^2 > 4$.

Intuition: existential statements are proven by giving a witness.

First-Order Logic: Deductive System

First-order logic is a generalization of propositional logic. All the rules of propositional logic are “inherited”.

But we must introduce rules for the quantifiers.

Universal Quantification (\forall): Rules

$$\frac{P(x)}{\forall x. P(x)} \forall-I^* \quad \frac{\forall x. P(x)}{P(t)} \forall-E$$

where **side condition** (also called: **proviso**) * means: x must be arbitrary.

Universal Quantification (\forall): Rules

$$\frac{P(x)}{\forall x. P(x)} \forall-I^* \qquad \frac{\forall x. P(x)}{P(t)} \forall-E$$

where **side condition** (also called: **proviso**) * means: x must be arbitrary.

Note that rules are **schematic**.

Universal Quantification: Side Condition

What does **arbitrary** mean? Consider the following “proof”

$$x = 0$$

Universal Quantification: Side Condition

What does **arbitrary** mean? Consider the following “proof”

$$\frac{x = 0}{\forall x. x = 0} \forall\text{-I}$$

Universal Quantification: Side Condition

What does **arbitrary** mean? Consider the following “proof”

$$\frac{\frac{[x = 0]^1}{\forall x. x = 0} \forall\text{-I}}{x = 0 \rightarrow \forall x. x = 0} \rightarrow\text{-I}^1$$

Universal Quantification: Side Condition

What does **arbitrary** mean? Consider the following “proof”

$$\begin{array}{c}
 \dfrac{[x = 0]^1}{\forall x. x = 0} \quad \forall\text{-I} \\
 \dfrac{x = 0 \rightarrow \forall x. x = 0}{\forall x. (x = 0 \rightarrow \forall x. x = 0)} \quad \rightarrow\text{-I}^1 \\
 \forall x. (x = 0 \rightarrow \forall x. x = 0) \quad \forall\text{-I}
 \end{array}$$

Universal Quantification: Side Condition

What does **arbitrary** mean? Consider the following “proof”

$$\begin{array}{c}
 \dfrac{[x = 0]^1}{\forall x. x = 0} \quad \forall\text{-I} \\
 \dfrac{x = 0 \rightarrow \forall x. x = 0}{\forall x. (x = 0 \rightarrow \forall x. x = 0)} \quad \rightarrow\text{-I}^1 \\
 \dfrac{\forall x. (x = 0 \rightarrow \forall x. x = 0)}{0 = 0 \rightarrow \forall x. x = 0} \quad \forall\text{-E}
 \end{array}$$

Universal Quantification: Side Condition

What does **arbitrary** mean? Consider the following “proof”

$$\begin{array}{c}
 \dfrac{[x = 0]^1}{\forall x. x = 0} \quad \forall\text{-I} \\
 \dfrac{x = 0 \rightarrow \forall x. x = 0}{\forall x. (x = 0 \rightarrow \forall x. x = 0)} \quad \rightarrow\text{-I}^1 \\
 \dfrac{\forall x. (x = 0 \rightarrow \forall x. x = 0)}{0 = 0 \rightarrow \forall x. x = 0} \quad \forall\text{-E} \\
 \dfrac{}{0 = 0} \quad \text{refl}
 \end{array}$$

Universal Quantification: Side Condition

What does **arbitrary** mean? Consider the following “proof”

$$\begin{array}{c}
 \dfrac{[x = 0]^1}{\forall x. x = 0} \text{ } \forall\text{-I} \\
 \dfrac{x = 0 \rightarrow \forall x. x = 0}{\forall x. (x = 0 \rightarrow \forall x. x = 0)} \text{ } \rightarrow\text{-I}^1 \\
 \dfrac{\forall x. (x = 0 \rightarrow \forall x. x = 0)}{0 = 0 \rightarrow \forall x. x = 0} \text{ } \forall\text{-E} \\
 \dfrac{0 = 0 \rightarrow \forall x. x = 0 \quad \overline{0 = 0} \text{ } refl}{\forall x. x = 0} \text{ } \rightarrow\text{-E}
 \end{array}$$

Universal Quantification: Side Condition

What does **arbitrary** mean? Consider the following “proof”

$$\begin{array}{c}
 \dfrac{[x = 0]^1}{\forall x. x = 0} \text{ } \forall\text{-I} \\
 \dfrac{x = 0 \rightarrow \forall x. x = 0}{\forall x. (x = 0 \rightarrow \forall x. x = 0)} \text{ } \rightarrow\text{-I}^1 \\
 \dfrac{\forall x. (x = 0 \rightarrow \forall x. x = 0)}{0 = 0 \rightarrow \forall x. x = 0} \text{ } \forall\text{-E} \\
 \dfrac{0 = 0 \rightarrow \forall x. x = 0 \quad 0 = 0}{\forall x. x = 0} \text{ } \rightarrow\text{-E} \text{ refl}
 \end{array}$$

Formal meaning of **side condition**: x not free in any open assumption on which $P(x)$ depends. **Violated!**

Another Proof? (1)

Is the following a proof? Is the conclusion valid?

$$\frac{\frac{[\forall x. \neg \forall y. x = y]^1}{\neg \forall y. y = y} \forall-E}{(\forall x. \neg \forall y. x = y) \rightarrow \neg \forall y. y = y} \rightarrow-I^1$$

Another Proof? (1)

Is the following a proof? Is the conclusion valid?

$$\frac{\frac{[\forall x. \neg \forall y. x = y]^1}{\neg \forall y. y = y} \forall-E}{(\forall x. \neg \forall y. x = y) \rightarrow \neg \forall y. y = y} \rightarrow-I^1$$

Conclusion is not valid.

The formula is false when $U_{\mathcal{A}}$ has at least 2 elements.

Another Proof? (1)

Is the following a proof? Is the conclusion valid?

$$\frac{\frac{[\forall x. \neg \forall y. x = y]^1}{\neg \forall y. y = y} \forall-E}{(\forall x. \neg \forall y. x = y) \rightarrow \neg \forall y. y = y} \rightarrow-I^1$$

Proof is incorrect.

Reason: **Substitution** must avoid **capturing** variables. Replacing x with y in $\forall-E$ is illegal because y is **bound** in $\neg \forall y. y = y$. This detail concerns substitution (and renaming of **bound** variables), not $\forall-E$.

Another Proof? (2)

$$\forall x. A(x) \wedge B(x)$$

Another Proof? (2)

$$\frac{\forall x. A(x) \wedge B(x)}{A(x) \wedge B(x)} \forall\text{-}E$$

Another Proof? (2)

$$\frac{\frac{\forall x. A(x) \wedge B(x)}{A(x) \wedge B(x)} \forall-E}{A(x)} \wedge-EL$$

Another Proof? (2)

$$\frac{\frac{\frac{\forall x. A(x) \wedge B(x)}{A(x) \wedge B(x)} \forall-E}{A(x)} \wedge-EL}{\forall x. A(x)} \forall-I$$

Another Proof? (2)

$$\begin{array}{c}
 \frac{\forall x. A(x) \wedge B(x)}{A(x) \wedge B(x)} \forall\text{-}E \\
 \frac{A(x) \wedge B(x)}{A(x)} \wedge\text{-}EL \\
 \frac{A(x)}{\forall x. A(x)} \forall\text{-}I
 \end{array}
 \qquad
 \begin{array}{c}
 \frac{\forall x. A(x) \wedge B(x)}{A(x) \wedge B(x)} \forall\text{-}E \\
 \frac{A(x) \wedge B(x)}{B(x)} \wedge\text{-}ER \\
 \frac{B(x)}{\forall x. B(x)} \forall\text{-}I
 \end{array}$$

Another Proof? (2)

$$\begin{array}{c}
 \frac{\forall x. A(x) \wedge B(x)}{A(x) \wedge B(x)} \forall\text{-}E \quad \frac{\forall x. A(x) \wedge B(x)}{A(x) \wedge B(x)} \forall\text{-}E \\
 \frac{A(x) \wedge B(x)}{A(x)} \wedge\text{-}EL \quad \frac{A(x) \wedge B(x)}{B(x)} \wedge\text{-}ER \\
 \frac{A(x)}{\forall x. A(x)} \forall\text{-}I \quad \frac{B(x)}{\forall x. B(x)} \forall\text{-}I \\
 \frac{\forall x. A(x) \quad \forall x. B(x)}{(\forall x. A(x)) \wedge (\forall x. B(x))} \wedge\text{-}I
 \end{array}$$

Another Proof? (2)

$$\begin{array}{c}
 \frac{[\forall x. A(x) \wedge B(x)]^1}{A(x) \wedge B(x)} \forall\text{-}E \quad \frac{[\forall x. A(x) \wedge B(x)]^1}{A(x) \wedge B(x)} \forall\text{-}E \\
 \frac{A(x) \wedge B(x)}{A(x)} \wedge\text{-}EL \quad \frac{A(x) \wedge B(x)}{B(x)} \wedge\text{-}ER \\
 \frac{A(x)}{\forall x. A(x)} \forall\text{-}I \quad \frac{B(x)}{\forall x. B(x)} \forall\text{-}I \\
 \frac{\forall x. A(x) \quad \forall x. B(x)}{(\forall x. A(x)) \wedge (\forall x. B(x))} \wedge\text{-}I \\
 \frac{(\forall x. A(x)) \wedge (\forall x. B(x))}{(\forall x. A(x) \wedge B(x)) \rightarrow ((\forall x. A(x)) \wedge (\forall x. B(x)))} \rightarrow\text{-}I^1
 \end{array}$$

Another Proof? (2)

$$\begin{array}{c}
 \frac{[\forall x. A(x) \wedge B(x)]^1}{A(x) \wedge B(x)} \forall\text{-}E \quad \frac{[\forall x. A(x) \wedge B(x)]^1}{A(x) \wedge B(x)} \forall\text{-}E \\
 \frac{A(x) \wedge B(x)}{A(x)} \wedge\text{-}EL \quad \frac{A(x) \wedge B(x)}{B(x)} \wedge\text{-}ER \\
 \frac{A(x)}{\forall x. A(x)} \forall\text{-}I \quad \frac{B(x)}{\forall x. B(x)} \forall\text{-}I \\
 \frac{\forall x. A(x) \quad \forall x. B(x)}{(\forall x. A(x)) \wedge (\forall x. B(x))} \wedge\text{-}I \\
 \frac{(\forall x. A(x)) \wedge (\forall x. B(x))}{(\forall x. A(x) \wedge B(x)) \rightarrow ((\forall x. A(x)) \wedge (\forall x. B(x)))} \rightarrow\text{-}I^1
 \end{array}$$

Yes (check side conditions of $\forall\text{-}I$).

Aside: $A \leftrightarrow B$

Define $A \leftrightarrow B$ as $A \rightarrow B \wedge B \rightarrow A$.

The following rule can be **derived** (in propositional logic, actually):

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array} \quad \begin{array}{c} [B] \\ \vdots \\ A \end{array}}{A \leftrightarrow B} \leftrightarrow -I$$

You could do this as an exercise!

Proof?

$$\frac{\frac{[A]^1}{\forall x. A} \forall-I \quad \frac{[\forall x. A]^1}{A} \forall-E}{A \leftrightarrow \forall x. A} \leftrightarrow-I^1$$

Proof?

$$\frac{\frac{[A]^1}{\forall x. A} \forall-I \quad \frac{[\forall x. A]^1}{A} \forall-E}{A \leftrightarrow \forall x. A} \leftrightarrow-I^1$$

Yes, but only if x not **free** in A .

Proof?

$$\frac{\frac{[A]^1}{\forall x. A} \forall\text{-}I \quad \frac{[\forall x. A]^1}{A} \forall\text{-}E}{A \leftrightarrow \forall x. A} \leftrightarrow\text{-}I^1$$

Yes, but only if x not **free** in A .

Similar requirement arises in proving
 $(\forall x. A \rightarrow B(x)) \leftrightarrow (A \rightarrow \forall x. B(x)).$

Existential Quantification

- We could define $\exists x. A$ as $\neg \forall x. \neg A$.
- Equivalence follows from our definition of semantics.

$$\mathcal{A}(\neg A) = \begin{cases} 1 & \text{if } \mathcal{A}(A) = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{A}(\forall x. A) = \begin{cases} 1 & \text{if for all } u \in U_{\mathcal{A}}, \mathcal{A}_{[x/u]}(A) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{A}(\exists x. A) = \begin{cases} 1 & \text{if for some } u \in U_{\mathcal{A}}, \mathcal{A}_{[x/u]}(A) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Conclude: $\mathcal{A}(\exists x. A) = \mathcal{A}(\neg \forall x. \neg A)$

Where do the Rules for \exists Come from?

- We can use definition $\exists x. A \equiv \neg \forall x. \neg A$ and the given rules for \forall to derive ND proof rules.

Where do the Rules for \exists Come from?

- We can use definition $\exists x. A \equiv \neg \forall x. \neg A$ and the given rules for \forall to derive ND proof rules.
- Alternatively, we can give rules as part of the deduction system and prove equivalence as a lemma, instead of by definition.

\exists -/ as a Derived Rule

The rule:

$$\frac{A(t)}{\exists x. A(x)} \exists\text{-/}$$

$$\exists x. A(x)$$

We want to have $\exists x. A(x)$ as conclusion.

\exists -/ as a Derived Rule

The rule:

$$\frac{A(t)}{\exists x. A(x)} \exists\text{-/}$$

$$\neg\forall x. \neg A(x)$$

But by definition that's $\neg\forall x. \neg A(x)$.

\exists -/ as a Derived Rule

The rule:

$$\frac{A(t)}{\exists x. A(x)} \exists\text{-/}$$

$$\forall x. \neg A(x)$$

$$\perp$$

$$\neg \forall x. \neg A(x)$$

We aim for applying \rightarrow -/ in the last step (recall \neg -definition).

\exists -/ as a Derived Rule

The rule:

$$\frac{A(t)}{\exists x. A(x)} \exists\text{-I}$$

$$\frac{\forall x. \neg A(x)}{\neg A(t)} \forall\text{-E}$$

\perp

$$\neg \forall x. \neg A(x)$$

We apply $\forall\text{-E}$.

\exists -/ as a Derived Rule

The rule:

$$\frac{A(t)}{\exists x. A(x)} \exists\text{-I}$$

$$\frac{\frac{\frac{\forall x. \neg A(x)}{\neg A(t)} \forall\text{-E} \quad A(t)}{\bot} \rightarrow\text{-E}}{\neg \forall x. \neg A(x)}$$

Making assumption $A(t)$ allows us to use $\rightarrow\text{-E}$ (recall \neg -definition).

\exists -/ as a Derived Rule

The rule:

$$\frac{A(t)}{\exists x. A(x)} \exists\text{-I}$$

$$\frac{\frac{[\forall x. \neg A(x)]^1}{\neg A(t)} \forall\text{-E} \quad A(t)}{\perp} \rightarrow\text{-E}$$

$$\frac{}{\neg \forall x. \neg A(x)} \rightarrow\text{-I}^1$$

Finally we can apply \rightarrow -I. Note that the assumption $A(t)$ is still open.

\exists - E as a Derived Rule

The rule:

$$\frac{\exists x. A(x) \quad \begin{array}{c} [A(x)] \\ \vdots \\ B \end{array}}{B} \exists\text{-}E$$

$$\exists x. A(x)$$

We will use $\exists x. A(x)$ as one assumption.

\exists - E as a Derived Rule

The rule:

$$\frac{\exists x. A(x) \quad \begin{array}{c} [A(x)] \\ \vdots \\ B \end{array}}{B} \exists\text{-}E$$

$$\neg \forall x. \neg A(x)$$

But by definition that's $\neg \forall x. \neg A(x)$.

\exists - E as a Derived Rule

The rule:

$$\frac{\exists x. A(x) \quad \begin{array}{c} [A(x)] \\ \vdots \\ B \end{array}}{B} \exists\text{-}E$$

$$\begin{array}{c} A(x) \\ \vdots \\ B \end{array}$$

$$\neg \forall x. \neg A(x)$$

We assume a hypothetical derivation.

\exists -E as a Derived Rule

The rule:

$$\frac{\frac{\exists x. A(x) \quad [A(x)]}{B} \quad B}{\exists x. A(x)} \exists\text{-E}$$

$$\frac{\neg B \quad \begin{array}{c} A(x) \\ \vdots \\ B \end{array}}{\perp} \rightarrow\text{-E}$$

$$\neg \forall x. \neg A(x)$$

We make an additional assumption and apply $\rightarrow\text{-E}$ (recall \neg -definition)

\exists -E as a Derived Rule

The rule:

$$\frac{\exists x. A(x) \quad \begin{array}{c} [A(x)] \\ \vdots \\ B \end{array}}{B} \exists\text{-}E$$

$$\frac{\neg B \quad \begin{array}{c} [A(x)]^2 \\ \vdots \\ B \end{array}}{\perp} \rightarrow\text{-}E$$

$$\frac{\perp}{\neg A(x)} \rightarrow\text{-}I^2$$

$$\neg \forall x. \neg A(x)$$

Now we can discharge the assumption $A(x)$ made in the hypothetical derivation.

\exists -E as a Derived Rule

The rule:

$$\frac{\exists x. A(x) \quad \begin{array}{c} [A(x)] \\ \vdots \\ B \end{array}}{B} \exists\text{-}E$$

$$\frac{\neg B \quad \begin{array}{c} [A(x)]^2 \\ \vdots \\ B \end{array}}{\perp} \rightarrow\text{-}E$$

$$\frac{\perp}{\neg A(x)} \rightarrow\text{-}I^2$$

$$\frac{\neg A(x)}{\forall x. \neg A(x)} \forall\text{-}I$$

$$\neg \forall x. \neg A(x)$$

At this step, the side condition from \forall -I applies. \exists -E will inherit it!

\exists -E as a Derived Rule

The rule:

$$\frac{\exists x. A(x) \quad \begin{array}{c} [A(x)] \\ \vdots \\ B \end{array}}{B} \exists\text{-}E$$

$$\frac{\neg \forall x. \neg A(x) \quad \frac{\frac{\neg B \quad \begin{array}{c} [A(x)]^2 \\ \vdots \\ B \end{array}}{\perp} \rightarrow\text{-}E \quad \frac{\perp}{\neg A(x)} \rightarrow\text{-}I^2}{\forall x. \neg A(x)} \forall\text{-}I}{\perp} \rightarrow\text{-}E$$

We apply $\rightarrow\text{-}E$.

\exists -E as a Derived Rule

The rule:

$$\frac{\exists x. A(x) \quad \begin{array}{c} [A(x)] \\ \vdots \\ B \end{array}}{B} \exists\text{-}E$$

$$\frac{\begin{array}{c} [A(x)]^2 \\ \vdots \\ B \end{array} \quad \frac{[\neg B]^1 \quad B}{\perp} \rightarrow\text{-}E}{\neg A(x)} \rightarrow\text{-}I^2$$

$$\frac{\neg \forall x. \neg A(x) \quad \frac{\neg A(x)}{\forall x. \neg A(x)} \forall\text{-}I}{\perp} \rightarrow\text{-}E$$

$$\frac{\perp}{B} RAA^1$$

We are done. Note that this proof uses **classical** reasoning.

Sample Derivation

Assumption: x does not occur free in B

$$\begin{array}{c}
 \frac{[\forall x. A(x) \rightarrow B]^1}{A(x) \rightarrow B} \forall\text{-}E \quad \frac{[A(x)]^3}{B} \rightarrow\text{-}E \\
 \frac{[\exists x. A(x)]^2}{B} \exists\text{-}E^3 \\
 \frac{(\exists x. A(x)) \rightarrow B}{(\forall x. A(x) \rightarrow B) \rightarrow ((\exists x. A(x)) \rightarrow B)} \rightarrow\text{-}I^2 \\
 \frac{}{(\forall x. A(x) \rightarrow B) \rightarrow ((\exists x. A(x)) \rightarrow B)} \rightarrow\text{-}I^1
 \end{array}$$

Conclusion on FOL

- Propositional logic is good for modeling simple patterns of reasoning like “if . . . then . . . else”.

Conclusion on FOL

- Propositional logic is good for modeling simple patterns of reasoning like “if . . . then . . . else”.
- In first-order logic, one has “things” and relations on / properties of “things”. Quantify over “things”. Powerful!
- Some people advocate intuitionistic, relevant, and other “deviant” logics.
- Limitation: cannot quantify over predicates.
- “A” world or “the” world is modeled in first-order logic using so-called first-order theories. This will be studied next lecture. ►

More Detailed Explanations

Boolean Functions

The set (or “type”) $Bool$ contains the two truth values $True, False$. A propositional formula containing n variables can be viewed as a function $Bool^n \rightarrow Bool$. For each combination of values $True, False$ for the variables, the whole formula assumes the value $True$ or $False$.

Relations/Functions, Infinity

In propositional logic, there is no notation for writing “thing x has property p ” or “things x and y are related as follows” or for denoting the “thing obtained from thing x by applying some operation”.

In particular, no statement about all elements of a possibly infinite domain can be expressed in propositional logic, since each formula involves only finitely many different variables, and up to **equivalence** and for a set containing n variables, there are only finitely many (to be precise 2^n) different propositional formulae.

What is a Domain?

For example, the set of integers, the set of characters, the set of people, you name it!

Any set of “things” that we want to reason about.

Alternating State

Say, of a pedestrian traffic light which alternates between red and non-red (=green).

The State Alternates over all Time

“The state alternates over all time” could be expressed by the first-order logic formula $\forall t. x(t) \leftrightarrow \neg x(t + 1)$. We explain it intuitively.

x is some property which may or may not hold at a given point of time t , expressed as a number. So we write $x(t)$ if x holds at t . The formula says that x is true at point t if and only if x is not true at point $t + 1$. To be more precise, one would have to axiomatize the fact that t is a number etc.

“Fixed Thing”?

As opposed to a variable which also stands for a “thing”.
This distinction will become clear [soon](#).

Function Notation

So a function symbol f denotes an operation that takes n “things” and returns a “thing”. $f(t_1, \dots, t_n)$ is a “thing” that depends on “things” t_1, \dots, t_n .

The generic notation for function application is like this: $f(t_1, \dots, t_n)$, but the brackets are omitted for nullary functions (= constants), and many common function symbols like $+$ are denoted **infix**, so we write $0 + 0$ instead of $+(0, 0)$. Another common notation is **prefix** notation without brackets, as in -2 . There are also other notations.

Equational Axiomatization

How do we formalize/axiomatize the fundamental truths of our domain of interest?

It turns out that much of this is done by stating which terms are equal to which other terms.

Here it is assumed that $=$ is interpreted as the identity on the domain.

“Some Thing”?

Just like a constant, a variable stands for a “thing”.

The most important difference between a constant and a variable is that one can **quantify** over a variable, so one can make statements such as “**for all** x . . . ” or “**there exists** x such that . . . ”.

What is Satisfiability? Validity?

Intuitively, **satisfiable** means “can be made true” and **valid** means “always true”.

More formally, this will be defined **later**.

Syntactic Categories

We have already learned about the syntactic category of **formulae** last lecture.

A **term** is an expression that stands for a “thing”.

Intuitively, this is what first-order logic is about: We have terms that stand for “things” and formulae that stand for statements/propositions about those “things”.

But couldn't a statement also be a “thing”? And couldn't a “thing” depend on a statement?

In first-order logic: **no!**

Signatures

There isn't simply **the** language of first-order logic! Rather, the definition of **a** first-order language is **parametrised** by giving a \mathcal{F} and a \mathcal{P} . Each symbol in \mathcal{F} and \mathcal{P} must have an associated **arity**, i.e., the number of arguments the function or predicate takes. This could be formalised by saying that the elements of \mathcal{F} are **pairs** of the form f/n , where f is the symbol itself and n , and likewise for \mathcal{P} . All that matters is that it is specified in some unambiguous way what the arity of each symbols is. One often calls the pair $\langle \mathcal{F}, \mathcal{P} \rangle$ a **signature**. Generally, a signature specifies the “fixed symbols” (as opposed to variables) of a particular logic language.

Strictly speaking, a first-order language is also parametrised by giving a set of variables Var , but this is inessential. Var is usually assumed to be a countably infinite set of symbols, and the particular choice of names of

these symbols is not relevant.

A Language

Term and *Form* together make up a first-order language. Note that strictly speaking, *Term* and *Form* depend on the signature, but we always assume that the signature is clear from the context.

The Scope of a Quantifier

We adopt the convention that the scope of a quantifier extends as much as possible to the right, e.g.

$$\forall x.p(x) \vee q(x)$$

is

$$\forall x.(p(x) \vee q(x))$$

and not

$$(\forall x.p(x)) \vee q(x)$$

This is a matter of dispute and other conventions are around, but the one we adopt here corresponds to Isabelle.

Compare this to the [precedences](#) and associativity in propositional logic.

Free, Bound, and Binding Occurrences

All occurrences of a variable in a term or formula are **bound** or **free** or **binding**. These notions are defined by induction on the structure of terms/formulae. This is why the following definition is along the lines of our definition of **terms** and **formulae**.

1. The (only) occurrence of x in the term x is a free occurrence of x in x ;
2. the free occurrences of x in $f(t_1, \dots, t_n)$ are the free occurrences of x in t_1, \dots, t_n ;
3. there are no free occurrences of x in \perp ;
4. the free occurrences of x in $p(t_1, \dots, t_n)$ are the free occurrences of x in t_1, \dots, t_n ;
5. the free occurrences of x in $\neg\phi$ are the free occurrences of x in ϕ ;
6. the free occurrences of x in $\psi \circ \phi$ are the free occurrences of x in ψ

- and the free occurrences of x in ϕ ($\circ \in \{\wedge, \vee, \rightarrow\}$);
7. the free occurrences of x in $\forall y. \psi$, where $y \neq x$, are the free occurrences of x in ψ ; likewise for \exists ;
 8. x has no free occurrences in $\forall x. \psi$; in $\forall x. \psi$, the (outermost) \forall binds all free occurrences of x in ψ ; the occurrence of x next to \forall is a **binding** occurrence of x ; likewise for \exists .

A variable occurrence is **bound** if it is not free and not binding.

We also define

$$FV(\phi) := \{x \mid x \text{ has a free occurrence in } \phi\}$$

Structures

As usual, there isn't just one way of formalizing things, and so we now explain some other notions that you may have heard in the context of semantics for first-order logic.

A **universe** is sometimes also called **domain**.

As you saw, a **structure** gives a meaning to **functions**, **predicates**, and **variables**.

An alternative formalization is to have three different mappings for this purpose:

1. an **algebra** gives a meaning to the function symbols (more precisely, an algebra is a pair consisting of a domain and a mapping giving a meaning to the function symbols);
2. in addition, an **interpretation** gives a meaning also to the predicate

symbols;

3. a **variable assignment**, also called **valuation**, gives a meaning to the variables.

As **before**, we assume that the **signature** is clear from the context. Strictly speaking, we should say “structure for a particular signature”. Details can be found in any textbook on logic [**vD80**].

Omitted Cases

The following cases have been omitted:

$$\mathcal{A}(\phi \wedge \psi) = \begin{cases} 1 & \text{if } \mathcal{A}(\phi) = 1 \text{ and } \mathcal{A}(\psi) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{A}(\phi \vee \psi) = \begin{cases} 1 & \text{if } \mathcal{A}(\phi) = 1 \text{ or } \mathcal{A}(\psi) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{A}(\phi \rightarrow \psi) = \begin{cases} 1 & \text{if } \mathcal{A}(\phi) = 0 \text{ or } \mathcal{A}(\psi) = 1 \\ 0 & \text{otherwise} \end{cases}$$

These cases are the same as in [propositional logic](#). This can also be found in any textbook on logic [[vD80](#)].

In mathematics and computer science, the word **or** is almost always meant to be **inclusive**. If it is meant to be **exclusive** (A or B hold but not both) this is usually mentioned explicitly.

The Notation $\mathcal{A}_{[x/u]}$

$\mathcal{A}_{[x/u]}$ is the structure \mathcal{A}' identical to \mathcal{A} , except that $x^{\mathcal{A}'} = u$.

Models

If you are happy with the definition of a model just given, this is fine. But if you are confused because you remember a different definition from your previous studies of logic, then these comments may help.

As explained [before](#), it is common to distinguish an **interpretation**, which gives a meaning to the symbols in the signature, from an **assignment**, which gives a meaning to the variables. Let us use \mathcal{I} to denote an interpretation and A to denote an assignment.

Recall that we wrote $\mathcal{A}(\cdot)$ for the meaning of a **term** or **formula**. In the alternative terminology, we write $\mathcal{I}(A)(\cdot)$ instead. This makes sense since in the alternative terminology, \mathcal{I} and A **together** contain the same information as \mathcal{A} in the original terminology. We define:

- For a given \mathcal{I} , we say that ϕ is **satisfiable in \mathcal{I}** if there exists an A so that $\mathcal{I}(A)(\phi) = 1$;

- for a given \mathcal{I} , we write $\mathcal{I} \models \phi$ and say ϕ is **true in \mathcal{I}** or **\mathcal{I} is a model of ϕ** , if for all A , we have $\mathcal{I}(A)(\phi) = 1$;
- we say ϕ is **satisfiable** if there exists a \mathcal{I} so that ϕ is satisfiable in \mathcal{I} ;
- we write $\models \phi$ and say ϕ is **valid** if for every (**suitable**) \mathcal{I} , we have $\mathcal{I} \models \phi$.

Note that **satisfiable** (without “for . . .”) and **valid** mean the same thing in both terminologies, whereas **true in . . .** means slightly different things, since a structure is not the same thing as an interpretation.

Suitable Structures

A **structure** is suitable for ϕ if it defines meanings for the **signature** of ϕ , i.e., for the symbols that occur in ϕ . Of course, these meanings must also respect the arities, so an n -ary function symbols must be interpreted as an n -ary function. **Without explicitly mentioning it**, we always assume that structures are suitable.

\mathbb{N}

\mathbb{N} denotes the natural numbers.

Confusion of Syntax and Semantics?

In logic, we insist on the distinction between **syntax** and **semantics**. In particular, we set up the formalism so that the **syntax is fixed first** and **then the semantics**, and so there could be different semantics for the same syntax.

But the dilemma is that once we want to give a particular semantics, we can only do so using again some kind of **language**, hence syntax. This is usually natural language interspersed with usual mathematical notation such as $<$, $+$ etc.

Some people try to mark the distinction between syntax and semantics somehow, e.g., by saying 0 is a constant that could mean anything, whereas **0** is the number zero as it exists in the mathematical world.

When we give semantics, the symbols $<$, $+$, and 1 have their usual mathematical meanings. The function that maps x to $x + 1$ is also called

successor function. Of course, when we write $m < n$, we assume that $m, n \in \mathbb{N}$, in this context.

Why is this a Model?

It is true that for all numbers n , n is less than $n + 1$.

Why is this not a Model?

The identity function maps every object to itself.

It is **not** true that for every character $\alpha \in \{a, b, c\}$,
 $(\alpha, \alpha) \in \{(a, b), (a, c)\}$. E.g., $(a, a) \notin \{(a, b), (a, c)\}$.

Implicit Quantifiers

In the statement

$$\text{if } x > 2 \text{ then } x^2 > 4$$

the \forall -quantifier is implicit. It should be

$$\text{for all } x, \text{ if } x > 2 \text{ then } x^2 > 4.$$

Adapt the Proof for Reals

The proof for natural numbers exploits the fact that $y > 0$ implies $y \geq 1$. This is of course not true in the reals.

A proof for the reals:

Consider an arbitrary x where $x > 2$. Then $x = 2 + y$ for some $y > 0$ and hence

$$x^2 = (2 + y)^2 = 4 + 4y + y^2 > 4$$

The proof looks even simpler than for the naturals, and moreover, it also holds for the naturals, since those are a subset of the reals. However, such observations may be deceptive.

Tacitly, the proof uses the following assumptions (which we know are true in the usual interpretation of mathematics):

$$y > 0 \quad \text{implies} \quad 4y > 0$$

$$y > 0 \text{ implies } y^2 > 0$$

$$z > 0 \text{ and } z' > 0 \text{ implies } z + z' > 0$$

$$z > 0 \text{ implies } w + z > w$$

Inheriting Rules

First-order logic inherits all the **rules of propositional logic**. Note however that the **metavariables** in the rules now range over first-order formulae.

Schematic Rules

Similarly as in the [previous lecture](#), one should note that P is not a predicate, but rather $P(x)$ is a **schematic** expression: $P(x)$ stands for any formula, possibly containing occurrences of x .

In the context of \forall - E , $P(t)$ stands for a formula where [all occurrences of \$x\$ are replaced by \$t\$](#) .

Reflexivity

When one has a predicate symbol $=$, it is usual to have a rule that says that $=$ is reflexive.

Don't worry about it at this stage, just take it that we have such a rule. We will look at this later.

Side Condition Violated!

The side condition is violated in the proof since in the first \forall -I step, x does occur free in $x = 0$.

Why is $(\forall x. \neg \forall y. x = y) \rightarrow \neg \forall y. y = y$ False?

Here we assume that the predicate symbol $=$ is interpreted by \mathcal{A} as equality on $U_{\mathcal{A}}$. Suppose $U_{\mathcal{A}}$ contains two elements α and β and $I_{\mathcal{A}}(x) = \alpha$ and $I_{\mathcal{A}}(y) = \beta$. Then $\mathcal{A}(x = y) = 0$, hence $\mathcal{A}(\forall y. x = y) = 0$, hence $\mathcal{A}(\neg \forall y. x = y) = 1$. Now one can see that $\mathcal{A}_{[x/u]}(\neg \forall y. x = y) = 1$ for all $u \in U_{\mathcal{A}}$, and hence $\mathcal{A}(\forall x. \neg \forall y. x = y) = 1$. On the other hand, $\mathcal{A}'(y = y) = 1$ for any \mathcal{A}' and hence $\mathcal{A}(\forall y. y = y) = 1$ and hence $\mathcal{A}(\neg \forall y. y = y) = 0$. Therefore, $\mathcal{A}((\forall x. \neg \forall y. x = y) \rightarrow \neg \forall y. y = y) = 0$.

Substitutions in FOL

The notation $s[x \leftarrow t]$ denotes the term obtained by substituting t for x in s . However, a substitution $[x \leftarrow t]$ replaces only the free occurrences of x in the term that it is applied to. A substitution is defined as follows:

1. $x[x \leftarrow t] = t$;
2. $y[x \leftarrow t] = y$ if y is a variable other than x ;
3. $f(t_1, \dots, t_n)[x \leftarrow t] = f(t_1[x \leftarrow t], \dots, t_n[x \leftarrow t])$ (where f is a function symbol, $n \geq 0$);
4. $p(t_1, \dots, t_n)[x \leftarrow t] = p(t_1[x \leftarrow t], \dots, t_n[x \leftarrow t])$ (where p is a predicate symbol, possibly \perp);
5. $(\neg\psi)[x \leftarrow t] = \neg(\psi[x \leftarrow t])$
6. $(\psi \circ \phi)[x \leftarrow t] = (\psi[x \leftarrow t] \circ \phi[x \leftarrow t])$ (where $\circ \in \{\wedge, \vee, \rightarrow\}$);
7. $(Qx.\psi)[x \leftarrow t] = Qx.\psi$ (where $Q \in \{\forall, \exists\}$);

8. $(Qy.\psi)[x \leftarrow t] = Qy.(\psi[x \leftarrow t])$ (where $Q \in \{\forall, \exists\}$) if $y \neq x$ and $y \notin FV(t)$;
9. $(Qy.\psi)[x \leftarrow t] = Qz.(\psi[y \leftarrow z][x \leftarrow t])$ (where $Q \in \{\forall, \exists\}$) if $y \neq x$ and $y \in FV(t)$ where z is a variable such that $z \notin FV(t)$ and $z \notin FV(\psi)$.

Avoiding Capture of Variables

A **substitution** (replacement of a variable by a term) must not replace **bound** occurrences of variables, and if we replace x with t in an expression ϕ , then this replacement should not turn **free** occurrences of variables in t into **bound** occurrences in ϕ . It is possible to avoid this by renaming variables.

This is part of the standard definition of a **substitution**. The problem is not related to \forall - E in particular.

The definition can be found in any textbook on logic [vD80]. We will also give a formal definition **later**, in a different context.

Check Side Conditions

In both cases, x does not occur **free** in $\forall x. A(x) \wedge B(x)$, which is the **open assumption** on which $A(x)$, respectively $B(x)$, depends.

Defining \leftrightarrow

By **defining** we mean, use $A \leftrightarrow B$ as shorthand for $A \rightarrow B \wedge B \rightarrow A$, in the same way as we regard **negation** as a shorthand.

Defining \exists

By **defining** we mean, use $\exists x. A$ as shorthand for $\neg \forall x. \neg A$, in the same way as we regard **negation** as a shorthand.

However, we have already introduced \exists as syntactic entity, and also its semantics. If we now want to treat it as being defined in terms of \forall , for the purposes of building a deductive system, we must be sure that $\exists x. A$ is semantically equivalent to $\neg \forall x. \neg A$, i.e., that $\mathcal{A}(\exists x. A) = \mathcal{A}(\neg \forall x. \neg A)$.

Where Do the Rules for \exists Come from?

- We can use definition $\exists x. A \equiv \neg \forall x. \neg A$ and the given rules for \forall to **derive** ND proof rules.

In this case, the **soundness** of the derived rules is guaranteed since

- the rules for \forall are sound;
- we have proven the equivalence of $\exists x. A$ and $\neg \forall x. \neg A$ semantically.

- Alternative: give rules as part of the deduction system and prove the equivalence as a lemma, instead of by definition.

In this case, the **soundness** must be proven by hand (however, proving rules sound is an aspect we **neglect** in this course). But once this is done, the equivalence of $\exists x. A$ and $\neg \forall x. \neg A$ can be proven **within the deductive system**, rather than by hand, provided that the deductive system is **complete**.

Hypothetical Derivation

We are constructing here a “schematic fragment” of a derivation tree. Within this construction, we simply assume a hypothetical derivation of B from assumption $A(x)$. When we are done with the construction of this fragment, we will collapse the fragment by throwing away all the nodes in the middle and only keep the root and leaves.

Note two points:

- We assume a hypothetical derivation of B from assumption $A(x)$. Somewhere in the middle of the constructed fragment, we will discharge the assumption $A(x)$. In the final rule \exists - E , this means an application of \exists - E involves discharging $A(x)$. Therefore \exists - E has brackets around the $A(x)$.
- The hypothetical derivation of B may contain other assumptions than $A(x)$. These are not discharged in the constructed fragment, and so in

the final rule \exists - E , we must also read the notation

$$\begin{array}{c} A(x) \\ \vdots \\ B \end{array}$$

as a derivation of B where one of the assumptions is $A(x)$. There may be other assumptions, but these are not discharged. This is no different from [previous rules](#) involving discharging.

Inheriting a Side Condition

\exists - E will inherit the side condition from \forall - I . Hence, the side condition for \exists - E is:

x must not be **free** in B or in hypotheses of the subderivation of B other than $A(x)$ (occurrences in $A(x)$ are allowed because the assumption $A(x)$ was discharged before the application of \forall - I).

Classical Reasoning

Defining $\exists x. A$ as $\neg \forall x. \neg A$ is only sensible in classical reasoning, since the derivation of the rule \exists -E requires the RAA rule.

The Power of First-Order Logic

In first-order logic, one has “things” and relations/properties that may or may not hold for these “things”. Quantifiers are used to speak about “all things” and “some things”.

For example, one can reason:

All men are mortal, Socrates is a man, therefore Socrates is mortal.

The idea underlying first-order logic is so general, abstract, and powerful that vast portions of human (mathematical) reasoning can be modeled with it.

In fact, first-order logic is the most prominent logic of all. Many people know about it: not only mathematicians and computer scientists, but also linguists, philosophers, psychologists, economists etc. are likely to learn about first-order logic in their education.

While some applications in the fields mentioned above require other logics, e.g. **modal logics**, those can often be reduced to first-order logic, so that first-order logic remains the point of reference.

On the other hand, logics that are strictly more expressive than first-order logic are only known to and studied by few specialists within mathematics and computer science.

This example about **Socrates** and **men** is a very well-known one. You may wonder: what is the history of this example?

In English, the example is commonly given using the word “man”, although one also finds “human”. Like many languages (e.g., French, Italian), English often uses “man” for “human being”, although this use of language may be considered discriminating against women.

E.g. [Tho95a]:

man [. . .] **1** an adult human male, esp. as distinct from a woman

or boy. **2** a human being; a person (*no man is perfect*).

While the example does not, strictly speaking, imply that “man” is used in the meaning of “human being”, this is strongly suggested both by the content of the example (or should women be immortal?) and the fact that languages that do have a word for “human being” (e.g. “Mensch” in German) usually give the example using this word. In fact, the example is originally in Old Greek, and there the word ἄνθρωπος (anthropos = human being), as opposed to ἀνὴρ (anér = human male), is used.

The example is a so-called **sylllogism of the first figure**, which the scholastics called **Barbara**. It was developed by Aristotle [**Ari**] in an abstract form, i.e., without using the concrete name “Socrates”. In his terminology, ἄνθρωπος is the middle term that is used as subject in the first premise and as predicate in the second premise (this is what is called **first figure**). Aristotle formulated the syllogism as follows: If A of all B

and B is said of all C, then A must be said of all C.

And why “Socrates”? It is not exactly clear how it came about that this particular syllogism is associated with Socrates. In any case, as far it is known, Socrates did not investigate any questions of logic. However, Aristotle frequently uses **Socrates** and **Kallias** as standard names for individuals [Ari]. Possibly there were statues of Socrates and Kallias standing in the hall where Aristotle gave his lectures, so it was convenient for him to point to the statues whenever he was making a point involving two individuals.

Other Logics

There are still controversies about what the best logic is for reasoning about “things” and properties/relations, and scope (quantification). Some argue for **intuitionistic**, relevance, modal and other “deviant” logics.

An example where first-order logic is inappropriate might be:

From “a dollar buys a candy bar” and “a dollar buys an ice cream” we cannot normally conclude “a dollar buys a candy bar and an ice cream”.

However, such analogies should be treated with care. Depending on how ice-creams, candy bars, dollars and buying are modeled, first-order logic may very well be appropriate.

Modal logics are logics that have **modality operators**, usually \Box and \Diamond .

Sometimes these denote **temporal** aspects, e.g., $\Box\phi$ means “ ϕ always holds”. But many other interpretations are possible, e.g., $\Box_A\phi$ could mean “ A knows that ϕ holds” [HC68].

In relevance logics, it is not true that $A \rightarrow B$ holds whenever A is false. Rather, A must somehow be “relevant” for B .

Limitations of First-Order Logic

The idea underlying first-order logic seems so general that it is not so apparent what its limitations could be. The limitations will become clear as we study more expressive logics.

For the moment, note the following: in first-order logic, we quantify over variables (hence, domain elements), not over predicates. The number of predicates is fixed in a particular first-order language. So for example, it is impossible to express the following:

For all unary predicates p , if there exists an x such that $p(x)$ is true, then there exists a smallest x such that $p(x)$ is true,

since we would be quantifying over p .

First-Order Logic with Equality

Overview

Last lecture: first-order logic.

This lecture:

- first-order logic with equality and first-order theories;
- set-theoretic reasoning.

We extend language and deductive system to formalize and reason about the (mathematical) world.

FOL with Equality

Equality is a logical symbol rather than a mathematical one.
Speak of **first-order logic with equality** rather than adding equality as “just another predicate”.

Syntax and Semantics

Syntax: $=$ is a binary infix predicate.

$t_1 = t_2 \in \textit{Form}$ if $t_1, t_2 \in \textit{Term}$.

Syntax and Semantics

Syntax: $=$ is a binary infix predicate.

$t_1 = t_2 \in \textit{Form}$ if $t_1, t_2 \in \textit{Term}$.

Semantics : recall a **structure** is a pair $\mathcal{A} = \langle U_{\mathcal{A}}, I_{\mathcal{A}} \rangle$ and $I_{\mathcal{A}}(t)$ is the interpretation of t .

$$I_{\mathcal{A}}(s = t) = \begin{cases} 1 & \text{if } I_{\mathcal{A}}(s) = I_{\mathcal{A}}(t) \\ 0 & \text{otherwise} \end{cases}$$

Note the three completely **different** uses of “ $=$ ” here!

Rules

- Equality is an equivalence relation

$$\frac{}{x = x} \text{ refl} \quad \frac{x = y}{y = x} \text{ sym} \quad \frac{x = y \quad y = z}{x = z} \text{ trans}$$

Rules

- Equality is an **equivalence relation**

$$\frac{}{x = x} \text{ refl} \quad \frac{x = y}{y = x} \text{ sym} \quad \frac{x = y \quad y = z}{x = z} \text{ trans}$$

- Equality is also a **congruence** on terms and all relations

$$\frac{x_1 = y_1 \cdots x_n = y_n}{t(x_1, \dots, x_n) = t(y_1, \dots, y_n)} \text{ cong}_1$$

$$\frac{x_1 = y_1 \cdots x_n = y_n \quad A(x_1, \dots, x_n)}{A(y_1, \dots, y_n)} \text{ cong}_2$$

Soundness of Rules

Equality in $I_{\mathcal{A}}$ is an equivalence relation and functions/predicates/logical-operators are “truth-functional”. Adding further rules gives us an equational theory, e.g. groups.

Congruence: Alternatives

One can specialize congruence rules to replace only **some** term occurrences.

$$\frac{x_1 = y_1 \cdots x_n = y_n}{t[z_1 \leftarrow x_1, \dots, z_n \leftarrow x_n] = t[z_1 \leftarrow y_1, \dots, z_n \leftarrow y_n]} \text{cong}_1$$

$$\frac{x_1 = y_1 \cdots x_n = y_n \quad A[z_1 \leftarrow y_1, \dots, z_n \leftarrow y_n]}{A[z_1 \leftarrow x_1, \dots, z_n \leftarrow x_n]} \text{cong}_2$$

One time the z 's are replaced with x 's and one time with y 's.

Examples

How many ways are there to choose some occurrences of x in $x^2 + y^2 > 12 \cdot x$?

Examples

How many ways are there to choose some occurrences of x in $x^2 + y^2 > 12 \cdot x$? 4, namely:

$$A = x^2 + y^2 > 12 \cdot x, \quad A = z^2 + y^2 > 12 \cdot x,$$

$$A = x^2 + y^2 > 12 \cdot z, \quad A = z^2 + y^2 > 12 \cdot z.$$

Examples

How many ways are there to choose some occurrences of x in $x^2 + y^2 > 12 \cdot x$? 4, namely:

$$A = x^2 + y^2 > 12 \cdot x, \quad A = z^2 + y^2 > 12 \cdot x,$$

$$A = x^2 + y^2 > 12 \cdot z, \quad A = z^2 + y^2 > 12 \cdot z.$$

We show two ways:

$$\frac{x = 3 \quad x^2 + y^2 > 12 \cdot x}{3^2 + y^2 > 12 \cdot x} \text{ with } A = z^2 + y^2 > 12 \cdot x$$

$$\frac{x = 3 \quad x^2 + y^2 > 12 \cdot x}{x^2 + y^2 > 12 \cdot 3} \text{ with } A = x^2 + y^2 > 12 \cdot z$$

Isabelle Rule

The Isabelle FOL rule is *simply* (using a tree syntax)

$$\frac{x = y \quad P(x)}{P(y)} \textit{subst}$$

or literally

$$\llbracket a = b; P(a) \rrbracket \Longrightarrow P(b)$$

Proving $\exists x. t = x$

$$\frac{\overline{t = t} \text{ refl}}{\exists x. t = x} \exists\text{-I}$$

Proving $\exists x. t = x$

$$\frac{\overline{t = t} \text{ refl}}{\exists x. t = x} \exists\text{-I}$$

In the rule $\frac{A(t)}{\exists x. A(x)} \exists\text{-I}$, “ $A(x)$ ” is metanotation. In the example, $A(x) = (t = x)$.

Proving $\exists x. t = x$

$$\frac{\overline{t = t} \text{ refl}}{\exists x. t = x} \exists\text{-I}$$

In the rule $\frac{A(t)}{\exists x. A(x)} \exists\text{-I}$, “ $A(x)$ ” is metanotation. In the example, $A(x) = (t = x)$.

Notational confusion avoided by a precise metalanguage. ►

More Detailed Explanations

Logical vs. Non-logical Symbols

In logic languages, it is common to distinguish between **logical** and **non-logical** symbols. We explain this for first-order logic.

Recall that there isn't just **the** language of first-order logic, but rather defining a particular signature gives us **a** first-order language. The **logical** symbols are those that are part of **any** first-order language and whose meaning is “hard-wired” into the formalism of first-order logic, like \wedge or \forall . The **non-logical** symbols are those given by a particular **signature**, and whose meaning must be defined “by the user” by giving a **structure**.

Above we say “mathematical” instead of “non-logical” because we assume that mathematics is our domain of discourse, so that the **signature** contains the symbols of “mathematics”.

Now what status should the equality symbol $=$ have? We will assume that $=$ is a symbol whose meaning is hard-wired into the formalism. One

then speaks of **first-order logic with equality**.

Alternatively, one could regard $=$ as an ordinary (binary infix) predicate. However, even if one does not give $=$ a special status, anyone reading $=$ has a certain expectation. Thus it would be very confusing to have a structure that defines $=$ as a, say, non-reflexive relation.

Three Different Uses of Equality

$$I_{\mathcal{A}}(s \text{ = } t) = \begin{cases} 1 & \text{if } I_{\mathcal{A}}(s) = I_{\mathcal{A}}(t) \\ 0 & \text{otherwise} \end{cases}$$

The first = is a predicate symbol.

Three Different Uses of Equality

$$I_{\mathcal{A}}(s=t) = \begin{cases} 1 & \text{if } I_{\mathcal{A}}(s)=I_{\mathcal{A}}(t) \\ 0 & \text{otherwise} \end{cases}$$

The first $=$ is a predicate symbol.

The second $=$ is a **definitional** occurrence: The expression on the left-hand side is **defined** to be equal to the value of the right-hand side.

Three Different Uses of Equality

$$I_{\mathcal{A}}(s=t) = \begin{cases} 1 & \text{if } I_{\mathcal{A}}(s) = I_{\mathcal{A}}(t) \\ 0 & \text{otherwise} \end{cases}$$

The first $=$ is a predicate symbol.

The second $=$ is a **definitional** occurrence: The expression on the left-hand side is **defined** to be equal to the value of the right-hand side.

The third $=$ is **semantic** equality, i.e., **the identity relation on the domain**.

Why Rules?

Since $=$ is a logical symbol in the formalism of first-order logic with equality, there should be **derivation rules** for $=$ to derive which formulas $a = b$ are true.

What is an Equivalence Relation?

In general mathematical terminology, a relation \equiv is an **equivalence relation** if the following three properties hold:

Reflexivity: $a \equiv a$ for all a ;

Symmetry: $a \equiv b$ implies $b \equiv a$;

Transitivity: $a \equiv b$ and $b \equiv c$ implies $a \equiv c$.

Example: being equal modulo 6.

“ a is equal b modulo 6” is often written $a \equiv b \pmod{6}$.

What is a Congruence?

In general mathematical terminology, a relation \cong is a **congruence w.r.t.** (or: **on**) f , where f has arity n , if $a_1 \cong b_1, \dots, a_n \cong b_n$ implies $f(a_1, \dots, a_n) \cong f(b_1, \dots, b_n)$.

Example: being equal modulo 6 is congruent w.r.t. multiplication.

$14 \equiv 8 \pmod{6}$ and $15 \equiv 9 \pmod{6}$, hence $14 \cdot 15 \equiv 8 \cdot 9 \pmod{6}$.

This can be defined in an analogous way for a property (relation) P .

Example: being equal modulo 6 is congruent w.r.t. divisibility by 3.

$15 \equiv 9 \pmod{6}$ and 15 is divisible by 3, hence 9 is divisible by 3.

$14 \equiv 8 \pmod{6}$ and 14 is not divisible by 3, hence 8 is not divisible by 3.

What Does this Notation Mean?

Why did we use letters t and A here?

Recall the rules for building **terms** and **atoms**.

Is $t(x_1, \dots, x_n)$ a term, and $A(x_1, \dots, x_n)$ an atom, obtained by one application of such a rule, i.e.: is t a function symbol in \mathcal{F} , applied to x_1, \dots, x_n , and is A a predicate symbol in \mathcal{P} , applied to x_1, \dots, x_n ?

What Does this Notation Mean?

Why did we use letters t and A here?

Recall the rules for building **terms** and **atoms**.

Is $t(x_1, \dots, x_n)$ a term, and $A(x_1, \dots, x_n)$ an atom, obtained by one application of such a rule, i.e.: is t a function symbol in \mathcal{F} , applied to x_1, \dots, x_n , and is A a predicate symbol in \mathcal{P} , applied to x_1, \dots, x_n ?

In general, no! The notations $t(x_1, \dots, x_n)$ and $A(x_1, \dots, x_n)$ are **metanotations**. $t(x_1, \dots, x_n)$ stands for any term in which x_1, \dots, x_n occur, and $A(x_1, \dots, x_n)$ stands for any atom in which x_1, \dots, x_n occur. This is why we used letters t (term) and A (atom) here instead of f (function) and P (predicate).

And in this context, the notation $t(y_1, \dots, y_n)$ stands for the term obtained from $t(x_1, \dots, x_n)$ by replacing all occurrences of x_1 with y_1 and so forth. In analogy the notation $A(y_1, \dots, y_n)$ is defined.

Note that in the schematic formulation of the rule, we use letters x and y to suggest variables, but the rule applies to arbitrary terms.

This description is not very formal, but this is not too problematic since we will be more formal **once we have some useful machinery for this at hand.**

Soundness of Equivalence Rules

On the semantic level, two things are equal if they are identical.

Semantic equality is an **equivalence relation**. This semantic fact is so fundamental that we cannot explain it any further.

So one can prove that $I_{\mathcal{A}}(s = s) = 1$ for all all terms s , because $I_{\mathcal{A}}(s) = I_{\mathcal{A}}(s)$ for all terms, and likewise for symmetry and transitivity.

Soundness of Congruence Rules

If $t(x)$ is a term containing x and $t(y)$ is the term obtained from $t(x)$ by replacing all occurrences of x with y , and moreover $I_{\mathcal{A}}(x = y) = 1$, then $I_{\mathcal{A}}(x) = I_{\mathcal{A}}(y)$. One can show by induction on the structure of t that $I_{\mathcal{A}}(t(x)) = I_{\mathcal{A}}(t(y))$.

So by “truth-functional” we mean that the value $I_{\mathcal{A}}(t(x))$ depends on $I_{\mathcal{A}}(x)$, not on x itself.

This can be generalized to n variables as in the rule.

An analogous proof can be done for rule $cong_2$.

Replacing Some Occurrences

The notation $t[z_1 \leftarrow x_1, \dots, z_n \leftarrow x_n]$ stands for the term obtained from t by simultaneously replacing each z_i ($i \in \{1, \dots, n\}$) with x_i .

$[z_1 \leftarrow x_1, \dots, z_n \leftarrow x_n]$ is called a **substitution**.

To have an unambiguous notation for “replacing some occurrences of x_1, \dots, x_n ”, we start from a term t containing variable occurrences z_1, \dots, z_n . On the LHS, these are replaced with x_1, \dots, x_n , on the RHS they are replaced with y_1, \dots, y_n . So on the RHS we have a term obtained from the one on the LHS by replacing some occurrences of x_1, \dots, x_n with y_1, \dots, y_n .

One can say that the z_1, \dots, z_n are introduced to **mark** the occurrences of x_1, \dots, x_n that should be replaced by y_1, \dots, y_n .

Note that in the schematic formulation of the rule, we use letters x and y to suggest variables, but the rule applies to arbitrary terms. The z 's

however are variables (substitutions replace variables, not arbitrary terms).

Example: $x^2 + y^2 > 12 \cdot x$

The atom $x^2 + y^2 > 12 \cdot x$ contains two occurrences of x . There are four ways to choose some occurrences of x in $x^2 + y^2 > 12 \cdot x$.

Each of those ways corresponds to an atom obtained from $x^2 + y^2 > 12 \cdot x$ by replacing some occurrences of x with z . That is, there are four different A 's such that $A[x/z] = x^2 + y^2 > 12 \cdot x$. Now the atom above the line in the examples is obtained by substituting x for z , and the atom below the line is obtained by substituting y for z .

Isabelle Rule

The Isabelle FOL rule is:

$$\frac{x = y \quad P(x)}{P(y)} \textit{subst}$$

In this rule, P is an Isabelle **metavariable**.

Why doesn't the Isabelle rule contain a z to **mark** which occurrences should be replaced?

We cannot understand this **yet**, but think of P as a formula where some positions are marked in such a way that once we apply P to t (we write $P(t)$), t will be inserted into all those positions. This is why $P(x)$ is a formula and $P(y)$ is a formula obtained by replacing some occurrences of x with y .

First-Order Theories

Example 1: Partial Orders

- The language of the theory of partial orders: \leq

Example 1: Partial Orders

- The language of the theory of partial orders: \leq
- Axioms

$$\forall x, y, z. x \leq y \wedge y \leq z \rightarrow x \leq z$$

$$\forall x, y. x \leq y \wedge y \leq x \leftrightarrow x = y$$

Example 1: Partial Orders

- The language of the theory of partial orders: \leq
- Axioms

$$\forall x, y, z. x \leq y \wedge y \leq z \rightarrow x \leq z$$

$$\forall x, y. x \leq y \wedge y \leq x \leftrightarrow x = y$$

- Alternative to axioms is to convert to rules

$$\frac{x \leq y \quad y \leq z}{x \leq z} \text{trans} \quad \frac{x \leq y \quad y \leq x}{x = y} \text{antisym} \quad \frac{x = y}{x \leq y} \leq\text{-refl}$$

Such a conversion is possible since implication is the main connective.

A Second Transitivity Rule

One may also consider adding the rule

$$\frac{x = y}{y \leq x} \leq\text{-refl2}$$

to the system. This rule can be **derived** as follows:

$$\frac{\frac{x = y}{y = x} \text{sym}}{y \leq x} \leq\text{-refl}$$

More on Orders

- A partial order \leq is a linear or total order when

$$\forall x, y. x \leq y \vee y \leq x$$

Note: no “pure” rule formulation of this disjunction.

More on Orders

- A partial order \leq is a linear or total order when

$$\forall x, y. x \leq y \vee y \leq x$$

Note: no “pure” rule formulation of this disjunction.

- A total order \leq is dense when, in addition

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

What does $<$ mean?

Structures for Orders . . .

Give **structures** for orders that are . . .

1. partial but not total:

Structures for Orders . . .

Give **structures** for orders that are . . .

1. partial but not total: \subseteq -relation;
2. total but not dense:

Structures for Orders . . .

Give **structures** for orders that are . . .

1. partial but not total: \subseteq -relation;
2. total but not dense: integers with \leq ;
3. dense:

Structures for Orders . . .

Give **structures** for orders that are . . .

1. partial but not total: \subseteq -relation;
2. total but not dense: integers with \leq ;
3. dense: reals with \leq .

Example 2: Groups

- Language: Function symbols \cdot , $^{-1}$, e

Example 2: Groups

- Language: Function symbols $_ \cdot _$, $_^{-1}$, e
- A **group** is a model of

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

It is an example of an **equational theory**.

Example 2: Groups

- Language: Function symbols $_ \cdot _$, $_^{-1}$, e
- A **group** is a model of

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

It is an example of an **equational theory**.

Two theorems: (1) $x^{-1} \cdot x = e$ and (2) $e \cdot x = x$

We will now prove them.

Theorem 1

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$x^{-1} \cdot x = e \quad (1)$$

$$x^{-1} \cdot x =$$

Theorem 1

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$x^{-1} \cdot x = e \quad (1)$$

$$x^{-1} \cdot x = x^{-1} \cdot (x \cdot e)$$

Theorem 1

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$x^{-1} \cdot x = e \quad (1)$$

$$x^{-1} \cdot x = x^{-1} \cdot (x \cdot e)$$

Theorem 1

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$x^{-1} \cdot x = e \quad (1)$$

$$x^{-1} \cdot x = x^{-1} \cdot (x \cdot e) = x^{-1} \cdot (x \cdot (x^{-1} \cdot x^{-1^{-1}}))$$

Theorem 1

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$x^{-1} \cdot x = e \quad (1)$$

$$x^{-1} \cdot x = x^{-1} \cdot (x \cdot e) = x^{-1} \cdot (x \cdot (x^{-1} \cdot x^{-1^{-1}}))$$

Theorem 1

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$x^{-1} \cdot x = e \quad (1)$$

$$\begin{aligned} x^{-1} \cdot x &= x^{-1} \cdot (x \cdot e) = x^{-1} \cdot (x \cdot (x^{-1} \cdot x^{-1^{-1}})) = \\ &= x^{-1} \cdot ((x \cdot x^{-1}) \cdot x^{-1^{-1}}) \end{aligned}$$

Theorem 1

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$x^{-1} \cdot x = e \quad (1)$$

$$\begin{aligned} x^{-1} \cdot x &= x^{-1} \cdot (x \cdot e) = x^{-1} \cdot (x \cdot (x^{-1} \cdot x^{-1^{-1}})) = \\ &= x^{-1} \cdot ((x \cdot x^{-1}) \cdot x^{-1^{-1}}) \end{aligned}$$

Theorem 1

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$x^{-1} \cdot x = e \quad (1)$$

$$\begin{aligned} x^{-1} \cdot x &= x^{-1} \cdot (x \cdot e) = x^{-1} \cdot (x \cdot (x^{-1} \cdot x^{-1^{-1}})) = \\ &= x^{-1} \cdot ((x \cdot x^{-1}) \cdot x^{-1^{-1}}) = x^{-1} \cdot (e \cdot x^{-1^{-1}}) \end{aligned}$$

Theorem 1

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$x^{-1} \cdot x = e \quad (1)$$

$$\begin{aligned} x^{-1} \cdot x &= x^{-1} \cdot (x \cdot e) = x^{-1} \cdot (x \cdot (x^{-1} \cdot x^{-1^{-1}})) = \\ &= x^{-1} \cdot ((x \cdot x^{-1}) \cdot x^{-1^{-1}}) = x^{-1} \cdot (e \cdot x^{-1^{-1}}) \end{aligned}$$

Theorem 1

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$x^{-1} \cdot x = e \quad (1)$$

$$\begin{aligned} x^{-1} \cdot x &= x^{-1} \cdot (x \cdot e) = x^{-1} \cdot (x \cdot (x^{-1} \cdot x^{-1^{-1}})) = \\ &= x^{-1} \cdot ((x \cdot x^{-1}) \cdot x^{-1^{-1}}) = x^{-1} \cdot (e \cdot x^{-1^{-1}}) = \\ &= (x^{-1} \cdot e) \cdot x^{-1^{-1}} \end{aligned}$$

Theorem 1

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$x^{-1} \cdot x = e \quad (1)$$

$$\begin{aligned} x^{-1} \cdot x &= x^{-1} \cdot (x \cdot e) = x^{-1} \cdot (x \cdot (x^{-1} \cdot x^{-1^{-1}})) = \\ &= x^{-1} \cdot ((x \cdot x^{-1}) \cdot x^{-1^{-1}}) = x^{-1} \cdot (e \cdot x^{-1^{-1}}) = \\ &= (x^{-1} \cdot e) \cdot x^{-1^{-1}} \end{aligned}$$

Theorem 1

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$x^{-1} \cdot x = e \quad (1)$$

$$\begin{aligned} x^{-1} \cdot x &= x^{-1} \cdot (x \cdot e) = x^{-1} \cdot (x \cdot (x^{-1} \cdot x^{-1-1})) = \\ &= x^{-1} \cdot ((x \cdot x^{-1}) \cdot x^{-1-1}) = x^{-1} \cdot (e \cdot x^{-1-1}) = \\ &= (x^{-1} \cdot e) \cdot x^{-1-1} = x^{-1} \cdot x^{-1-1} \end{aligned}$$

Theorem 1

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$x^{-1} \cdot x = e \quad (1)$$

$$\begin{aligned} x^{-1} \cdot x &= x^{-1} \cdot (x \cdot e) = x^{-1} \cdot (x \cdot (x^{-1} \cdot x^{-1^{-1}})) = \\ &= x^{-1} \cdot ((x \cdot x^{-1}) \cdot x^{-1^{-1}}) = x^{-1} \cdot (e \cdot x^{-1^{-1}}) = \\ &= (x^{-1} \cdot e) \cdot x^{-1^{-1}} = \textcolor{red}{x^{-1}} \cdot \textcolor{red}{x^{-1^{-1}}} \end{aligned}$$

Theorem 1

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$x^{-1} \cdot x = e \quad (1)$$

$$\begin{aligned} x^{-1} \cdot x &= x^{-1} \cdot (x \cdot e) = x^{-1} \cdot (x \cdot (x^{-1} \cdot x^{-1^{-1}})) = \\ x^{-1} \cdot ((x \cdot x^{-1}) \cdot x^{-1^{-1}}) &= x^{-1} \cdot (e \cdot x^{-1^{-1}}) = \\ (x^{-1} \cdot e) \cdot x^{-1^{-1}} &= x^{-1} \cdot x^{-1^{-1}} = e \end{aligned}$$

Theorem 1

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$x^{-1} \cdot x = e \quad (1)$$

$$\begin{aligned} x^{-1} \cdot x &= x^{-1} \cdot (x \cdot e) = x^{-1} \cdot (x \cdot (x^{-1} \cdot x^{-1^{-1}})) = \\ x^{-1} \cdot ((x \cdot x^{-1}) \cdot x^{-1^{-1}}) &= x^{-1} \cdot (e \cdot x^{-1^{-1}}) = \\ (x^{-1} \cdot e) \cdot x^{-1^{-1}} &= x^{-1} \cdot x^{-1^{-1}} = e. \end{aligned}$$

Theorem 2

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$e \cdot x = x \quad (2)$$

$$e \cdot x$$

Theorem 2

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$e \cdot x = x \quad (2)$$

$$e \cdot x = (x \cdot x^{-1}) \cdot x$$

Theorem 2

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$e \cdot x = x \quad (2)$$

$$e \cdot x = (x \cdot x^{-1}) \cdot x$$

Theorem 2

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$e \cdot x = x \quad (2)$$

$$e \cdot x = (x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x)$$

Theorem 2

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$e \cdot x = x \quad (2)$$

$$e \cdot x = (x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x) \quad (\text{Theorem 1})$$

Theorem 2

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$e \cdot x = x \quad (2)$$

$$e \cdot x = (x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x) = x \cdot e$$

Theorem 2

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$e \cdot x = x \quad (2)$$

$$e \cdot x = (x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x) = x \cdot e$$

Theorem 2

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$e \cdot x = x \quad (2)$$

$$e \cdot x = (x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x) = x \cdot e = x$$

Theorem 2

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$e \cdot x = x \quad (2)$$

$$e \cdot x = (x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x) = x \cdot e = x.$$

Lessons Learned from these Examples

Equational proofs are often tricky!

- Equalities used in different directions, “eureka” terms, etc.
- In some cases (the word problem is) decidable.

Equational versus ND Proofs

- Above proofs were of a particular, equational form.

Equational versus ND Proofs

- Above proofs were of a particular, **equational form**.
- In Isabelle this is accomplished by term rewriting.
Term rewriting is a process for replacing equals by equals
(**see later**).

Equational versus ND Proofs

- Above proofs were of a particular, **equational form**.
- In Isabelle this is accomplished by term rewriting.
Term rewriting is a process for replacing equals by equals (**see later**).
- Alternative is **natural deduction**:
 - requires explicit proofs using equality rules;
 - tedious in practice. **Try it on above examples! ►**

More Detailed Explanations

Theories

Recall our [intuitive explanation of theories](#).

A theory involves certain function and/or predicate symbols for which certain “laws” hold.

Depending on the context, these symbols may co-exist with other symbols.

Technically, the laws are added as rules (in particular, axioms) to the [proof system](#).

A [structure](#) in which these rules are true is then called a [model](#) of the rules.

Partial Orders

A partial order is a binary relation that is reflexive, transitive, and anti-symmetric: $a \leq b$ and $b \leq a$ implies $a = b$.

A Language Consisting of \leq ?

\leq is (by convention) a binary infix predicate symbol.

The theory of **partial orders** involves only this symbol, but that does not mean that there could not be any other symbols in the context.

Antisymmetry and Reflexivity

Note that $\forall x, y. x \leq y \wedge y \leq x \leftrightarrow x = y$ encodes both antisymmetry (\rightarrow) and reflexivity (\leftarrow). Recall that $A \leftrightarrow B$ as shorthand for $A \rightarrow B \wedge B \rightarrow A$.

Transitivity

The axiom $\forall x, y, z. x \leq y \wedge y \leq z \rightarrow x \leq z$ encodes **transitivity**.

Axioms vs. Rules

One can see that using $\rightarrow-I$ and $\rightarrow-E$, one can always convert a proof using the axioms to one using the **proper** rules.

More generally, an axiom of the form $\forall x_1, \dots, x_n. A_1 \wedge \dots \wedge A_n \rightarrow B$ can be converted to a rule

$$\frac{A_1 \quad \dots \quad A_n}{B} .$$

Do it in Isabelle!

Linear and Dense Orders

We define these notions according to usual mathematical terminology.

A **partial order** \leq is a **linear** or **total** order if for all a, b , either $a \leq b$ or $b \leq a$.

A **partial order** \leq is **dense** if for all a, b where $a < b$, there exists a c such that $a < b$ and $b < c$.

“Pure” Rule Formulation

The axiom $\forall x, y. x \leq y \vee y \leq x$ cannot be phrased as a **proper** rule in the style of, for example, the **transitivity axiom**.

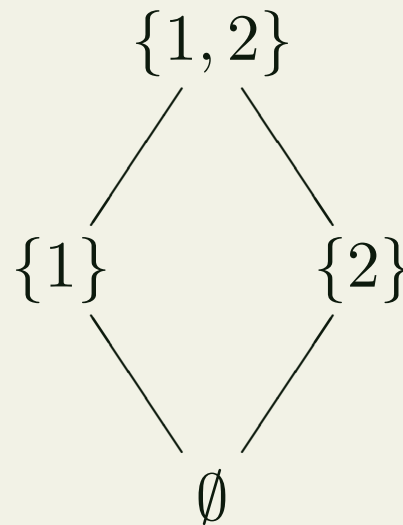
<

We use $s < t$ as shorthand for $s \leq t \wedge \neg s = t$.

We say that $<$ is the **strict** part of the **partial order** \leq .

The \subseteq -Relation

The \subseteq -relation is partial but not total. As an example, consider the \subseteq -relation on the set of subsets of $\{1, 2\}$.



Depicting **partial orders** by a such a graph is quite common. Here, node a is below node b and connected by an arc if and only if $a < b$ and there exists no c with $a < c < b$.

In this example, we have the **partial order**

$$\{(\emptyset, \emptyset), (\{1\}, \{1\}), (\{1\}, \{1\}), (\{1, 2\}, \{1, 2\}), \\ (\emptyset, \{1\}), (\emptyset, \{1\}), (\{1\}, \{1, 2\}), (\{1\}, \{1, 2\})\}.$$

Group Language

$_ \cdot _$ is a binary infix function symbol (in fact, only \cdot is the symbol, but the notation $_ \cdot _$ is used to indicate the fact that the symbol stands between its arguments).

$_^{-1}$ is a unary function symbol written as superscript. Again, the $_$ is used to indicate where the argument goes.

e is a nullary function symbol (= constant).

Note that groups are very common in mathematics, and many different notations, i.e., function names and fixity (infix, prefix. . .) are used for them.

Group

In general mathematical terminology, a **group** consists of three function symbols \cdot , $^{-1}$, e , obeying the following laws:

Associativity $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all a, b, c ,

Right neutral $a \cdot e = a$ for all a ,

Right inverse $a \cdot a^{-1} = e$ for all a .

Equational Theory

An **equational theory** is a set of equations. Each equation is an axiom. Sometimes, each equation is surrounded by several \forall -quantifiers binding all the free variables in the equation, but often the equation is regarded as implicitly universally quantified.

More generally, a **conditional equational theory** consists of **proper** rules where the premises are called **conditions** [Höl90].

Note also that sometimes, one also considers the **basic rules of equality** as being part of every equational theory. Whenever one has an equational theory, one implies that the basic rules are present; whether or not one assumes that they are formally elements of the equational theory is just a technical detail.

A Model a Group?

A **model** of the group axioms is a **structure** in which the group axioms are true.

However, when we say something like, “this model **is** a group”, then this is a slight abuse of terminology, since there may be other function symbols around that are also interpreted by the structure.

So when we say “this model **is** a group”, we mean, “this model is a model of the group axioms for function symbols \cdot , $^{-1}$, and e clear from the context”.

“Eureka” terms

By “eureka” terms we mean terms that have to be guessed in order to find a proof. At least at first sight, it seems like these terms simply fall from the sky.

The Greek **heureka** is 1st person singular perfect of **heuriskō**, “to find”. It was exclaimed by Archimedes upon discovering how to test the purity of Hiero’s crown.

The Word Problem

The word problem w.r.t. an equational theory (here: the group axioms) is the problem of deciding whether two terms s and t are equal in the theory, that is to say, whether the formula $s = t$ is true in any **model** of the theory.

Equational Proofs

An equational proof consists simply of a sequence of equations, written as $t_1 = t_2 = \dots = t_n$, where each t_{i+1} is obtained from t_i by replacing some subterm s with a term s' , provided the equality $s = s'$ holds.

This style of proof can be justified by the rules given for equality, in particular the **congruences**. However, it looks very different from the **natural deduction style**.

Proof of Theorem 2 by Natural Deduction

$$e \cdot x = x$$

Most steps use the congruence rule *cong*₂.

Proof of Theorem 2 by Natural Deduction

$$\frac{x \cdot e = x \quad e \cdot x = x \cdot e}{e \cdot x = x}$$

Most steps use the congruence rule *cong*₂.

Proof of Theorem 2 by Natural Deduction

r-neutr

$$x \cdot e = x$$

$$e \cdot x = x \cdot e$$

$$e \cdot x = x$$

Most steps use the congruence rule *cong*₂.

Proof of Theorem 2 by Natural Deduction

r-neutr	$\frac{}{x^{-1} \cdot x = e}$	$\frac{}{e \cdot x = x \cdot (x^{-1} \cdot x)}$
$x \cdot e = x$	$e \cdot x = x \cdot e$	
	$e \cdot x = x$	

Most steps use the congruence rule *cong*₂.

Proof of Theorem 2 by Natural Deduction

r-neutr	$\frac{\text{Theorem 1}}{x^{-1} \cdot x = e}$	<hr style="border: 0; border-top: 1px solid black; margin-bottom: 10px;"/> $e \cdot x = x \cdot (x^{-1} \cdot x)$ <hr style="border: 0; border-top: 1px solid black; margin-top: 10px;"/>
$x \cdot e = x$ <hr style="border: 0; border-top: 1px solid black; margin-top: 10px;"/>	$e \cdot x = x \cdot e$ <hr style="border: 0; border-top: 1px solid black; margin-top: 10px;"/>	
$e \cdot x = x$		

Most steps use the congruence rule *cong*₂.

Proof of Theorem 2 by Natural Deduction

r-neutr	Theorem 1 $x^{-1} \cdot x = e$	$\frac{(x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x)}{e \cdot x = x \cdot (x^{-1} \cdot x)}$	$\frac{e \cdot x = (x \cdot x^{-1}) \cdot x}{e \cdot x = x \cdot e}$
$x \cdot e = x$	$e \cdot x = x$		

Most steps use the congruence rule *cong*₂.

Proof of Theorem 2 by Natural Deduction

$$\begin{array}{c}
 \boxed{\text{r-neutr}} \quad \frac{x \cdot e = x}{} \\
 \text{Theorem 1} \quad \frac{x^{-1} \cdot x = e}{\phantom{x^{-1} \cdot x = e}} \\
 \boxed{\text{assoc}} \quad \frac{(x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x) \quad e \cdot x = (x \cdot x^{-1}) \cdot x}{e \cdot x = x \cdot (x^{-1} \cdot x)} \\
 \hline
 x \cdot e = x \quad e \cdot x = x \cdot e \\
 \hline
 e \cdot x = x
 \end{array}$$

Most steps use the congruence rule cong_2 .

Proof of Theorem 2 by Natural Deduction

	r-neutr	Theorem 1	assoc	cont. below
$x \cdot e = x$	$x^{-1} \cdot x = e$	$(x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x)$	$e \cdot x = (x \cdot x^{-1}) \cdot x$	
		$e \cdot x = x \cdot (x^{-1} \cdot x)$		
		$e \cdot x = x \cdot e$		
	$e \cdot x = x$			

Most steps use the congruence rule *cong*₂.

$$e \cdot x = (x \cdot x^{-1}) \cdot x$$

$$\frac{\frac{}{e = x \cdot x^{-1}} \text{sym} \quad \frac{}{e \cdot x = e \cdot x} \text{refl}}{e \cdot x = (x \cdot x^{-1}) \cdot x}$$

$$\frac{\boxed{\text{r-inv}}}{\frac{x \cdot x^{-1} = e}{e = x \cdot x^{-1}} \text{sym} \quad \frac{}{e \cdot x = e \cdot x} \text{refl}}{e \cdot x = (x \cdot x^{-1}) \cdot x}$$

Proof of Theorem 2 by Natural Deduction, Complete

$$\begin{array}{c}
 \boxed{\text{r-neutr}} \quad \frac{x \cdot e = x}{x \cdot e = x} \quad \frac{\text{Theorem 1} \quad x^{-1} \cdot x = e}{x^{-1} \cdot x = e} \quad \frac{\boxed{\text{assoc}} \quad (x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x)}{(x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x)} \\
 \frac{x \cdot e = x \quad x^{-1} \cdot x = e \quad (x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x)}{e \cdot x = x \cdot e} \quad \frac{\boxed{\text{r-inv}} \quad (x \cdot x^{-1}) = e}{(x \cdot x^{-1}) = e} \quad \frac{e = (x \cdot x^{-1}) \quad \text{sym} \quad e \cdot x = e \cdot x}{e \cdot x = (x \cdot x^{-1}) \cdot x} \quad \frac{e \cdot x = (x \cdot x^{-1}) \cdot x \quad \text{refl}}{e \cdot x = x}
 \end{array}$$

Each framed box in the derivation tree stands for a sub-tree consisting of a **group axiom** and possibly several applications of \forall -E.

Naïve Set Theory

Naïve Set Theory: Basics

- A **set** is a collection of objects where order and repetition are unimportant.

Sets are central in mathematical reasoning [Vel94]. E.g., set of prime numbers.

Naïve Set Theory: Basics

- A **set** is a collection of objects where order and repetition are unimportant.

Sets are central in mathematical reasoning [Vel94]. E.g., set of prime numbers.

- In what follows we consider a simple, intuitive formalization: “naïve set theory”.

We will be somewhat less formal than usual. Our goal is to understand standard mathematical practice.

Later, in **HOL**, we will be completely formal.

Sets: Language

Assuming any first-order language with equality, we add:

- **set-comprehension** $\{x|P(x)\}$ and a binary **membership predicate** \in .

Sets: Language

Assuming any first-order language with equality, we add:

- **set-comprehension** $\{x|P(x)\}$ and a binary **membership predicate** \in .
- **Term/formula distinction inadequate**: need a syntactic category for sets.

Sets: Language

Assuming any first-order language with equality, we add:

- **set-comprehension** $\{x|P(x)\}$ and a binary **membership predicate** \in .
- **Term/formula distinction inadequate**: need a syntactic category for sets.
- We will be more formal about syntax later (**HOL**).

Sets: Language

Assuming any first-order language with equality, we add:

- **set-comprehension** $\{x|P(x)\}$ and a binary **membership predicate** \in .
- **Term/formula distinction inadequate**: need a syntactic category for sets.
- We will be more formal about syntax later (**HOL**).
- Comprehension is a binding operator: x **bound in** $\{x|P(x)\}$.

Examples

- $\forall x. x \in \{y | y \bmod 6 = 0\} \rightarrow (x \bmod 2 = 0 \wedge x \bmod 3 = 0).$

Examples

- $\forall x. x \in \{y \mid y \bmod 6 = 0\} \rightarrow (x \bmod 2 = 0 \wedge x \bmod 3 = 0).$
- What does the following say?

$$2 \in \{w \mid 6 \notin \{x \mid x \text{ is divisible by } w\}\}$$

Examples

- $\forall x. x \in \{y | y \bmod 6 = 0\} \rightarrow (x \bmod 2 = 0 \wedge x \bmod 3 = 0).$
- What does the following say?

$$2 \in \{w | 6 \notin \{x | x \text{ is divisible by } w\}\}$$

Answer: $6 \notin \{x | x \text{ divisible by } 2\}$ i.e., 6 not divisible by 2.

Proof Rules for Sets

Introduction, elimination, extensional equality

$$\frac{P(t)}{t \in \{x | P(x)\}} \text{compr-I} \quad \frac{t \in \{x | P(x)\}}{P(t)} \text{compr-E}$$

$$\frac{\forall x. x \in A \leftrightarrow x \in B}{A = B} =-I \quad \frac{A = B}{\forall x. x \in A \leftrightarrow x \in B} =-E$$

The following equivalence is derivable:

$$\forall x. P(x) \leftrightarrow x \in \{y | P(y)\}$$

Digression: Sorted Reasoning

- In mathematical arguments we often (implicitly) assume that variables are restricted to some **universe of discourse**.
E.g., $x^2 < 9$ (universe either \mathbb{R} , \mathbb{N} , . . .)
- To **avoid ambiguity** we can include sort information in formulae:

members x of U where $P(x) \equiv \{x \in U \mid P(x)\}$

Formally

$$\{x \in U \mid P(x)\} \equiv \{x \mid U(x) \wedge P(x)\}.$$

Sorted Reasoning in an Unsorted Logic

We may introduce the additional set comprehension syntax $\{x \in U \mid P(x)\}$, but our logic is still **unsorted**. We have

$$y \in \{x \in U \mid P(x)\} \leftrightarrow y \in \{x \mid U(x) \wedge P(x)\} \leftrightarrow U(y) \wedge P(y)$$

Sorted Quantification

$$\forall x \in U. P(x) \equiv \forall x. U(x) \rightarrow P(x)$$

$$\exists x \in U. P(x) \equiv \exists x. U(x) \wedge P(x)$$

Operations on Sets

- Functions on sets

$$A \cap B \equiv \{x | x \in A \wedge x \in B\}$$

$$A \cup B \equiv \{x | x \in A \vee x \in B\}$$

$$A \setminus B \equiv \{x | x \in A \wedge x \notin B\}$$

- Predicates on sets

$$A \subseteq B \equiv \forall x. x \in A \rightarrow x \in B$$

Examples of Operations on Sets

One often depicts sets as circles or bubbles.

What are $A \cap B$, $A \cup B$, $A \setminus B$?

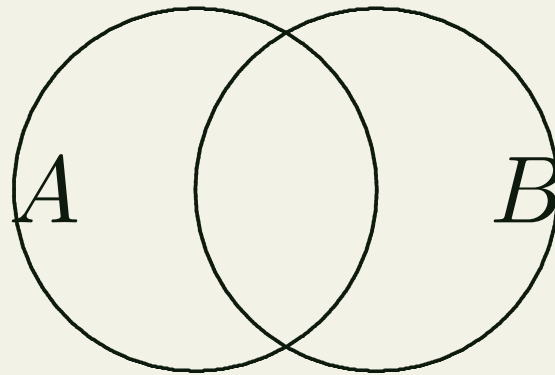
A

B

Examples of Operations on Sets

One often depicts sets as circles or bubbles.

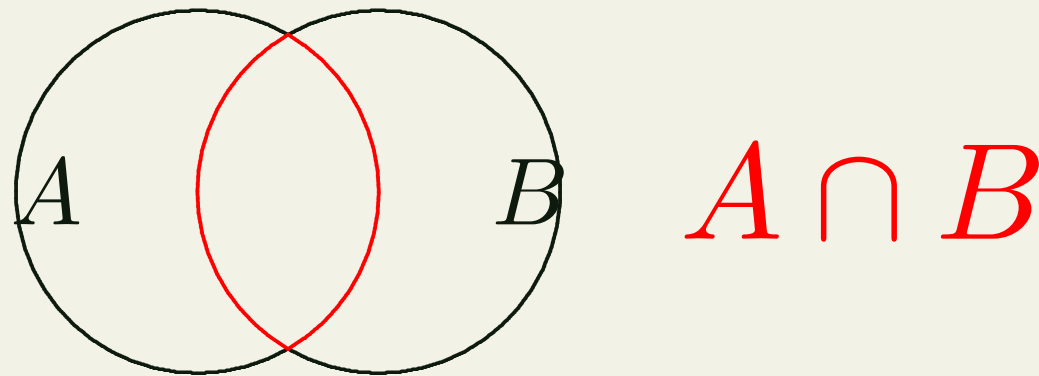
What are $A \cap B$, $A \cup B$, $A \setminus B$?



Examples of Operations on Sets

One often depicts sets as circles or bubbles.

What are $A \cap B$, $A \cup B$, $A \setminus B$?

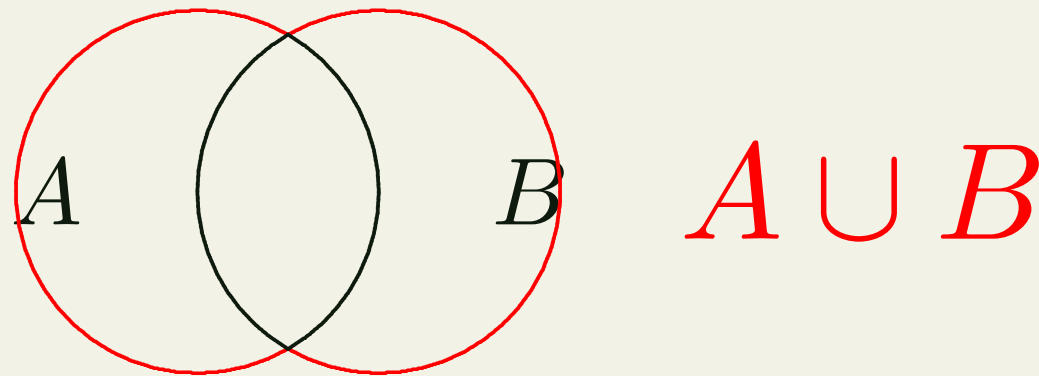


$A \cap B$

Examples of Operations on Sets

One often depicts sets as circles or bubbles.

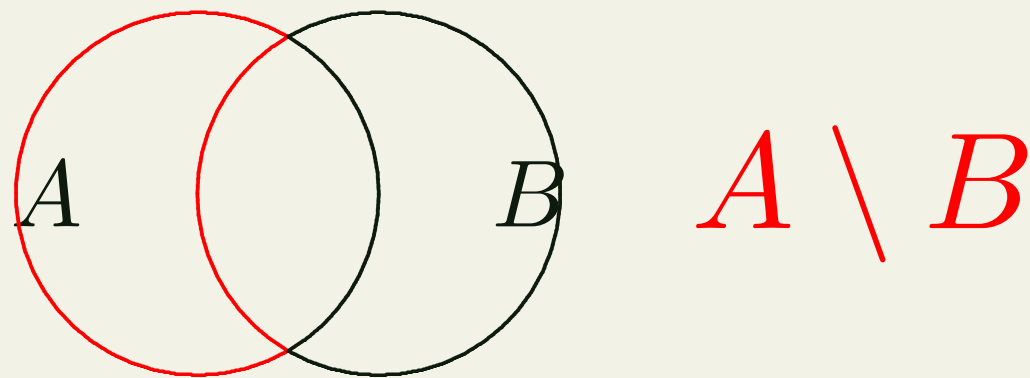
What are $A \cap B$, $A \cup B$, $A \setminus B$?



Examples of Operations on Sets

One often depicts sets as circles or bubbles.

What are $A \cap B$, $A \cup B$, $A \setminus B$?



Correspondence between Set-Theoretic and Logical Operators

$$x \in A \cap B \iff x \in A \wedge x \in B$$

$$x \in A \cup B \iff x \in A \vee x \in B$$

$$x \in A \setminus B \iff x \in A \wedge x \notin B$$

These correspondences follow from the definitions of the set-theoretic operators and $\forall x. P(x) \iff x \in \{y | P(y)\}$.

Correspondence between Set-Theoretic and Logical Operators

$$x \in A \cap B \leftrightarrow x \in A \wedge x \in B$$

$$x \in A \cup B \leftrightarrow x \in A \vee x \in B$$

$$x \in A \setminus B \leftrightarrow x \in A \wedge x \notin B$$

These correspondences follow from the definitions of the set-theoretic operators and $\forall x. P(x) \leftrightarrow x \in \{y | P(y)\}$.

Example: what is the logical form of $x \in ((A \cap B) \cup (A \cap C))$?

Correspondence between Set-Theoretic and Logical Operators

$$x \in A \cap B \leftrightarrow x \in A \wedge x \in B$$

$$x \in A \cup B \leftrightarrow x \in A \vee x \in B$$

$$x \in A \setminus B \leftrightarrow x \in A \wedge x \notin B$$

These correspondences follow from the definitions of the set-theoretic operators and $\forall x. P(x) \leftrightarrow x \in \{y | P(y)\}$.

Example: what is the logical form of

$$x \in ((A \cap B) \cup (A \cap C))?$$

$$(x \in A \wedge x \in B) \vee (x \in A \wedge x \in C)$$

Proof of $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

Proof of $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ (1)

Venn diagram (Is this a proof?)

Proof of $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ (2)

Natural deduction (natural language)

Proof of $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ (2)

Natural deduction (**natural language**)

By **extensionality**, suffices to show

$$\forall x. x \in A \cap (B \cup C) \leftrightarrow x \in (A \cap B) \cup (A \cap C).$$

Proof of $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ (2)

Natural deduction (natural language)

By **extensionality**, suffices to show

$$\forall x. x \in A \cap (B \cup C) \leftrightarrow x \in (A \cap B) \cup (A \cap C).$$

For an arbitrary x , this is equivalent to establishing

$$\begin{aligned} (x \in A \wedge (x \in B \vee x \in C)) &\leftrightarrow \\ (x \in A \wedge x \in B) \vee (x \in A \wedge x \in C) \end{aligned}$$

Proof of $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ (2)

Natural deduction (natural language)

By **extensionality**, suffices to show

$$\forall x. x \in A \cap (B \cup C) \leftrightarrow x \in (A \cap B) \cup (A \cap C).$$

For an arbitrary x , this is equivalent to establishing

$$\begin{aligned} (x \in A \wedge (x \in B \vee x \in C)) &\leftrightarrow \\ (x \in A \wedge x \in B) \vee (x \in A \wedge x \in C) \end{aligned}$$

But that is a propositional tautology.

Same in Isabelle

Last proof carries over to Isabelle: extensionality, rewriting, tautology checking. Do it!

Prove: for all Sets A and B , $((A \cup B) \setminus B) \subseteq A$

Not obvious? Just follow your nose!

Prove: for all Sets A and B , $((A \cup B) \setminus B) \subseteq A$

Not obvious? Just follow your nose!

Let A and B be arbitrary sets.

Prove: for all Sets A and B , $((A \cup B) \setminus B) \subseteq A$

Not obvious? Just follow your nose!

Let A and B be arbitrary sets.

Let x be element of $(A \cup B) \setminus B$.

Prove: for all Sets A and B , $((A \cup B) \setminus B) \subseteq A$

Not obvious? Just follow your nose!

Let A and B be arbitrary sets.

Let x be element of $(A \cup B) \setminus B$.

So $(x \in A \vee x \in B) \wedge \neg x \in B$.

Prove: for all Sets A and B , $((A \cup B) \setminus B) \subseteq A$

Not obvious? Just follow your nose!

Let A and B be arbitrary sets.

Let x be element of $(A \cup B) \setminus B$.

So $(x \in A \vee x \in B) \wedge \neg x \in B$.

Therefore $x \in A$.

Prove: for all Sets A and B , $((A \cup B) \setminus B) \subseteq A$

Not obvious? Just follow your nose!

Let A and B be arbitrary sets.

Let x be element of $(A \cup B) \setminus B$.

So $(x \in A \vee x \in B) \wedge \neg x \in B$.

Therefore $x \in A$.

Therefore $x \in (A \cup B) \setminus B \rightarrow x \in A$.

Prove: for all Sets A and B , $((A \cup B) \setminus B) \subseteq A$

Not obvious? Just follow your nose!

Let A and B be arbitrary sets.

Let x be element of $(A \cup B) \setminus B$.

So $(x \in A \vee x \in B) \wedge \neg x \in B$.

Therefore $x \in A$.

Therefore $x \in (A \cup B) \setminus B \rightarrow x \in A$.

Therefore $((A \cup B) \setminus B) \subseteq A$.

Prove: for all Sets A and B , $((A \cup B) \setminus B) \subseteq A$

Not obvious? Just follow your nose!

Let A and B be arbitrary sets.

Let x be element of $(A \cup B) \setminus B$.

So $(x \in A \vee x \in B) \wedge \neg x \in B$.

Therefore $x \in A$.

Therefore $x \in (A \cup B) \setminus B \rightarrow x \in A$.

Therefore $((A \cup B) \setminus B) \subseteq A$.

This semi-formal proof **combines** forward reasoning with backward reasoning. This is common in practice and usually easy to unscramble.

Extending Set Comprehensions

Recall set comprehensions $\{x \mid P(x)\}$.

Extending Set Comprehensions

Recall set comprehensions $\{x|P(x)\}$.

Now what do you think this is?

$$\{f(x)|P(x)\}$$

Extending Set Comprehensions

Recall set comprehensions $\{x|P(x)\}$.

Now what do you think this is?

$$\{f(x)|P(x)\} \equiv \{y|\exists x. P(x) \wedge y = f(x)\}$$

Extending Set Comprehensions

Recall set comprehensions $\{x|P(x)\}$.

Now what do you think this is?

$$\{f(x)|P(x)\} \equiv \{y|\exists x. P(x) \wedge y = f(x)\}$$

Example: $t \in \{x^2|x > 5\}$ equivalent to

Extending Set Comprehensions

Recall set comprehensions $\{x|P(x)\}$.

Now what do you think this is?

$$\{f(x)|P(x)\} \equiv \{y|\exists x. P(x) \wedge y = f(x)\}$$

Example: $t \in \{x^2|x > 5\}$ equivalent to $\exists x. x > 5 \wedge t = x^2$.

True for $t \in \{36, 49, \dots\}$

Indexing

Sometimes, it is natural to denote a function f applied to an argument x as “ f indexed by x ”, so f_x , rather than $f(x)$.

Indexing

Sometimes, it is natural to denote a function f applied to an argument x as “ f indexed by x ”, so f_x , rather than $f(x)$.

Example: let S = set of students and let m_s stand for “the mother of s ”, for s a student. Call S an **index set**.

$$x \in \{m_s | s \in S\} \iff x \in \{y | \exists s. s \in S \wedge y = m_s\}$$

$$\iff \exists s. s \in S \wedge x = m_s$$

$$\iff \exists s \in S. x = m_s$$

Uses extended comprehensions, indexing syntax, and sorted quantification.

Logical Forms of the New Notation

Question: what is the logical form of $\{x_i | i \in I\} \subseteq A$?

Logical Forms of the New Notation

Question: what is the logical form of $\{x_i | i \in I\} \subseteq A$?

$$\begin{aligned} &\forall x. x \in \{x_i | i \in I\} \rightarrow x \in A, \quad \text{i.e.,} \\ &\forall x. (\exists i \in I. x = x_i) \rightarrow x \in A. \end{aligned}$$

Logical Forms of the New Notation

Question: what is the logical form of $\{x_i | i \in I\} \subseteq A$?

$$\forall x. x \in \{x_i | i \in I\} \rightarrow x \in A, \quad \text{i.e.,}$$

$$\forall x. (\exists i \in I. x = x_i) \rightarrow x \in A.$$

Intuition suggests that $\forall i \in I. x_i \in A$ is also correct, i.e.,

$$(\forall x. (\exists i \in I. x = x_i) \rightarrow x \in A) \leftrightarrow (\forall i \in I. x_i \in A).$$

Proving this would be another exercise on using extended comprehensions, indexing syntax, and sorted quantification.

Indexed Families

Can formulate sets as **indexed families**.

Let S = set of students, C_s = courses taken by student s .

Then

$$\{C_s | s \in S\}$$

is the set whose elements are those sets of courses taken by some student.

Logical Forms of Powersets

$$\wp(A) = \{x \mid x \subseteq A\}.$$

What is the logical form of:

1. $x \in \wp(A)$?

Logical Forms of Powersets

$$\wp(A) = \{x \mid x \subseteq A\}.$$

What is the **logical form** of:

1. $x \in \wp(A)$?

$$x \subseteq A, \text{ i.e., } \forall y. (y \in x \rightarrow y \in A)$$

2. $\wp(A) \subseteq \wp(B)$?

Logical Forms of Powersets

$$\wp(A) = \{x \mid x \subseteq A\}.$$

What is the **logical form** of:

1. $x \in \wp(A)$?

$$x \subseteq A, \text{ i.e., } \forall y. (y \in x \rightarrow y \in A)$$

2. $\wp(A) \subseteq \wp(B)$?

$$\forall x. x \in \wp(A) \rightarrow x \in \wp(B), \text{ i.e., }$$

Logical Forms of Powersets

$$\wp(A) = \{x \mid x \subseteq A\}.$$

What is the **logical form** of:

1. $x \in \wp(A)$?

$$x \subseteq A, \text{ i.e., } \forall y. (y \in x \rightarrow y \in A)$$

2. $\wp(A) \subseteq \wp(B)$?

$$\forall x. x \in \wp(A) \rightarrow x \in \wp(B), \text{ i.e.,}$$

$$\forall x. x \subseteq A \rightarrow x \subseteq B, \text{ i.e.,}$$

Logical Forms of Powersets

$$\wp(A) = \{x \mid x \subseteq A\}.$$

What is the **logical form** of:

1. $x \in \wp(A)$?

$$x \subseteq A, \text{ i.e., } \forall y. (y \in x \rightarrow y \in A)$$

2. $\wp(A) \subseteq \wp(B)$?

$$\forall x. x \in \wp(A) \rightarrow x \in \wp(B), \text{ i.e.,}$$

$$\forall x. x \subseteq A \rightarrow x \subseteq B, \text{ i.e.,}$$

$$\forall x. (\forall y. y \in x \rightarrow y \in A) \rightarrow (\forall y. y \in x \rightarrow y \in B)$$

Exercise: prove that the last answer is equivalent to $A \subseteq B$, i.e., $\forall x. x \in A \rightarrow x \in B$.

Outlook

Sets can have other sets as elements.

Implicitly assume that universe of discourse is **collection** of all sets.

Russell's Paradox

Suppose $U := \{x \mid \top\}$. Then $U \in U$.

Quite strange but no contradiction yet.

Russell's Paradox

Suppose $U := \{x \mid \top\}$. Then $U \in U$.

Quite strange but no contradiction yet.

Now split sets into two categories:

1. unusual sets like U that are elements of themselves, and
2. more typical sets that are not.

Russell's Paradox

Suppose $U := \{x \mid \top\}$. Then $U \in U$.

Quite strange but no contradiction yet.

Now split sets into two categories:

1. unusual sets like U that are elements of themselves, and
2. more typical sets that are not.

Let $R := \{A \mid A \notin A\}$.

Using **logical form** we derive: $\forall A. (A \in R \leftrightarrow A \notin A)$

Russell's Paradox

Suppose $U := \{x \mid \top\}$. Then $U \in U$.

Quite strange but no contradiction yet.

Now split sets into two categories:

1. unusual sets like U that are elements of themselves, and
2. more typical sets that are not.

Let $R := \{A \mid A \notin A\}$.

Using **logical form** we derive: $\forall A. (A \in R \leftrightarrow A \notin A)$

Substituting R for A (\forall -E) yields $R \in R \leftrightarrow R \notin R$, which is a logical contradiction. **What does this tell us about sets?**

Where Do We Go from here?

- The λ -calculus as basis for a metalanguage to avoid notational confusion

Where Do We Go from here?

- The λ -calculus as basis for a metalanguage to avoid notational confusion
- Resolution and other deduction techniques: understanding Isabelle better and achieving a higher level of automation

Where Do We Go from here?

- The λ -calculus as basis for a metalanguage to avoid notational confusion
- Resolution and other deduction techniques: understanding Isabelle better and achieving a higher level of automation
- Higher-order logic: a formalism for (among other things) non-naïve set theory ►

More Detailed Explanations

Set Comprehension

Set comprehension is a way of defining sets. $\{x|P(x)\}$ stands for the set of elements of the universe for which $P(x)$ (some formula usually containing x) holds.

Is a Set a Term?

It is more adequate to regard a set as a term than as a formula. A set is a “thing”, not a statement about “things”.

After all, we have the predicate \in expecting a set on the RHS (and even the LHS may be a set!), and predicates take terms as arguments.

However, the syntax used in set comprehensions is not legal syntax for terms, since $P(x)$ is a formula.

This is why we introduce a special syntactic category for sets.

Extensional Equality

Two things are **extensionally equal** if they are “equal in their effects”. Thus two sets are equal if they have the same members, regardless of what syntactic expressions are used to define those sets. Note that extensional equality may be undecidable.

Deriving Equivalence for Comprehensions

$$\begin{array}{c}
 \frac{[P(x)]^1}{x \in \{y|P(y)\}} \text{ compr-I} \qquad \frac{[x \in \{y|P(y)\}]^2}{P(x)} \text{ compr-E} \\
 \hline
 \frac{P(x) \rightarrow x \in \{y|P(y)\} \rightarrow -I^1 \quad x \in \{y|P(y)\} \rightarrow P(x) \rightarrow -I^2}{P(x) \rightarrow x \in \{y|P(y)\} \wedge x \in \{y|P(y)\} \rightarrow P(x)} \wedge -I \\
 \hline
 \frac{P(x) \leftrightarrow x \in \{y|P(y)\}}{\forall x. P(x) \leftrightarrow x \in \{y|P(y)\}} \text{ iff } \forall -I
 \end{array}$$

Rules $\wedge -I$, $\rightarrow -I$, $\forall -I$ were defined in previous lectures. The step marked with **iff** is not a proof step in the technical sense. We only make the expansion of a shorthand notation explicit.

Universes

We already know what a **universe** or **domain** is. To interpret a particular language, we have a **structure** interpreting all function symbols as functions on the universe.

However, it is often adequate to subdivide the universe into several “sub-universes”. Those are called **sorts**. Note that a sort is a set.

For example, in a usual mathematical context, one may distinguish \mathcal{R} (the real numbers) and \mathcal{N} (the natural numbers) to say that \sqrt{x} requires x to be of sort \mathcal{R} and $x!$ requires x to be of sort \mathcal{N} .

Avoiding Ambiguity

We want to make explicit the sort of the variable in question. So we do not want the set of all x such that $P(x)$ holds, but only the ones of the right sort, so the ones for which $x \in U$ (U being the sort/universe) holds. Note there is a certain confusion here, since we write $x \in U$ in one place (so U should be a set) and $U(x)$ in another (so U should be a predicate. This confusion is deliberate and quite common. One can identify a set (sort) U with a unary predicate U such that $U(t)$ is interpreted as *True* iff t is a member of U .

The whole expression $\{x \in U | P(x)\}$ is a special kind of syntax.

Therefore, you must look at it as a whole: it makes no sense to see any meaning just in, say, the bit $x \in U$ in this expression. It is called **set**

comprehension, and it is defined by

$$\{x \in U \mid P(x)\} \equiv \{x \mid U(x) \wedge P(x)\}.$$

Sorted Logic

In sorted logic, sorts are part of the syntax. So the **signature** contains a fixed set of sorts. For each constant, it is specified what its sort is. For each function symbol, it is specified what the sort of each argument is, and what the sort of the result is. For each predicate symbol, it is specified what the sort of each argument is.

Terms and formulas that do not respect the sorts are not well-formed, and so they are not assigned a meaning.

In contrast, our logic is unsorted. The special syntax we provide for sorted reasoning is just **syntactic sugar**, i.e., we use it as shorthand and since it has an intuitive reasoning, but it has no impact on how expressive our logic is.

For any formal language (programming language, logic, etc.), the term “syntactic sugar” refers to syntax that is provided for the sake of

readability and brevity, but which does not affect the expressiveness of the language.

It is usually a good idea to consider the language without the syntactic sugar for any theoretical considerations about the language, since it makes the language simpler and the considerations less error-prone. However, the correspondence between the syntactic sugar and the basic syntax should be stated formally.

Sorted Quantification

So $\forall x \in U. P(x)$ is simply a shorthand or **syntactic sugar** for $\forall x. x \in U \rightarrow P(x)$, and analogously for $\exists x \in U. P(x)$.

Set Functions

\cap is called **intersection**.

\cup is called **union**.

\setminus is called **set difference**.

\subseteq is called **inclusion**.

The Logical Form

When we transform an expression containing set operators $\cap, \cup, \setminus, \subseteq$ into an expression using $\wedge, \vee, \neg, \rightarrow$, we call the latter the **logical form** of the expression.

Is a Venn Diagram a Proof?

A **Venn diagram** draws sets as bubbles. Intersecting sets are drawn as overlapping bubbles, and the overlapping area is meant to depict the intersection of the sets.

A Venn diagram is not a proof in the sense defined **earlier**.

Moreover, it would not even be acceptable as a proof according to usual mathematical practice. If it is unknown whether two sets have a non-empty intersection, how are we supposed to draw them? Trying to make a case distinctions (drawing several diagrams depending on the cases) is error-prone.

Venn diagrams are useful for illustration purposes, but they are not proofs.

Natural Language

We intersperse formal notation with natural language here in order to give an intuitive and short proof.

We can also do this more formally in Isabelle.

Explanations for each Step

Let A and B be arbitrary sets.

$(\forall-I)$

Explanations for each Step

Let A and B be arbitrary sets. (\forall -I)

Let x be an element of $(A \cup B) \setminus B$ (temporary assumption)

Explanations for each Step

Let A and B be arbitrary sets.	$(\forall-I)$
Let x be an element of $(A \cup B) \setminus B$	(temporary assumption)
So $(x \in A \vee x \in B) \wedge \neg x \in B$	(equivalent proposition)

Explanations for each Step

Let A and B be arbitrary sets. (\forall -I)
Let x be an element of $(A \cup B) \setminus B$ (temporary assumption)
So $(x \in A \vee x \in B) \wedge \neg x \in B$ (equivalent proposition)
Therefore $x \in A$ (P follows from $(P \vee Q) \wedge \neg Q$)

Explanations for each Step

Let A and B be arbitrary sets.	$(\forall-I)$
Let x be an element of $(A \cup B) \setminus B$	(temporary assumption)
So $(x \in A \vee x \in B) \wedge \neg x \in B$	(equivalent proposition)
Therefore $x \in A$	(P follows from $(P \vee Q) \wedge \neg Q$)
Therefore $x \in (A \cup B) \setminus B \rightarrow x \in A$	$(\rightarrow-I)$

Explanations for each Step

Let A and B be arbitrary sets.	$(\forall-I)$
Let x be an element of $(A \cup B) \setminus B$	(temporary assumption)
So $(x \in A \vee x \in B) \wedge \neg x \in B$	(equivalent proposition)
Therefore $x \in A$	(P follows from $(P \vee Q) \wedge \neg Q$)
Therefore $x \in (A \cup B) \setminus B \rightarrow x \in A$	$(\rightarrow-I)$
Therefore $((A \cup B) \setminus B) \subseteq A$	(def of \subseteq)

Explanations for each Step

Let A and B be arbitrary sets. (\forall -I)

Let x be an element of $(A \cup B) \setminus B$ (temporary assumption)

So $(x \in A \vee x \in B) \wedge \neg x \in B$ (equivalent proposition)

Therefore $x \in A$ (P follows from $(P \vee Q) \wedge \neg Q$)

Therefore $x \in (A \cup B) \setminus B \rightarrow x \in A$ (\rightarrow -I)

Therefore $((A \cup B) \setminus B) \subseteq A$ (def of \subseteq)

Concerning forward and backwards reasoning, one may look at it as follows: we first construct the derivation step at the root of the proof tree (\forall -I), and then we jump to a leaf (by making the temporary assumption) and work downwards from there.

Definition of \subseteq

$$\{x_i | i \in I\} \subseteq A \equiv \forall x. x \in \{x_i | i \in I\} \rightarrow x \in A$$

follows from the definition of \subseteq .

Details of Logical Form

We want to show

$$\forall x. x \in \{x_i | i \in I\} \rightarrow x \in A \equiv \forall x. (\exists i \in I. x = x_i) \rightarrow x \in A$$

$x \in \{x_i i \in I\}$	\equiv	(def. of notation)
$x \in \{y \exists i. i \in I \wedge y = x_i\}$	\equiv	<i>compr-l</i>
$\exists i. i \in I \wedge x = x_i$	\equiv	(Sorted quantification)
$\exists i \in I. x = x_i$		

Intuition for Indexed Sets

It may be helpful to pronounce both forms out loud in natural language to get an intuitive feeling that they are equivalent.

Proof

Want to prove

$$(\forall x. (\exists i \in I. x = x_i) \rightarrow x \in A) \leftrightarrow (\forall i \in I. x_i \in A)$$

Proof

Want to prove

$$(\forall x. (\exists i \in I. x = x_i) \rightarrow x \in A) \leftrightarrow (\forall i \in I. x_i \in A)$$

- “ \rightarrow ”

Let $i \in I$ be arbitrary. Now from assumption (for the instance x_i) we have $(\exists j \in I. x_i = x_j) \rightarrow x_i \in A$. But premise is true for $i = j$, so $x_i \in A$.

Proof

Want to prove

$$(\forall x. (\exists i \in I. x = x_i) \rightarrow x \in A) \leftrightarrow (\forall i \in I. x_i \in A)$$

- “ \rightarrow ”

Let $i \in I$ be arbitrary. Now from assumption (for the instance x_i) we have $(\exists j \in I. x_i = x_j) \rightarrow x_i \in A$. But premise is true for $i = j$, so $x_i \in A$.

- “ \leftarrow ”

Let x be arbitrary and assume $\exists i \in I. x = x_i$. So for some $i \in I$, we have $x = x_i$. Now $\forall i \in I. x_i \in A$. Hence $x \in A$.

“ \rightarrow ” in more detail: Want to prove

$$(\forall x. (\exists i \in I. x = x_i) \rightarrow x \in A) \leftrightarrow (\forall i \in I. x_i \in A)$$

“ \rightarrow ” in more detail: Want to prove

$$(\forall x.(\exists i \in I. x = x_i) \rightarrow x \in A) \leftrightarrow (\forall i \in I. x_i \in A)$$

We show $\forall i \in I. x_i \in A$ assuming $\forall x.(\exists i \in I. x = x_i) \rightarrow x \in A$.

So we show that for **arbitrary** $i \in I$, assuming

$\forall x.(\exists i \in I. x = x_i) \rightarrow x \in A$, we have $x_i \in A$. So let $i \in I$ be arbitrary.

Since we have $\forall x.(\exists i \in I. x = x_i) \rightarrow x \in A$, by rule \forall - E we can specialize to $(\exists j \in I. x_i = x_j) \rightarrow x_i \in A$. But premise $(\exists j \in I. x_i = x_j)$ is true for $i = j$, and so $x_i \in A$, which is what was to be proven.

This proof could be made more formal by drawing a proof tree or using Isabelle.

“ \leftarrow ” in more Detail: Want to prove

$$(\forall x.(\exists i \in I. x = x_i) \rightarrow x \in A) \leftrightarrow (\forall i \in I. x_i \in A)$$

“ \rightarrow ” in more detail: Want to prove

$$(\forall x.(\exists i \in I. x = x_i) \rightarrow x \in A) \leftrightarrow (\forall i \in I. x_i \in A)$$

We show $\forall i \in I. x_i \in A$ assuming $\forall x.(\exists i \in I. x = x_i) \rightarrow x \in A$.

So we show that for **arbitrary** $i \in I$, assuming

$\forall x.(\exists i \in I. x = x_i) \rightarrow x \in A$, we have $x_i \in A$. So let $i \in I$ be arbitrary.

Since we have $\forall x.(\exists i \in I. x = x_i) \rightarrow x \in A$, by rule \forall - E we can specialize to $(\exists j \in I. x_i = x_j) \rightarrow x_i \in A$. But premise $(\exists j \in I. x_i = x_j)$ is true for $i = j$, and so $x_i \in A$, which is what was to be proven.

This proof could be made more formal by drawing a proof tree or using Isabelle.

“ \leftarrow ” in more Detail: Want to prove

$$(\forall x.(\exists i \in I. x = x_i) \rightarrow x \in A) \leftrightarrow (\forall i \in I. x_i \in A)$$

We show $\forall x. (\exists i \in I. x = x_i) \rightarrow x \in A$, assuming $\forall i \in I. x_i \in A$.

So we show that for **arbitrary** x , assuming $\forall i \in I. x_i \in A$, we have $(\exists i \in I. x = x_i) \rightarrow x \in A$. So let x be arbitrary.

To show $(\exists i \in I. x = x_i) \rightarrow x \in A$, assume $\exists i \in I. x = x_i$. So for some $i \in I$, we have $x = x_i$. Now by our earlier assumption $\forall i \in I. x_i \in A$, and so it follows that $x \in A$. thus we have shown $x \in A$ under the assumption $(\exists i \in I. x = x_i)$, thus we have shown $(\exists i \in I. x = x_i) \rightarrow x \in A$, which is what was to be proven.

This proof could be made more formal by drawing a proof tree or using Isabelle.

Families

The word **family** is sometimes used for a set of sets [Vel94].

Collections and Sets

We speak of **collection** of all sets rather than **set** of all sets in order to pretend that we are being careful since we are not sure if there is such a thing as a **set of all sets**. Therefore we use the “neutral” word **collection** whose meaning is obvious. . .

Collections and Sets

We speak of **collection** of all sets rather than **set** of all sets in order to pretend that we are being careful since we are not sure if there is such a thing as a **set of all sets**. Therefore we use the “neutral” word **collection** whose meaning is obvious. . .

Is it?

Collections and Sets

We speak of **collection** of all sets rather than **set** of all sets in order to pretend that we are being careful since we are not sure if there is such a thing as a **set of all sets**. Therefore we use the “neutral” word **collection** whose meaning is obvious. . .

Is it?

Recall that we have defined **set** as **collection of objects** in the first place. So it is rather futile to suggest now that there should be some difference between collections and sets.

The fact of the matter is: the approach of allowing arbitrary collections of “objects” and regarding such collections as “objects” themselves is **naïve**. We will see this shortly.

Logical Characterization

Recall $R := \{A \mid A \notin A\}$ and recall the notion of **logical form**.

Let A be arbitrary (for the formal reasoning applied here, **arbitrary** means: it could be a set, a number, a dog, the pope, anything whatsoever).

By the **rules for set comprehension**, we can prove

$A \in \{A \mid A \notin A\} \rightarrow A \notin A$ and $A \notin A \rightarrow A \in \{A \mid A \notin A\}$, and so by **definition of \leftrightarrow** , we have $A \in R \leftrightarrow A \notin A$, and since A was arbitrary, by **\forall -I**, we have $\forall A. (A \in R \leftrightarrow A \notin A)$.

What does this Tell us about Sets?

It tells us that there can be no such thing as the set of all sets.

The fundamental flaw of naïve set theory is in saying that a set is a collection of “objects” without worrying what an object is. If we make no restriction as to what an object is, then a set is obviously also an object. But then we effectively base the definition of the new concept **set** on the existence of sets, so the definition is circular.

The intuition for the solution to this dilemma is not difficult: A set is a collection of objects of which we are already sure that they exist. In particular, since we are only just about to define sets, these objects may not themselves be sets.

Once we have such sets, we can introduce “sets of second order”, that is, sets that contain sets of the first kind. This process can be continued ad infinitum.

The formal details will come later.

True

Assume that \top is **syntactic sugar** for a proposition that is always true, say $\top \equiv \perp \rightarrow \perp$. We have not introduced this, but it is convenient.

So **semantically**, we have $I_{\mathcal{A}}(\top) = 1$ for all $I_{\mathcal{A}}$.

A Strange Set Comprehension

Recall that a **set comprehension** has the form $\{x | P(x)\}$, where $P(x)$ is a formula usually containing x .

The set comprehension $U := \{x \mid \top\}$ is strange since \top does not contain x .

But by the **introduction rule for set comprehensions**, this means that $x \in U$ for **any** x . Thus in particular, $U \in U$.

Higher-Order Logic

Higher-order logic is a solution to the dilemma posed by Russell's paradox.

It is a surprisingly simple formalism which can be extended conservatively: this means that it can be ensured that the extensions cannot compromise the truth or falsity of statements that were already expressible before the extension.

The λ -Calculus

The λ -Calculus: Motivation

A way of writing **functions**. E.g., $\lambda x. x + 5$ is the function taking any number n to $n + 5$. Theory underlying **functional programming**.

Turing-complete model of computation.

One of the most important formalisms of (theoretical) computer science!

The λ -Calculus: Motivation

A way of writing **functions**. E.g., $\lambda x. x + 5$ is the function taking any number n to $n + 5$. Theory underlying **functional programming**.

Turing-complete model of computation.

One of the most important formalisms of (theoretical) computer science!

Why is it interesting for us? The λ -calculus is used for representing object logics in Isabelle. It is the core of Isabelle's metalogic!

Outline of this Lecture

- The untyped λ -calculus
- The simply typed λ -calculus (λ^{\rightarrow})
- An extension of the typed λ -calculus
- Higher-order unification

Untyped λ -Calculus

From functional programming , you may be familiar with **function definitions** such as

$$f\ x = x + 5$$

The λ -calculus is a formalism for writing **nameless** functions.
The function $\lambda x. x + 5$ corresponds to f .

Untyped λ -Calculus

From functional programming , you may be familiar with **function definitions** such as

$$f\ x = x + 5$$

The λ -calculus is a formalism for writing **nameless** functions.

The function $\lambda x. x + 5$ corresponds to f .

The **application** to say, 3, is written $(\lambda x. x + 5)(3)$. Its result is computed by substituting 3 for x , yielding $3 + 5$, which in usual arithmetic evaluates to 8.

Syntax

$(x \in Var, c \in Const)$

$$e ::= x \mid c \mid (ee) \mid (\lambda x. e)$$

The objects generated by this grammar are called λ -terms or simply terms.

Conventions: iterated λ & left-associated application

$$\begin{aligned} (\lambda x. (\lambda y. (\lambda z. ((xz)(yz))))) &\equiv (\lambda xyz. ((xz)(yz))) \\ &\equiv \lambda xyz. xz(yz) \end{aligned}$$

Is $\lambda x. x + 5$ a λ -term?

Substitution

- Reduction based on substitutions

$$(\lambda x. g x 3)(5) = (g x 3)[x \leftarrow 5] = g 5 3$$

- Must respect free and bound variables,

$$(\lambda x. x(\lambda x. xy))(e) = ((x(\lambda x. xy))[x \leftarrow e] = e(\lambda x. xy)$$

- Same problems as with quantifiers

$$\frac{\forall x. (P(x) \wedge \exists x. Q(x, y))}{P(e) \wedge \exists x. Q(x, y)} \forall\text{-}E \qquad \frac{\forall x. (P(x) \wedge \exists y. Q(x, y))}{P(y) \wedge \exists z. Q(y, z)} \forall\text{-}E$$

Bound, Free, Binding Occurrences

Recall the notions of **bound**, **free**, and **binding** occurrences of variables in a term. Same thing here:

λ -calculus

FOL

$FV(x) :=$

Bound, Free, Binding Occurrences

Recall the notions of **bound**, **free**, and **binding** occurrences of variables in a term. Same thing here:

λ -calculus

$$FV(x) := \{x\}$$

$$FV(c) :=$$

FOL

$$= FV(x)$$

Bound, Free, Binding Occurrences

Recall the notions of **bound**, **free**, and **binding** occurrences of variables in a term. Same thing here:

λ -calculus	FOL
$FV(x) := \{x\}$	$= FV(x)$
$FV(c) := \emptyset$	$= FV(c)$
$FV(MN) :=$	

Bound, Free, Binding Occurrences

Recall the notions of **bound**, **free**, and **binding** occurrences of variables in a term. Same thing here:

λ -calculus	FOL
$FV(x) := \{x\}$	$= FV(x)$
$FV(c) := \emptyset$	$= FV(c)$
$FV(MN) := FV(M) \cup FV(N)$	$= FV(M \wedge N)$
$FV(\lambda x. M) :=$	

Bound, Free, Binding Occurrences

Recall the notions of **bound**, **free**, and **binding** occurrences of variables in a term. Same thing here:

λ -calculus	FOL
$FV(x) := \{x\}$	$= FV(x)$
$FV(c) := \emptyset$	$= FV(c)$
$FV(MN) := FV(M) \cup FV(N)$	$= FV(M \wedge N)$
$FV(\lambda x. M) := FV(M) \setminus \{x\}$	$= FV(\forall x. M)$

Example: $FV(xy(\lambda yz. xyz)) = \{x, y\}$

Definition of Substitution

$M[x \leftarrow N]$ means substitute N for x in M

1. $x[x \leftarrow N] =$

2. $a[x \leftarrow N] =$

3. $(PQ)[x \leftarrow N] =$

4. $(\lambda x. P)[x \leftarrow N] =$

5. $(\lambda y. P)[x \leftarrow N] =$

6. $(\lambda y. P)[x \leftarrow N] =$

Definition of Substitution

$M[x \leftarrow N]$ means substitute N for x in M

1. $x[x \leftarrow N] = N$
2. $a[x \leftarrow N] = a$ if a is a constant or variable other than x
3. $(PQ)[x \leftarrow N] = (P[x \leftarrow N]Q[x \leftarrow N])$
4. $(\lambda x. P)[x \leftarrow N] = \lambda x. P$
5. $(\lambda y. P)[x \leftarrow N] = \lambda y. P[x \leftarrow N]$ if $y \neq x$ and $y \notin FV(N)$
6. $(\lambda y. P)[x \leftarrow N] = \lambda z. P[y \leftarrow z][x \leftarrow N]$ if $y \neq x$ and $y \in FV(N)$ where z is a variable such that $z \notin FV(NP)$

Definition of Substitution

$M[x \leftarrow N]$ means substitute N for x in M

1. $x[x \leftarrow N] = N$
2. $a[x \leftarrow N] = a$ if a is a constant or variable other than x
3. $(PQ)[x \leftarrow N] = (P[x \leftarrow N]Q[x \leftarrow N])$
4. $(\lambda x. P)[x \leftarrow N] = \lambda x. P$
5. $(\lambda y. P)[x \leftarrow N] = \lambda y. P[x \leftarrow N]$ if $y \neq x$ and $y \notin FV(N)$
6. $(\lambda y. P)[x \leftarrow N] = \lambda z. P[y \leftarrow z][x \leftarrow N]$ if $y \neq x$ and $y \in FV(N)$ where z is a variable such that $z \notin FV(NP)$

Cases similar to those for quantifiers: λ binding is 'generic'.

Substitution: Example

$$(x(\lambda x. xy))[x \leftarrow \lambda z. z]$$

Substitution: Example

$$\begin{aligned} (x(\lambda x. xy))[x \leftarrow \lambda z. z] &\stackrel{3}{=} x[x \leftarrow \lambda z. z](\lambda x. xy)[x \leftarrow \lambda z. z] \\ &\stackrel{1,4}{=} (\lambda z. z)\lambda x. xy \end{aligned}$$

Substitution: Example

$$\begin{aligned}(x(\lambda x. xy))[x \leftarrow \lambda z. z] &\stackrel{3}{=} x[x \leftarrow \lambda z. z](\lambda x. xy)[x \leftarrow \lambda z. z] \\ &\stackrel{1,4}{=} (\lambda z. z)\lambda x. xy\end{aligned}$$

$$(\lambda x. xy)[y \leftarrow x]$$

Substitution: Example

$$\begin{aligned}
 (x(\lambda x. xy))[x \leftarrow \lambda z. z] &\stackrel{3}{=} x[x \leftarrow \lambda z. z](\lambda x. xy)[x \leftarrow \lambda z. z] \\
 &\stackrel{1,4}{=} (\lambda z. z)\lambda x. xy
 \end{aligned}$$

$$\begin{aligned}
 (\lambda x. xy)[y \leftarrow x] &\stackrel{6}{=} \lambda z. ((xy)[x \leftarrow z][y \leftarrow x]) \\
 &\stackrel{3,1,2}{=} \lambda z. (zy[y \leftarrow x]) \\
 &\stackrel{3,2,1}{=} \lambda z. zx
 \end{aligned}$$

In the last example, clause 6 avoids capture, i.e., $\lambda x. xx$.

Reduction: Intuition

Reduction is the notion of “computing”, or “evaluation”, in the λ -calculus.

$$f\ x = x + 5 \rightsquigarrow f = \lambda x. x + 5$$

$$f\ 3 = 3 + 5 \rightsquigarrow (\lambda x. x + 5)(3) \rightarrow_{\beta} (x + 5)[x \leftarrow 3] = 3 + 5$$

β -reduction replaces a parameter by an argument.

This should propagate into contexts, e.g.

$$\lambda x. (\underline{(\lambda x. x + 5)(3)}) \rightarrow_{\beta} \lambda x. (3 + 5).$$

Reduction: Definition

- β -reduction: $(\lambda x.M)N \rightarrow_{\beta} M[x \leftarrow N]$
- Rules for contraction (of redices) in contexts:

$$\frac{M \rightarrow_{\beta} M'}{NM \rightarrow_{\beta} NM'}$$

$$\frac{M \rightarrow_{\beta} M'}{MN \rightarrow_{\beta} M'N}$$

$$\frac{M \rightarrow_{\beta} M'}{\lambda z.M \rightarrow_{\beta} \lambda z.M'}$$

- Reduction is reflexive-transitive closure

$$\frac{M \rightarrow_{\beta} N}{M \rightarrow_{\beta}^* N}$$

$$\frac{}{M \rightarrow_{\beta}^* M}$$

$$\frac{M \rightarrow_{\beta}^* N \quad N \rightarrow_{\beta}^* P}{M \rightarrow_{\beta}^* P}$$

- A term without redices is in β -normal form.

Reduction: Examples

$$\underline{(\lambda x. \lambda y. g \ x \ y) a \ b} \rightarrow_{\beta}$$

Reduction: Examples

$$\underline{(\lambda x. \lambda y. g x y) a b} \rightarrow_{\beta} \underline{(\lambda y. (g a y)) b} \rightarrow_{\beta}$$

Reduction: Examples

$$\underline{(\lambda x. \lambda y. g x y) a b} \rightarrow_{\beta} \underline{(\lambda y. (g a y)) b} \rightarrow_{\beta} g a b$$

$$\text{So } (\lambda x. \lambda y. g x y) a b \rightarrow_{\beta}^* g a b$$

Shows Currying

Reduction: Examples

$$\underline{(\lambda x. \lambda y. g x y) a b} \rightarrow_{\beta} \underline{(\lambda y. (g a y)) b} \rightarrow_{\beta} g a b$$

$$\text{So } (\lambda x. \lambda y. g x y) a b \rightarrow_{\beta}^* g a b$$

Shows **Currying**

$$\underline{(\lambda x. x x)(\lambda x. x x)} \rightarrow_{\beta}$$

Reduction: Examples

$$\underline{(\lambda x. \lambda y. g x y) a b} \rightarrow_{\beta} \underline{(\lambda y. (g a y)) b} \rightarrow_{\beta} g a b$$

$$\text{So } (\lambda x. \lambda y. g x y) a b \rightarrow_{\beta}^* g a b$$

Shows **Currying**

$$\underline{(\lambda x. xx)(\lambda x. xx)} \rightarrow_{\beta} \underline{(\lambda x. xx)(\lambda x. xx)} \rightarrow_{\beta} \dots$$

Shows **divergence**

Reduction: Examples

$$\underline{(\lambda x. \lambda y. g x y) a b} \rightarrow_{\beta} \underline{(\lambda y. (g a y)) b} \rightarrow_{\beta} g a b$$

$$\text{So } (\lambda x. \lambda y. g x y) a b \rightarrow_{\beta}^* g a b$$

Shows **Currying**

$$\underline{(\lambda x. xx)(\lambda x. xx)} \rightarrow_{\beta} \underline{(\lambda x. xx)(\lambda x. xx)} \rightarrow_{\beta} \dots$$

Shows **divergence**

$$\text{But } \underline{(\lambda xy. y)((\lambda x. xx)(\lambda x. xx))} \rightarrow_{\beta} \lambda y. y$$

Conversion

- β -conversion: “symmetric closure” of β -reduction

$$\frac{M \rightarrow_{\beta}^* N}{M =_{\beta} N} \qquad \frac{M =_{\beta} N}{N =_{\beta} M}$$

Conversion

- β -conversion: “symmetric closure” of β -reduction

$$\frac{M \rightarrow_{\beta}^* N}{M =_{\beta} N} \qquad \frac{M =_{\beta} N}{N =_{\beta} M}$$

- α -conversion: bound variable renaming (usually implicitly)

$$\lambda x.M =_{\alpha} \lambda z.M[x \leftarrow z] \quad \text{where } z \notin FV(M)$$

Conversion

- β -conversion: “symmetric closure” of β -reduction

$$\frac{M \rightarrow_{\beta}^* N}{M =_{\beta} N} \qquad \frac{M =_{\beta} N}{N =_{\beta} M}$$

- α -conversion: bound variable renaming (usually implicitly)

$$\lambda x.M =_{\alpha} \lambda z.M[x \leftarrow z] \quad \text{where } z \notin FV(M)$$

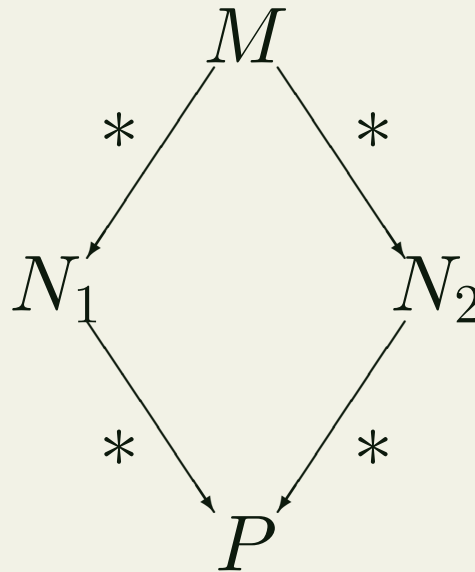
- η -conversion: for normal-form analysis

$$M =_{\eta} \lambda x.(Mx) \quad \text{if } x \notin FV(M)$$

λ -Calculus Meta-Properties

Confluence (equivalently, **Church-Rosser**): reduction is order-independent.

For all M, N_1, N_2 , if $M \rightarrow_{\beta}^* N_1$ and $M \rightarrow_{\beta}^* N_2$, then exists a P where $N_1 \rightarrow_{\beta}^* P$ and $N_2 \rightarrow_{\beta}^* P$.



Uniqueness of Normal Forms

Corollary of the Church-Rosser property:

If $M \rightarrow_{\beta}^* N_1$ and $M \rightarrow_{\beta}^* N_2$ where N_1 and N_2 in normal form, then

Uniqueness of Normal Forms

Corollary of the Church-Rosser property:

If $M \rightarrow_{\beta}^* N_1$ and $M \rightarrow_{\beta}^* N_2$ where N_1 and N_2 in normal form, then $N_1 =_{\alpha} N_2$.

Uniqueness of Normal Forms

Corollary of the Church-Rosser property:

If $M \rightarrow_{\beta}^* N_1$ and $M \rightarrow_{\beta}^* N_2$ where N_1 and N_2 in normal form, then $N_1 =_{\alpha} N_2$.

Example:

$$(\lambda xy. y)(\underline{(\lambda x. xx)a}) \rightarrow_{\beta} \underline{(\lambda xy. y)(aa)} \rightarrow_{\beta} \lambda y. y$$

$$\underline{(\lambda xy. y)(\underline{(\lambda x. xx)a})} \rightarrow_{\beta} \lambda y. y$$

Turing Completeness

The λ -calculus can represent all computable functions.

Simple Type Theory λ^{\rightarrow}

Motivation: Suppose you have constants 1, 2 with usual meaning. Is it sensible to write 1 2 (1 applied to 2)?

Simple Type Theory λ^{\rightarrow}

Motivation: Suppose you have constants 1, 2 with usual meaning. Is it sensible to write 1 2 (1 applied to 2)?

λ^{\rightarrow} (simply typed λ -calculus, simple type theory) restricts syntax to “meaningful expressions”.

In untyped λ -calculus, we have syntactic objects called terms.

Simple Type Theory λ^{\rightarrow}

Motivation: Suppose you have constants 1, 2 with usual meaning. Is it sensible to write 1 2 (1 applied to 2)?

λ^{\rightarrow} (**simply typed λ -calculus**, **simple type theory**) restricts syntax to “meaningful expressions”.

In untyped λ -calculus, we have **syntactic objects** called **terms**.

We now introduce syntactic objects called **types**.

Simple Type Theory λ^{\rightarrow}

Motivation: Suppose you have constants 1, 2 with usual meaning. Is it sensible to write 1 2 (1 applied to 2)?

λ^{\rightarrow} (simply typed λ -calculus, simple type theory) restricts syntax to “meaningful expressions”.

In untyped λ -calculus, we have syntactic objects called terms.

We now introduce syntactic objects called types.

We will say “a term has a type” or “a term is of a type”.

Two Syntaxes

- Syntax for **types** (\mathcal{B} a set of **base types**, $T \in \mathcal{B}$)

$$\tau ::= T \mid \tau \rightarrow \tau$$

Two Syntaxes

- Syntax for **types** (\mathcal{B} a set of **base types**, $T \in \mathcal{B}$)

$$\tau ::= T \mid \tau \rightarrow \tau$$

Examples: \mathbb{N} , $\mathbb{N} \rightarrow \mathbb{N}$, $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$, $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$

Two Syntaxes

- Syntax for **types** (\mathcal{B} a set of **base types**, $T \in \mathcal{B}$)

$$\tau ::= T \mid \tau \rightarrow \tau$$

Examples: \mathbb{N} , $\mathbb{N} \rightarrow \mathbb{N}$, $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$, $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$

- Syntax for (**raw**) **terms**: λ -calculus augmented with types

$$e ::= x \mid c \mid (ee) \mid (\lambda x^{\tau}. e)$$

$$(x \in \text{Var}, c \in \text{Const})$$

Signatures and Contexts

Generally (in various logic-related formalisms) a **signature** defines the “fixed” symbols of a language, and a **context** defines the “variable” symbols of a language.

Signatures and Contexts

Generally (in various logic-related formalisms) a **signature** defines the “fixed” symbols of a language, and a **context** defines the “variable” symbols of a language. In λ^{\rightarrow} ,

- a **signature** Σ is a sequence ($c \in \text{Const}$)

$$\Sigma ::= \langle \rangle \mid \Sigma, c : \tau$$

- a **context** Γ is a sequence ($x \in \text{Var}$)

$$\Gamma ::= \langle \rangle \mid \Gamma, x : \tau$$

What's the difference to signatures you have seen so far?

Type Assignment Calculus

We now define **type judgements**: “a term **has** a type” or “a term **is of** a type”. Generally this depends on a signature Σ and a context Γ . For example

$$\Gamma \vdash_{\Sigma} c \ x : \sigma$$

where $\Sigma = c : \tau \rightarrow \sigma$ and $\Gamma = x : \tau$.

Type Assignment Calculus

We now define **type judgements**: “a term **has** a type” or “a term **is of** a type”. Generally this depends on a signature Σ and a context Γ . For example

$$\Gamma \vdash_{\Sigma} c \ x : \sigma$$

where $\Sigma = c : \tau \rightarrow \sigma$ and $\Gamma = x : \tau$.

We usually leave Σ **implicit** and write \vdash instead of \vdash_{Σ} .

If Γ is empty it is omitted.

Type Assignment Calculus: Rules

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau} \text{assum}$$

$$\Gamma, x : \tau, \Delta \vdash x : \tau \quad \text{hyp}$$

$$\frac{\Gamma \vdash e : \sigma \rightarrow \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash ee' : \tau} \text{app}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x^\sigma. e : \sigma \rightarrow \tau} \text{abs}$$

Type Assignment Calculus: Rules

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau} \text{assum}$$

$$\Gamma, x : \tau, \Delta \vdash x : \tau \quad \text{hyp}$$

$$\frac{\Gamma \vdash e : \sigma \rightarrow \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash ee' : \tau} \text{app}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x^\sigma. e : \sigma \rightarrow \tau} \text{abs}$$

Note analogy to minimal logic over \rightarrow .

β -reduction defined as before, has subject reduction property and is strongly normalizing.

Example 1

$$\vdash \lambda x^{\sigma}. \lambda y^{\tau}. x :$$

Example 1

$$\vdash \lambda x^{\sigma}. \lambda y^{\tau}. x : \sigma \rightarrow (\tau \rightarrow \sigma)$$

Example 1

$$\frac{}{\vdash \lambda x^{\sigma}. \lambda y^{\tau}. x : \sigma \rightarrow (\tau \rightarrow \sigma)} \text{abs}$$

Example 1

$$\frac{x : \sigma \vdash \lambda y^{\tau}. x : \tau \rightarrow \sigma}{\vdash \lambda x^{\sigma}. \lambda y^{\tau}. x : \sigma \rightarrow (\tau \rightarrow \sigma)} \text{abs}$$

Example 1

$$\frac{\frac{}{x : \sigma \vdash \lambda y^{\tau}. x : \tau \rightarrow \sigma} \text{abs}}{\vdash \lambda x^{\sigma}. \lambda y^{\tau}. x : \sigma \rightarrow (\tau \rightarrow \sigma)} \text{abs}$$

Example 1

$$\frac{\frac{x : \sigma, y : \tau \vdash x : \sigma}{x : \sigma \vdash \lambda y^{\tau}. x : \tau \rightarrow \sigma} \text{abs}}{\vdash \lambda x^{\sigma}. \lambda y^{\tau}. x : \sigma \rightarrow (\tau \rightarrow \sigma)} \text{abs}$$

Example 1

$$\frac{\frac{\frac{}{x : \sigma, y : \tau \vdash x : \sigma} \text{hyp}}{x : \sigma \vdash \lambda y^{\tau}. x : \tau \rightarrow \sigma} \text{abs}}{\vdash \lambda x^{\sigma}. \lambda y^{\tau}. x : \sigma \rightarrow (\tau \rightarrow \sigma)} \text{abs}$$

Example 1

$$\frac{\frac{\frac{}{x : \sigma, y : \tau \vdash x : \sigma} \text{hyp}}{x : \sigma \vdash \lambda y^{\tau}. x : \tau \rightarrow \sigma} \text{abs}}{\vdash \lambda x^{\sigma}. \lambda y^{\tau}. x : \sigma \rightarrow (\tau \rightarrow \sigma)} \text{abs}$$

Note the use of **schematic types**!

Example 1

$$\frac{\frac{x : \sigma, y : \tau \vdash x : \sigma}{x : \sigma \vdash \lambda y^{\tau}. x : \tau \rightarrow \sigma} \text{abs}}{\vdash \lambda x^{\sigma}. \lambda y^{\tau}. x : \sigma \rightarrow (\tau \rightarrow \sigma)} \text{abs}$$

Note the use of **schematic types**!

Also note that applications of ***hyp*** are usually not explicitly marked in proof.

Example 2

$$\Gamma = f : \sigma \rightarrow \sigma \rightarrow \tau, x : \sigma$$

$$\vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}. \lambda x^{\sigma}. f \ x \ x :$$

Example 2

$$\Gamma = f : \sigma \rightarrow \sigma \rightarrow \tau, x : \sigma$$

$$\vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}. \lambda x^{\sigma}. f x x : (\sigma \rightarrow \sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau$$

Example 2

$$\Gamma = f : \sigma \rightarrow \sigma \rightarrow \tau, x : \sigma$$

$$\frac{}{\vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}. \lambda x^{\sigma}. f x x : (\sigma \rightarrow \sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau} \text{abs}$$

Example 2

$$\Gamma = f : \sigma \rightarrow \sigma \rightarrow \tau, x : \sigma$$

$$\frac{f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^{\sigma}. f x x : \sigma \rightarrow \tau}{\vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}. \lambda x^{\sigma}. f x x : (\sigma \rightarrow \sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau} \text{abs}$$

Example 2

$$\Gamma = f : \sigma \rightarrow \sigma \rightarrow \tau, x : \sigma$$

$$\frac{\frac{}{f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^{\sigma}. f x x : \sigma \rightarrow \tau} \text{abs}}{\vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}. \lambda x^{\sigma}. f x x : (\sigma \rightarrow \sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau} \text{abs}$$

Example 2

$$\Gamma = f : \sigma \rightarrow \sigma \rightarrow \tau, x : \sigma$$

$$\frac{\Gamma \vdash f x x : \tau}{f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^\sigma. f x x : \sigma \rightarrow \tau} \text{abs}$$

$$\frac{}{\vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}. \lambda x^\sigma. f x x : (\sigma \rightarrow \sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau} \text{abs}$$

Example 2

$$\Gamma = f : \sigma \rightarrow \sigma \rightarrow \tau, x : \sigma$$

$$\begin{array}{c}
 \hline
 \Gamma \vdash f \ x \ x : \tau \quad \text{app} \\
 \hline
 f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^\sigma. f \ x \ x : \sigma \rightarrow \tau \quad \text{abs} \\
 \hline
 \vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}. \lambda x^\sigma. f \ x \ x : (\sigma \rightarrow \sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau \quad \text{abs}
 \end{array}$$

Example 2

$$\Gamma = f : \sigma \rightarrow \sigma \rightarrow \tau, x : \sigma$$

$$\begin{array}{c}
 \frac{\Gamma \vdash f x : \sigma \rightarrow \tau \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash f x x : \tau} \textit{app} \\
 \frac{\Gamma \vdash f x x : \tau}{f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^{\sigma}. f x x : \sigma \rightarrow \tau} \textit{abs} \\
 \frac{f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^{\sigma}. f x x : \sigma \rightarrow \tau}{\vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}. \lambda x^{\sigma}. f x x : (\sigma \rightarrow \sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau} \textit{abs}
 \end{array}$$

Example 2

$$\Gamma = f : \sigma \rightarrow \sigma \rightarrow \tau, x : \sigma$$

$$\begin{array}{c}
 \frac{\Gamma \vdash f x : \sigma \rightarrow \tau}{\Gamma \vdash f x x : \tau} \text{app} \quad \Gamma \vdash x : \sigma \\
 \frac{\Gamma \vdash f x x : \tau}{f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^{\sigma}. f x x : \sigma \rightarrow \tau} \text{abs} \\
 \frac{f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^{\sigma}. f x x : \sigma \rightarrow \tau}{\vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}. \lambda x^{\sigma}. f x x : (\sigma \rightarrow \sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau} \text{abs}
 \end{array}$$

Example 2

$$\Gamma = f : \sigma \rightarrow \sigma \rightarrow \tau, x : \sigma$$

$$\begin{array}{c}
 \Gamma \vdash f : \sigma \rightarrow \sigma \rightarrow \tau \quad \Gamma \vdash x : \sigma \\
 \hline
 \Gamma \vdash f x : \sigma \rightarrow \tau \quad \Gamma \vdash x : \sigma \\
 \hline
 \Gamma \vdash f x x : \tau \\
 \hline
 f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^\sigma. f x x : \sigma \rightarrow \tau \\
 \hline
 \vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}. \lambda x^\sigma. f x x : (\sigma \rightarrow \sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau
 \end{array}
 \begin{array}{l}
 \\
 \text{app} \\
 \\
 \text{app} \\
 \\
 \text{abs} \\
 \text{abs}
 \end{array}$$

Example 3

$$\Sigma = f : \sigma \rightarrow \sigma \rightarrow \tau$$

$$\Gamma = x : \sigma$$

$$\Gamma \vdash f \ x \ x : \tau$$

Example 3

$$\Sigma = f : \sigma \rightarrow \sigma \rightarrow \tau$$

$$\Gamma = x : \sigma$$

$$\frac{\frac{f : \sigma \rightarrow \sigma \rightarrow \tau \in \Sigma}{\Gamma \vdash f : \sigma \rightarrow \sigma \rightarrow \tau} \text{assum} \quad \Gamma \vdash x : \sigma}{\Gamma \vdash f x : \sigma \rightarrow \tau} \text{app} \quad \frac{\Gamma \vdash f x : \sigma \rightarrow \tau \quad \Gamma \vdash x : \sigma}{\Gamma \vdash f x x : \tau} \text{app}$$

Note that this time, f is a constant.

Example 3

$$\Sigma = f : \sigma \rightarrow \sigma \rightarrow \tau$$

$$\Gamma = x : \sigma$$

$$\frac{\Gamma \vdash f : \sigma \rightarrow \sigma \rightarrow \tau \quad \Gamma \vdash x : \sigma}{\Gamma \vdash f x : \sigma \rightarrow \tau} \text{app} \quad \frac{\Gamma \vdash f x : \sigma \rightarrow \tau \quad \Gamma \vdash x : \sigma}{\Gamma \vdash f x x : \tau} \text{app}$$

Note that this time, f is a constant.

We will often suppress applications of *assum*.

An Extension of the Typed λ -Calculus

We will now look at the typed λ -calculus **extended** by **polymorphism** and **type classes**.

As we will see **later**, this is the universal representation for object logics in Isabelle.

Polymorphism: Intuition

In functional programming, the function *append* for concatenating two lists works the same way on integer lists and on character lists: *append* is **polymorphic**.

Type language must be generalized to include **type variables** (denoted by $\alpha, \beta \dots$) and **type constructors**.

Example: *append* has type $\alpha \text{ list} \rightarrow \alpha \text{ list} \rightarrow \alpha \text{ list}$, and by type instantiation, it can also have type, say, $\text{int list} \rightarrow \text{int list} \rightarrow \text{int list}$.

Polymorphism: Two Syntaxes

- Syntax for **polymorphic types** (\mathcal{B} a set of **type constructors** including \rightarrow), $T \in \mathcal{B}$, α is a **type variable**)

$$\tau ::= \alpha \mid (\tau, \dots, \tau) T$$

Polymorphism: Two Syntaxes

- Syntax for **polymorphic types** (\mathcal{B} a set of **type constructors** including \rightarrow), $T \in \mathcal{B}$, α is a **type variable**)

$$\tau ::= \alpha \mid (\tau, \dots, \tau) T$$

Examples: \mathbb{N} , $\mathbb{N} \rightarrow \mathbb{N}$, $\alpha \text{ list}$, $\mathbb{N} \text{ list}$, $(\mathbb{N}, \text{bool}) \text{ pair}$.

Polymorphism: Two Syntaxes

- Syntax for **polymorphic types** (\mathcal{B} a set of **type constructors** including \rightarrow), $T \in \mathcal{B}$, α is a **type variable**)

$$\tau ::= \alpha \mid (\tau, \dots, \tau) T$$

Examples: \mathbb{N} , $\mathbb{N} \rightarrow \mathbb{N}$, $\alpha \text{ list}$, $\mathbb{N} \text{ list}$, $(\mathbb{N}, \text{bool}) \text{ pair}$.

- Syntax for (**raw**) **terms** as before:

$$e ::= x \mid c \mid (ee) \mid (\lambda x^\tau. e)$$

$$(x \in \text{Var}, c \in \text{Const})$$

Polymorphic Type Assignment Calculus

Type substitutions (denoted Θ) defined in analogy to substitutions in FOL. Apart from application of Θ in rule *assum*, type assignment is as for λ^{\rightarrow} :

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau \Theta} \text{assum}^*$$

$$\Gamma, x : \tau, \Delta \vdash x : \tau \quad \text{hyp}$$

$$\frac{\Gamma \vdash e : \sigma \rightarrow \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash ee' : \tau} \text{app}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x^{\sigma}. e : \sigma \rightarrow \tau} \text{abs}$$

*: Θ is any type substitution.

Type Classes: Intuition

Type classes are a way of “making ad-hoc polymorphism less ad-hoc” [HHPW96, WB89].

Type classes are used to group together types with certain **properties**, in particular, types for which certain **symbols** are defined.

We only **sketch** the formalisation here, and refer to [HHPW96, Nip93, NP93] for details.

Type Classes in Isabelle

- **Syntactic classes** (similarly as in Haskell): E.g., declare that there exists a class *ord* which is a **subclass** of class *term*, and that for any $\tau :: \text{ord}$, the constant \leq is defined and has type $\tau \rightarrow \tau \rightarrow \text{bool}$. Isabelle has **syntax for this**.
- **Axiomatic classes**: Declare (axiomatize) that certain theorems should hold for a $\tau :: \kappa$ where κ is a type class. E.g., axiomatize that \leq is reflexive by an (Isabelle) theorem " $x \leq x$ ". Isabelle has **syntax for this**.

To use a class, we can declare **members** of it, e.g., \mathbb{N} is a member of *ord*.

Syntax: Classes, Types, and Terms

Based on

- a set of **type classes**, say $\mathcal{K} = \{ord, order, lattice, \dots\}$,
 - a set of **type constructors**, say
 $\mathcal{B} = \{bool, _ \rightarrow _, ind, _ list, _ set \dots\}$,
 - a set of **constants** $Const$ and a set of **variables** Var ,
- we define

- **Polymorphic types**: $\tau ::= \alpha \mid \alpha :: \kappa \mid (\tau, \dots, \tau) T$
- **Raw terms (as before)**: $e ::= x \mid c \mid (ee) \mid (\lambda x^\tau. e)$
 $(\alpha \text{ is type variable, } T \in \mathcal{B}, \kappa \in \mathcal{K}, x \in Var, c \in Const)$

Type Assignment Calculus with Type Classes

Assume some syntax for declaring $\tau :: \kappa$ and $\kappa \prec \kappa'$. In addition introduce the rule

$$\frac{\tau :: \kappa \quad \kappa \prec \kappa'}{\tau :: \kappa'} \textit{subclass}$$

Type assignment rules as **before**, but type substitution Θ in

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau\Theta} \textit{assum}$$

must respect **class constraints**: for each $\alpha :: \kappa$ occurring in τ where $\alpha\Theta = \sigma$, **judgement** $\sigma :: \kappa$ must hold.

Example

Suppose that by virtue of declarations, we have $\mathbb{N} :: \text{order}$, $\text{order} \prec \text{ord}$, and $\leq : \alpha :: \text{ord} \rightarrow \alpha \rightarrow \text{bool} \in \Sigma$. Derive

$$\frac{\mathbb{N} :: \text{order} \quad \text{order} \prec \text{ord}}{\mathbb{N} :: \text{ord}} \text{subclass}$$

and then $(\Theta = [\alpha \leftarrow \mathbb{N}])$

$$\frac{(\leq : (\alpha :: \text{ord}) \rightarrow \alpha \rightarrow \text{bool}) \in \Sigma}{\vdash \leq : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{bool}} \text{assum}$$

which respects the class constraint since the judgement $\mathbb{N} :: \text{ord}$ was derived above.

Type Assignment and $\alpha\beta\eta$ -Conversion

Type construction:

- Type construction is decidable.

Type Assignment and $\alpha\beta\eta$ -Conversion

Type construction:

- Type construction is decidable.
- There is a practically useful implementation for type-construction (Hindley-Milner Algorithm \mathcal{W} [Mil78, NN99]).

Term congruence ($e =_{\alpha\beta\eta} e'?$) is decidable.

Higher-Order Unification

The λ -calculus is “the” metalogic. Hence we now (sometimes) call its variables “metavariables” for emphasis and we precede them with ?. E.g. they can stand for object-level formulae. Will be more precise later.

Higher-Order Unification

The λ -calculus is “the” metalogic. Hence we now (sometimes) call its variables “metavariables” for emphasis and we precede them with ?. E.g. they can stand for object-level formulae. Will be more precise later.

Two issues concerning metavariables are:

- suitable renamings of metavariables;
- unification before rule application.

What Is Higher-Order Unification?

Unification of terms e, e' : find **substitution** θ for metavariables such that $e\theta =_{\alpha\beta\eta} e'\theta$.

Examples:

$$?X + ?Y =_{\alpha\beta\eta} x + x$$

$$?P(x) =_{\alpha\beta\eta} x + x$$

$$f(?Xx) =_{\alpha\beta\eta} ?Y x$$

What Is Higher-Order Unification?

Unification of terms e, e' : find **substitution** θ for metavariables such that $e\theta =_{\alpha\beta\eta} e'\theta$.

Examples:

$$?X + ?Y =_{\alpha\beta\eta} x + x$$

$$\textcolor{red}{?P}(x) =_{\alpha\beta\eta} x + x$$

$$f(?Xx) =_{\alpha\beta\eta} ?Y x$$

Why **higher-order**? Metavariables may be instantiated to **functions**, e.g. $[?P \leftarrow \lambda y.y + y]$.

Higher-Order Unification: Facts

- Unification **modulo** $\alpha\beta$ (HO-unification) is semi-decidable (in Isabelle: incomplete).
- Unification **modulo** $\alpha\beta\eta$ is undecidable (in Isabelle: incomplete).

Higher-Order Unification: Facts

- Unification **modulo** $\alpha\beta$ (HO-unification) is semi-decidable (in Isabelle: incomplete).
- Unification **modulo** $\alpha\beta\eta$ is undecidable (in Isabelle: incomplete).
- HO-unification is well-behaved for most practical cases.
- Important fragments (like **HO-patterns**) are decidable.
- HO-unification has possibly infinitely many solutions.

Summary on λ -Calculus

λ -calculus is a formalism for writing functions.

β -reduction is the notion of “computing” in λ -calculus.

λ -calculus is Turing-complete.

λ^{\rightarrow} restricts syntax to “meaningful” λ -terms.

Extension of typed λ -calculus used to represent syntax of object logics. λ -terms stand for object terms/formulae, possibly containing “distinguished occurrences” of (object) variables. This will be explained thoroughly next lecture.

HO-unification important in constructing proofs. ►

More Detailed Explanations

$$3 + 5 = 8?$$

As you might guess, the formalism of the λ -calculus is not directly related to usual arithmetic and so it is not built into this formalism that $3 + 5$ should evaluate to 8. However, it may be a reasonable choice, depending on the context, to extend the λ -calculus in this way, but this is not our concern at the moment.

Var and Const

Similarly as for first-order logic, a language of the untyped λ -calculus is characterized by giving a set of variables and a set of constants.

One can think of *Const* as a signature.

Note that *Const* could be empty.

Note also that the word **constant** has a different meaning in the λ -calculus from that of first-order logic. In both formalisms, constants are just symbols.

In first-order logic, a constant is a special case of a function **symbol**, namely a function symbol of arity 0.

In the λ -calculus, one does not speak of function **symbols**. In the untyped λ -calculus, **any** λ -term (including a constant) can be **applied** to another term, and so any λ -term can be called a “unary function”. A constant being applied to a term is something which would contradict

the intuition about constants in first-order logic. So for the λ -calculus, think of constant as opposed to a variable, an application, or an abstraction.

How do We Call those Terms?

A λ -term can either be

- a variable (case x), or
- a constant (case c), or
- an application of a λ -term to another λ -term (case (ee)), or
- an abstraction over a variable x (case $(\lambda x. e)$).

Backus-Naur Form

A notation like

$$e ::= x \mid c \mid (ee) \mid (\lambda x. e)$$

$$\tau ::= T \mid \tau \rightarrow \tau$$

$$e ::= x \mid c \mid (ee) \mid (\lambda x^\tau. e)$$

$$P ::= x \mid \neg P \mid P \wedge P \mid P \rightarrow P \dots$$

for specifying syntax is called **Backus-Naur form** (BNF) for expressing grammars. For example, the first BNF-clause reads: a λ -term can be a variable, or a constant, or a λ -term applied to a λ -term, or a **λ -abstraction**, which is a λ -term of the form $\lambda x. e$, where e is a λ -term. The BNF is a very common formalism for specifying syntax, e.g., of

programming languages. See [here](#) or [here](#).

$(\lambda\text{-})$ Terms

So just like **first-order logic**, the λ -calculus has a syntactic category called **terms**. But the word “term” has a different meaning for the λ -calculus than for first-order logic, and so one can say λ -term for emphasis.

Note that at this stage, we have no syntactic category called “formula” for the λ -calculus.

λ -Calculus: Notational Conventions

We write $\lambda x_1 x_2 \dots x_n . e$ instead of $\lambda x_1 . (\lambda x_2 . (\dots e) \dots)$.

$e_1 e_2 \dots e_n$ is equivalent to $(\dots (e_1 e_2) \dots e_n) \dots$, not $(e_1 (e_2 \dots e_n) \dots)$.

Note that this is in contrast to the [associativity of logical operators](#).

There are some good reasons for these conventions.

Infix Notation

Strictly speaking, $\lambda x. x + 5$ does not adhere to the definition of syntax of λ -terms, at least if we parse it in the usual way: $+$ is an infix constant applied to arguments x and 5 .

If we parse $x + 5$ as $((x+)5)$, i.e., x applied to (the constant) $+$, and the resulting term applied to (the constant) 5 , then $\lambda x. x + 5$ would indeed adhere to the definition of syntax of λ -terms, but of course, this is pathological and not intended here.

It is convenient to allow for extensions of the syntax of λ -terms, allowing for:

- application to several arguments rather than just one;
- **infix notation**.

Such an extension is inessential for the expressive power of the

λ -calculus. Instead of having a binary infix constant $+$ and writing $\lambda x. x + 5$, we could have a constant *plus* according to the original syntax and write $\lambda x. ((plus\ x)\ 5)$ (i.e., write $+$ in a **Curry**ed way).

Reduction

Reduction is the notion of “computing”, or “evaluation”, in the λ -calculus.

Notations for Substitutions

Here we use the notation $e[x \leftarrow t]$ for the term obtained from e by replacing x with t . There is also the notation $e[t/x]$, and confusingly, also $e[x/t]$. We will attempt to be consistent within this course, but be aware that you may find such different notations in the literature.

λ Binding Is ‘Generic’

Recall the [definition](#) of substitution for first-order logic.

We observe that [binding](#) and [substitution](#) are some very general concepts. So far, we have seen four binding operators: \exists , \forall and λ , and [set comprehensions](#). The λ operator is the most generic of those operators, in that it does not have a fixed meaning hard-wired into it in the way that the quantifiers do. In fact, it is possible to have it as the only operator on the level of the metalogic. We will see this [later](#).

Avoiding Capture

If it wasn't for clause 6, i.e., if we applied clause 5 ignoring the requirement on freeness, then $(\lambda x. xy)[y \leftarrow x]$ would be $\lambda x. xx$.

Parameters and Arguments

In the λ -term $(\lambda x.M)N$, we say that N is an argument (and the function $\lambda x.M$ is applied to this argument), and every occurrence of x in M is a parameter (we say this because x is bound by the λ).

This terminology may be familiar to you if you have experience in functional programming, but actually, it is also used in the context of function and procedure declarations in imperative programming.

Propagation into Contexts

In

$$\lambda x. (\underline{(\lambda x. x + 5)(3)}),$$

the underlined part is a subterm occurring in a context. β -reduction should be applicable to this subterm.

Like a Proof System

As you see, β -reduction is defined using rules (two of them being **axioms**, the rest **proper rules**) in the same way that we have defined **proof systems for logic** before. Note that we wrote the first **axiom** defining β -reduction without a horizontal bar.

Redex

In a λ -term, a subterm of the form $(\lambda x. M)N$ is called a **redex** (plural **redices**). It is a subterm to which β -reduction can be applied.

Currying

You may be familiar with functions taking several arguments, or equivalently, a tuple of arguments, rather than just one argument.

In the λ -calculus, but also in functional programming, it is common not to have tuples and instead use a technique called **Currying** (**Schönfinkeln** in German). So instead of writing $g(a, b)$, we write $g a b$, which is read as follows: g is a function which takes an argument a and returns a function which then takes an argument b .

Recall that application **associates to the left**, so $g a b$ is read $(g a) b$.

Currying will become even clearer once we introduce the **typed λ -calculus**.

Divergence

We say that a β -reduction sequence **diverges** if it is infinite.

Note that for $(\lambda xy. y)((\lambda x. xx)(\lambda x. xx))$, there is a finite β -reduction sequence

$$(\lambda xy. y)((\lambda x. xx)(\lambda x. xx)) \rightarrow_{\beta} \lambda y. y$$

but there is also a diverging sequence

$$(\lambda xy. y)((\lambda x. xx)(\lambda x. xx)) \rightarrow_{\beta} (\lambda xy. y)((\lambda x. xx)(\lambda x. xx)) \rightarrow_{\beta} \dots$$

α -Conversion

α -conversion is usually applied implicitly, i.e., without making it an explicit step. So for example, one would simply write:

$$\lambda z. z =_{\beta} \lambda x. x$$

η -Conversion

η -conversion is defined as

$$M =_{\eta} \lambda x. (Mx) \quad \text{if } x \notin FV(M)$$

It is needed for reasoning about normal forms.

$$g\ x =_{\eta} \lambda y. g\ x\ y \quad \text{reflects} \quad g\ x\ b =_{\beta} (\lambda y. g\ x\ y)b$$

More specifically: if we did not have the η -conversion rule, then $g\ x$ and $\lambda y. g\ x\ y$ would not be “equivalent” up to conversion. But that seems unreasonable, because they behave the same way when applied to b . Applied to b , both terms can be converted to $g\ x\ b$. This is why it is reasonable to introduce a rule such that $g\ x$ and $\lambda y. g\ x\ y$ are “equivalent” up to conversion.

Confluence and Church-Rosser

A reduction \rightarrow is called **confluent** if

for all M, N_1, N_2 , if $M \rightarrow^* N_1$ and $M \rightarrow^* N_2$, then there exists a P where $N_1 \rightarrow^* P$ and $N_2 \rightarrow^* P$.

A reduction is called **Church-Rosser** if

for all N_1, N_2 , if $N_1 \xleftrightarrow{*} N_2$, then there exists a P where $N_1 \rightarrow^* P$ and $N_2 \rightarrow^* P$.

Here, $\leftarrow := (\rightarrow)^{-1}$ is the inverse of \rightarrow , and $\leftrightarrow := \leftarrow \cup \rightarrow$ is the **symmetric closure** of \rightarrow , and $\xleftrightarrow{*} := (\leftrightarrow)^*$ is the **reflexive transitive symmetric closure** of \rightarrow .

So for example, if we have

$$M_1 \rightarrow M_2 \rightarrow M_3 \rightarrow M_4 \leftarrow M_5 \leftarrow M_6 \rightarrow M_7 \leftarrow M_8 \leftarrow M_9$$

then we would write $M_1 \xrightarrow{*} M_9$.

Confluence is equivalent to the Church-Rosser property [BN98, page 10].

λ -Calculus Metaproperties

By metaproperties, we mean properties about reduction and conversion sequences in general.

Turing Completeness

The untyped λ -calculus is Turing complete. This is usually shown not by mimicking a Turing machine in the λ -calculus, but rather by exploiting the fact that the Turing computable functions are the same class as the μ -recursive functions. In a lecture on theory of computation, you have probably learned that the μ -recursive functions are obtained from the primitive recursive functions by so-called **unbounded minimalization**, while the primitive recursive functions are built from the 0-place zero function, projection functions and the successor function using composition and primitive recursion [LP81].

The proof that the untyped λ -calculus can compute all μ -recursive functions is thus based on showing that each of the mentioned ingredients can be encoded in the untyped λ -calculus. While we are not going to study this, one crucial point is that it should be possible to

encode the natural numbers and the arithmetic operations in the untyped λ -calculus.

Term Language

We also say that we have defined a **term language**. A particular language is given by a signature, although for the untyped λ -calculus this is simply the set of constants *Const*.

Type Language

We can say that we define a **type language**, i.e., a language consisting of types. A particular type language is characterized by a giving a set of base types \mathcal{B} . One might also call \mathcal{B} a **type signature**.

A typical example of a set of base types would be $\{\mathbb{N}, \text{bool}\}$, where \mathbb{N} represents the natural numbers and *bool* the Boolean values \perp and \top . All that matters is that \mathcal{B} is some fixed set “defined by the user”.

Types: Intuition

The type $\mathbb{N} \rightarrow \mathbb{N}$ is the type of a function that takes a natural number and returns a natural number.

The type $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$ is the type of a function that takes a function, which takes a natural number and returns a natural number, and returns a natural number.

Types Are Right-Associative

To save parentheses, we use the following convention: types associate to the right, so $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ stands for $\mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$.

Recall that application **associates to the left**. This may seem confusing at first, but actually, it turns out that the two conventions concerning associativity fit together very neatly.

Raw Terms

In the context of **typed** versions of the λ -calculus, **raw** terms are terms built ignoring any **typing conditions**. So raw terms are simply terms as defined for the **untyped λ -calculus**, possibly augmented with **type superscripts**.

Augmenting with Types

So far, this is just syntax!

The notation $(\lambda x^\tau. e)$ simply specifies that **binding** occurrences of variables in simple type theory are tagged with a superscript, where the use of the letter τ makes it clear (in this particular context) that the superscript must be some **type**, defined by the grammar we just gave.

Var **and** *Const*

Var and *Const* are the sets of variables and constants, respectively, as for the **untyped λ -calculus**.

Sequences

A sequence is a collection of objects which differs from **sets** in that a sequence contains the objects in a certain **order**, and there can be **multiple** occurrences of an object.

We write a sequence containing the objects o_1, \dots, o_n as $\langle o_1, \dots, o_n \rangle$, or sometimes simply o_1, \dots, o_n .

If Ω is the sequence o_1, \dots, o_n , then we write Ω, o for the sequence $\langle o_1, \dots, o_n, o \rangle$ and o, Ω for the sequence $\langle o, o_1, \dots, o_n \rangle$.

An empty sequence is denoted by $\langle \rangle$.

Type Binding

We call an expression of the form $x : \tau$ or $c : \tau$ a **type binding**.

The use of the letter τ makes it clear (in this particular context) that the superscript must be some **type**, defined by the grammar we just gave.

Signatures in Various Formalisms

For **propositional logic**, we did not use the notion of signature, although we mentioned that strictly speaking, there is not just **the** language of propositional logic, but rather **a** language of propositional logic which depends on the choice of the **variables**.

In **first-order logic**, a signature was a pair $(\mathcal{F}, \mathcal{P})$ defining the function and predicate symbols, although strictly speaking, the signature should also specify the arities of the symbols in some way. Recall that we did not bother to fix a precise technical way of specifying those arities. We were content with saying that they are specified in “some unambiguous way”.

In **sorted logic**, the signature must also specify the sorts of all symbols. But we did not study sorted logic in any detail.

In the untyped λ -calculus, the signature is simply the **set of constants**.

Summarizing, we have not been very precise about the notion of a

signature so far.

For λ^{\rightarrow} , the rules for “legal” terms become more tricky, and it is important to be formal about signatures.

In λ^{\rightarrow} , a signature associates a **type** with each constant symbol by writing $c : \tau$.

Usually, we will assume that $Const$ is clear from the context, and that Σ contains an expression of the form $c : \tau$ for each $c \in Const$, and in fact, that Σ is clear from the context as well. Since Σ contains an expression of the form $c : \tau$ for each $c \in Const$, it is redundant to give $Const$ explicitly. It is sufficient to give Σ .

Type Judgement

The expression

$$\Gamma \vdash_{\Sigma} c\ x : \sigma$$

is called a **type judgement**. It says that given the signature $\Sigma = x : \tau$ and the context $\Gamma = c : \tau \rightarrow \sigma$, the term

$c\ x$ **has type** σ or

$c\ x$ **is of type** σ or

$c\ x$ **is assigned type** σ .

Recall that you have seen **other judgements** before.

\in for Sequences?

Recall that Σ is a **sequence**. By abuse of notation, we sometimes identify this sequence with a set and allow ourselves to write $c : \tau \in \Sigma$.

We may also write $\Sigma \subseteq \Sigma'$ meaning that $c : \tau \in \Sigma$ implies $c : \tau \in \Sigma'$.

System of Rules

Type assignment is defined as a system of rules for deriving **type judgements**, in the same way that we have defined **derivability judgements** for **logics**, and **β -reduction** for the untyped λ -calculus.

Minimal Logic over \rightarrow

Recall the **sequent rules** of the \rightarrow / \wedge fragment of propositional logic. Consider now only the \rightarrow fragment. We call this fragment **minimal logic over \rightarrow** .

If you take the rule

$$\Gamma, x : \tau, \Delta \vdash x : \tau \quad \textit{hyp}$$

of λ^{\rightarrow} and throw away the terms (so you keep only the types), you obtain essentially the rule for assumptions

$$\Gamma \vdash A \quad (\text{where } A \in \Gamma)$$

of propositional logic.

Likewise, if you do the same with the rule

$$\frac{\Gamma \vdash e : \sigma \rightarrow \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash ee' : \tau} \textit{app}$$

of $\lambda \rightarrow$, you obtain essentially the rule

$$\frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \rightarrow\text{-}E$$

of propositional logic.

Finally, if you do the same with the rule

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x^\sigma. e : \sigma \rightarrow \tau} \textit{abs}$$

of λ^{\rightarrow} , you obtain essentially the rule

$$\frac{A, \Gamma \vdash B}{\Gamma \vdash A \rightarrow B} \rightarrow\text{-I}$$

of propositional logic.

Note that in this setting, there is no analogous propositional logic rule for

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau} \text{assum}$$

So for the moment, we can observe a close analogy between λ^{\rightarrow} , for Σ being empty, and the \rightarrow fragment of propositional logic, which is also called **minimal logic over \rightarrow** .

Such an analogy between a **type theory** (of which λ^{\rightarrow} is an example) and a logic is referred to in the literature as **Curry-Howard isomorphism**. One

also speaks of **propositions as types** [GLT89]. The isomorphism is so fundamental that it is common to characterize type theories by the logic they represent, so for example, one might say:

λ^{\rightarrow} is the type theory of minimal logic over \rightarrow .

Subject Reduction

Subject reduction is the following property: **reduction** does not change the type of a term, so if $\vdash_{\Sigma} M : \tau$ and $M \rightarrow_{\beta} N$, then $\vdash_{\Sigma} N : \tau$.

(Strongly) Normalizing β -Reduction

The simply-typed λ -calculus, unlike the **untyped λ -calculus**, is **normalizing**, that is to say, every term has a normal form. Even more, it is **strongly** normalizing, that is, this normal form is reached regardless of the reduction order.

An Alternative for *hyp*

One could also formulate *hyp* as follows:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{hyp}$$

That would be in close analogy to LF, a system not treated here.

Schematic Types

In this example, you may regard σ and τ as base types (this would require that $\sigma, \tau \in \mathcal{B}$), but in fact, it is more natural to regard them as **metavariables** standing for arbitrary types. Whatever types you substitute for σ and τ , you obtain a derivation of a type judgement.

This is in analogy to **schematic derivations in a logic**.

Note also that Σ is irrelevant for the example and hence arbitrary.

Constants vs. Variables

In Example 3, we have $f : \sigma \rightarrow \sigma \rightarrow \tau \in \Sigma$, and so f is a **constant**.

In Example 2, we have $f : \sigma \rightarrow \sigma \rightarrow \tau \in \Gamma$, and so f is a **variable**.

Looking at the different derivations of the type judgement $\Gamma \vdash f x x : \tau$ in Examples 2 and 3, you may find that they are very similar, and you may wonder: What is the point? Why do we distinguish between constants and variables?

In fact, one could simulate constants by variables. When setting up a type theory or programming language, there are choices to be made about whether there should be a distinction between variables and constants, and what it should look like. There is a famous epigram by Alan Perlis:

One man's constant is another man's variable.

For our purposes, it is much clearer conceptually to make the distinction. For example, if we want to introduce the natural numbers in our λ^{\rightarrow} language, then it is intuitive that there should be constants $1, 2, \dots$ denoting the numbers. If $1, 2, \dots$ were variables, then we could write strange expressions like $\lambda 2^{\mathbb{N} \rightarrow \mathbb{N}}. y$, so we could use 2 as a variable of type $\mathbb{N} \rightarrow \mathbb{N}$.

(Parametric) Polymorphism

In functional programming, you will come across functions that operate uniformly on many different types. For example, a function *append* for concatenating two lists works the same way on integer lists and on character lists. Such functions are called **polymorphic**.

More precisely, this kind of polymorphism, where a function does exactly the same thing regardless of the type instance, is called **parametric polymorphism**, as opposed to **ad-hoc polymorphism**.

In a type system with polymorphism, the notion of **base type** (which is just a **type constant**, i.e., one symbol) is generalized to a **type constructor** with an arity ≥ 0 . A type constructor of arity n applied to n types is then a **type**. For example, there might be a type constructor *list* of arity 1, and *int* of arity 0. Then, *int list* is a type.

Note that application of a type constructor to a type is written in **postfix**

notation, unlike any notation for function application [we have seen](#).

However, other conventions exist, [even within Isabelle](#).

See [[Pau96](#), [Tho95b](#), [Tho99](#)] for details on the polymorphic type systems of functional programming languages.

Ad-hoc Polymorphism

Ad-hoc polymorphism, also called **overloading**, refers to functions that do different (although usually similar) things on different types. For example, a function \leq may be defined as $'a' \leq 'b' \dots$ on characters and $1 \leq 2 \dots$ on integers. In this case, the symbol \leq must be declared and defined separately for each type.

This is in contrast to **parametric polymorphism**, but also somewhat different from **type classes**.

Type classes are a way of “making ad-hoc polymorphism less ad-hoc” [HHPW96, WB89].

Type Classes

Type classes are a way of “making ad-hoc polymorphism less ad-hoc” [HHPW96, WB89].

Type classes are used to group together types with certain **properties**, in particular, types for which certain **symbols** are defined.

For example, for some types, a symbol \leq (which is a **binary infix predicate**) may exist and for some not, and we could have a type class *ord* containing all types for which it exists.

Suppose you want to sort a list of elements (smaller elements should come before bigger elements). This is only defined for elements of a type for which the symbol \leq exists.

Note that while a symbol such as \leq may have a similar meaning for different types (for example, integers and reals), one cannot say that it means **exactly the same thing** regardless of the type of the argument to

which it is applied. In fact, \leq has to be defined separately for each type in *ord*.

This is in contrast to **parametric polymorphism**, but also somewhat different from **ad-hoc polymorphism**: The types of the symbols must not be declared separately. E.g., one has to declare only once that \leq is of type $(a :: \textit{ord}, \alpha)$.

Polymorphic Type Language

As before, we define a **type language**, i.e., a language consisting of types, and a particular type language is characterized by a giving a certain set of symbols \mathcal{B} . But unlike before, \mathcal{B} is now a set of **type constructors**. Each type constructor has an arity associated with it just like a **function in first-order logic**. The intention is that a type constructor may be **applied** to types.

Following the conventions of ML [Pau96], we write types in **postfix notation**, something we have not seen before. I.e., the type constructor comes **after** the arguments it is applied to.

It makes perfect sense to view the function construction arrow \rightarrow as **type constructor**, however written infix rather than postfix.

So the \mathcal{B} is some fixed set “defined by the user”, but it should definitely always include \rightarrow .

Type Substitutions

A **type substitution** replaces a type variable by a type, just like in **first-order logic**, a substitution replaces a variable by a term.

Syntactic Classes

A syntactic class is a class of types for which certain **symbols** are declared to exist. Isabelle has a syntax for such declarations. E.g., the declaration

```
sort ord < term
const <= : [ $\alpha :: \text{ord}$ ,  $\alpha$ ] => bool
```

may form part of an Isabelle theory file. It declares a type class *ord* which is a **subclass** (that's what the $<$ means; in mathematical notation it will be written \prec) of a class *term*, meaning that any type in *ord* is also in *term*. We will write the “**class judgement**” $\text{ord} \prec \text{term}$. The class *term* must be defined elsewhere.

The second line declares a symbol $<=$. Such a declaration is preceded by the keyword `const`. The notation $\alpha :: \text{ord}$ stands for a type variable **constrained** to be in class *ord*. So $<=$ is declared to be of type

$[\alpha :: ord, \alpha] \Rightarrow bool$, meaning that it takes two arguments of a type in the class *ord* and returns a term of type *bool*. The symbol $\Rightarrow (= >)$ is the **function type arrow** in Isabelle. Note that the second occurrence of α is written without $:: ord$. This is because it is enough to state the class constraint once.

Note also that $[\alpha :: ord, \alpha] = > bool$ is in fact just another way of writing $\alpha :: ord = > \alpha = > bool$, similarly as for **goals**.

Haskell [HHPW96] has type classes but ML [Pau96] hasn't.

Axiomatic Classes

In addition to declaring the **syntax** of a type class, one can axiomatize the semantics of the symbols. Again, Isabelle has a syntax for such declarations. E.g., the declaration

```
axclass order < ord
  order_refl: ''x <= x ''
  order_trans: ''[| x <= y; y <= z |] ==> x <= z''
  ...
```

may form part of an Isabelle theory file. It declares an **axiomatic** type class *order* which is a **subclass** of *ord* defined **above**.

The next two lines are the **axioms**. Here, *order_refl* and *order_trans* are the names of the axioms. Recall that \implies is the implication symbol in Isabelle (that is to say, the metalevel implication).

Whenever an Isabelle theory **declares** that a type is a member of such a class, it must **prove** those axioms.

The rationale of having axiomatic classes is that it allows for proofs that hold in different but similar mathematical structures to be done only once. So for example, all theorems that hold for dense orders can be proven for **all** dense orders with one single proof.

Members of a Type Class

One also speaks of a type being an **instance** of a type class, but this is slightly confusing, since we also say that a **type** can be an instance of another **type**, e.g., $\mathbb{N} \rightarrow \mathbb{N}$ is an instance of α , since $\alpha[\alpha \leftarrow (\mathbb{N} \rightarrow \mathbb{N})] = \mathbb{N} \rightarrow \mathbb{N}$. So it is better to speak of a member of a type class.

Isabelle provides a syntax for declaring that a type is a member of a type class, e.g.

```
instance nat :: ord
```

declares that type `nat` is a member of class `ord`.

If the class κ is a syntactic class, such a declaration **must come** with a **definition** of the **symbols** that are declared to exist for κ .

In addition, if κ is an axiomatic class, such a declaration **must come** with a **proof** of the **axioms**.

If a type τ is (by declaration) a member of class κ , we write the “**class judgement**” $\tau :: \kappa$.

Renaming

Whenever a rule is applied, the metavariables occurring in it must be renamed to **fresh** variables to ensure that no metavariable in the rule has been used in the proof before.

The notion fresh is often casually used in logic, and it means: this variable has never been used before. To be more precise, one should say: never been used before in the relevant context.

Unification

The mechanism to instantiate metavariables as needed is called **(higher-order) unification**. Unification is the process of finding a **substitution** that makes two terms equal.

We will later see more formally **what it is** and also **where it is used**.

Type Class Syntax

The set \mathcal{K} we gave is **incomplete** and just **exemplary**.

So the set of type classes involved in an Isabelle theory is a finite set of names (written lower-case), typically including *ord*, *order*, and *lattice*.

We have seen some Isabelle syntax for **declaring** the type classes **previously**.

In grammars and elsewhere, κ is the letter we use for “type class”.

Type Constructor Syntax

As before, the set \mathcal{B} we gave is **incomplete** (there are "...") and just **exemplary**. We might call \mathcal{B} a **type signature**.

Note also that an $_$ is used to denote the **arity of a type constructor**.

- $_ \textit{list}$ means that *list* is unary type constructor;
- $_ \rightarrow _$ means that \rightarrow is a binary infix type constructor.

The notation using $_$ is slightly abusive since the $_$ is not actually part of the type constructor. $_ \textit{list}$ is not a type constructor; *list* is a type constructor.

So the set of type constructors involved in an Isabelle theory is a finite set of names (written lower-case) with each having an arity associated, typically including *bool*, \rightarrow , and *list*. Note however that *bool* is fundamental (since object level predicates are modeled functions taking

terms to a Boolean), and so is \rightarrow , the constructor of the function space between two types.

In grammars and elsewhere, T is the letter we use for “type constructor”.

\rightarrow as Type Constructor

In λ^{\rightarrow} , types were built from base types using a “special symbol” \rightarrow .

When we generalize λ^{\rightarrow} to a λ -calculus with polymorphism, this “special symbol” becomes a **type constructor**. However, the **syntax** is still special, and it is interpreted in a particular way.

Polymorphic Types Syntax

$$\tau ::= \alpha \mid \alpha :: \kappa \mid (\tau, \dots, \tau) T \quad (\alpha \text{ is type variable})$$

is a grammar defining what polymorphic types are (syntactically). **As before**, τ is the non-terminal we use for (now: polymorphic) types.

This grammar is not exemplary but generic, and it deserves a closer look.

A type variable is a variable that stands for a **type**, as opposed to a **term**.

We have not given a grammar for type variables, but assume that there is a countable set of type variables disjoint from the set of term variables. We use α as the non-terminal for a type variable (abusing notation, we often also use α to denote an actual type variable).

First, note that a type variable may be followed by a **class constraint** $:: \kappa$ (**recall** that κ is the non-terminal for type classes). However, a type variable is not necessarily followed by such a constraint, for example if

the type variable already occurs elsewhere and is constrained in that place. We have already seen this.

Moreover, a polymorphic type is obtained by preceding a type constructor with a tuple of types. The arity of the tuple must be equal to the declared arity of the type constructor.

It is not shown here that for some special type constructors, such as \rightarrow , the argument may also be written infix.

Type Construction

Type construction is the problem of given a Σ , Γ and e , finding a t such that $\Sigma, \Gamma \vdash e : \tau$.

Sometimes one also considers the problem where Γ is unknown and must also be constructed.

Term Congruence

$\alpha\beta\eta$ -conversion is defined as for λ^{\rightarrow} . Given two (extended) λ -terms e and e' , it is decidable whether $e =_{\alpha\beta\eta} e'$.

Solutions for Unification Problems

A solution for $?X + ?Y =_{\alpha\beta\eta} x + x$ is $[?X \leftarrow x, ?Y \leftarrow x]$.

A solution for $?P(x) =_{\alpha\beta\eta} x + x$ is $[?P \leftarrow (\lambda y.y + y)]$.

A solution for $f(?Xx) =_{\alpha\beta\eta} ?Y x$ is $[?X \leftarrow (\lambda z.z), ?Y \leftarrow (\lambda z.f z)]$.

Unification Modulo

Unification of terms e, e' modulo $\alpha\beta$ means finding a substitution θ for metavariables such that $\theta(e) =_{\alpha\beta} \theta(e')$.

Likewise, unification of terms e, e' modulo $\alpha\beta\gamma$ means finding a substitution σ for metavariables such that $\sigma(e) =_{\alpha\beta\gamma} \sigma(e')$.

Encoding Syntax

Metatheory: Motivation

Previously, we have seen the (polymorphically) typed λ -calculus (with type classes).

Now, we will see how the typed λ -calculus can be used as a metalanguage for representing the syntax of an object logic, e.g. first-order logic.

Idea: An object-level proposition is a meta-level term.

Metalogic type o for propositions.

The terms of type o encode object level propositions:

$\phi \in Prop$ iff $\ulcorner \phi \urcorner : o$.

Later: How do we represent the proofs/provability?

Why Have a Metalogic?

Why should we have a **meta-** or **framework** logic rather than implementing provers for each object logic individually?

- + Implement 'core' only once
- + Shared support for automation
- + Conceptual framework for exploring what a logic is

But

- + / – Metalayer between user and logic
- Makes assumptions about structure of logic

λ^{\rightarrow} : Review

- Syntax for types (\mathcal{B} a set of base types, $T \in \mathcal{B}$)

$$\tau ::= T \mid \tau \rightarrow \tau$$

Examples: \mathbb{N} , $\mathbb{N} \rightarrow \mathbb{N}$, $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$, $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$

λ^{\rightarrow} : Review

- Syntax for types (\mathcal{B} a set of **base types**, $T \in \mathcal{B}$)

$$\tau ::= T \mid \tau \rightarrow \tau$$

Examples: \mathbb{N} , $\mathbb{N} \rightarrow \mathbb{N}$, $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$, $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$

- Syntax for terms: **λ -calculus** augmented **with types**

$$e ::= x \mid c \mid (ee) \mid (\lambda x^{\tau}. e)$$

$$(x \in \text{Var}, c \in \text{Const})$$

Type Assignment

- Signature $\Sigma ::= \langle \rangle \mid \Sigma, c : \tau$.
- Context $\Gamma ::= \langle \rangle \mid \Gamma, x : \tau$.
- Type assignment rules

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau} \text{assum}$$

$$\Gamma, x : \tau, \Delta \vdash x : \tau \quad \text{hyp}$$

$$\frac{\Gamma \vdash e : \sigma \rightarrow \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash ee' : \tau} \text{app}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x^\sigma. e : \sigma \rightarrow \tau} \text{abs}$$

Representing Syntax of Propositional Logic

Let the following fragment of propositional logic, which we call *Prop*, be our object logic:

$$P ::= x \mid \neg P \mid P \wedge P \mid P \rightarrow P$$

Representing Syntax of Propositional Logic

Let the following fragment of propositional logic, which we call *Prop*, be our object logic:

$$P ::= x \mid \neg P \mid P \wedge P \mid P \rightarrow P$$

Let λ^{\rightarrow} be our metalogic. Declare

- $\mathcal{B} = \{o\}$.
- Signature types constants:

$$\Sigma = \langle not : \quad , and : \quad , imp : \quad \rangle$$

Representing Syntax of Propositional Logic

Let the following fragment of propositional logic, which we call *Prop*, be our object logic:

$$P ::= x \mid \neg P \mid P \wedge P \mid P \rightarrow P$$

Let λ^{\rightarrow} be our metalogic. Declare

- $\mathcal{B} = \{o\}$.
- Signature types constants:

$$\Sigma = \langle not : o \rightarrow o, and : o \rightarrow o \rightarrow o, imp : o \rightarrow o \rightarrow o \rangle$$

- Context types variables.

Digression: Programming Languages

λ^{\rightarrow} is the theory underlying typed functional programming. Our declaration of \mathcal{B} and Σ on the previous slide corresponds to the declaration of an **algebraic datatype** in a functional programming language [Pau96]:

```
datatype Prop =
  VarInject of Variable | not of Prop
  | and of Prop * Prop | imp of Prop * Prop
```

Example of First-Order Syntax

$$a : o \vdash \text{imp} (\text{not } a) a : o$$

Example of First-Order Syntax

$$a : o \vdash \text{imp} (\text{not } a) a : o$$

$$\begin{array}{c}
 \begin{array}{c}
 a : o \vdash \text{imp} : o \rightarrow o \rightarrow o \\
 \hline
 a : o \vdash \text{imp} (\text{not } a) : o \rightarrow o
 \end{array}
 \quad
 \begin{array}{c}
 \begin{array}{c}
 a : o \vdash \text{not} : o \rightarrow o \quad a : o \vdash a : o \\
 \hline
 a : o \vdash \text{not } a : o
 \end{array} \\
 \hline
 a : o \vdash \text{imp} (\text{not } a) a : o
 \end{array}
 \quad
 a : o \vdash a : o
 \end{array}$$

Applications of *hyp* and *assum* suppressed. Otherwise always rule *app*.

Non-example of First-Order Syntax

$a : o \vdash \text{not } (\text{imp } a) \ a : o$

Non-example of First-Order Syntax

$$a : o \vdash \text{not } (\text{imp } a) \ a : o$$

$$\frac{a : o \vdash \text{not} : o \rightarrow o \quad \frac{a : o \vdash \text{imp} : o \rightarrow o \rightarrow o \quad a : o \vdash a : o}{a : o \vdash \text{imp } a : o \rightarrow o}}{a : o \vdash \text{imp } a : o \rightarrow o} \text{ ???}$$

Non-example of First-Order Syntax

$$a : o \vdash \text{not } (\text{imp } a) \ a : o$$

$$\frac{a : o \vdash \text{not} : o \rightarrow o \quad \frac{a : o \vdash \text{imp} : o \rightarrow o \rightarrow o \quad a : o \vdash a : o}{a : o \vdash \text{imp } a : o \rightarrow o}}{a : o \vdash \text{imp } a : o \rightarrow o} \text{ ???}$$

No proof possible! (Requires analysis of **normal forms**.)

Bijection between $Prop$ and o

We desire **bijection** $\ulcorner \cdot \urcorner : Prop \rightarrow o$ that is

- **adequate**: each proposition in $Prop$ can be represented by a λ^{\rightarrow} term of type o :

$$P \in Prop \text{ then } \Gamma \vdash \ulcorner P \urcorner : o$$

Bijection between $Prop$ and o

We desire **bijection** $\ulcorner \cdot \urcorner : Prop \rightarrow o$ that is

- **adequate**: each proposition in $Prop$ can be represented by a λ^{\rightarrow} term of type o :

$$P \in Prop \text{ then } \Gamma \vdash \ulcorner P \urcorner : o$$

- **faithful**: each λ^{\rightarrow} term of type o represents a proposition in $Prop$:

$$\Gamma \vdash t : o \text{ then } \ulcorner t \urcorner^{-1} \in Prop$$

Adequacy of Bijection

Example: $(\neg a) \rightarrow b \in Prop$ therefore $imp (not\ a)\ b : o$

Adequacy of Bijection

Example: $(\neg a) \rightarrow b \in Prop$ therefore *imp* (*not* *a*) *b* : *o*

Formalize mapping $\lceil \cdot \rceil$:

$$\lceil x \rceil = x \quad \text{for } x \text{ a variable}$$

$$\lceil \neg P \rceil = \text{not } \lceil P \rceil$$

$$\lceil P \wedge Q \rceil = \text{and } \lceil P \rceil \lceil Q \rceil$$

$$\lceil P \rightarrow Q \rceil = \text{imp } \lceil P \rceil \lceil Q \rceil$$

Adequacy of Bijection

Example: $(\neg a) \rightarrow b \in Prop$ therefore $imp (not\ a)\ b : o$

Formalize mapping $\ulcorner \cdot \urcorner$:

$$\ulcorner x \urcorner = x \quad \text{for } x \text{ a variable}$$

$$\ulcorner \neg P \urcorner = not\ \ulcorner P \urcorner$$

$$\ulcorner P \wedge Q \urcorner = and\ \ulcorner P \urcorner\ \ulcorner Q \urcorner$$

$$\ulcorner P \rightarrow Q \urcorner = imp\ \ulcorner P \urcorner\ \ulcorner Q \urcorner$$

Formal statement accounts for variables:

If $P \in Prop$, and if for each propositional variable x in P , we have $x : o \in \Gamma$, then $\Gamma \vdash \ulcorner P \urcorner : o$.

Adequacy of Bijection

Example: $(\neg a) \rightarrow b \in Prop$ therefore $imp (not\ a)\ b : o$

Formalize mapping $\ulcorner \cdot \urcorner$:

$$\ulcorner x \urcorner = x \quad \text{for } x \text{ a variable}$$

$$\ulcorner \neg P \urcorner = not\ \ulcorner P \urcorner$$

$$\ulcorner P \wedge Q \urcorner = and\ \ulcorner P \urcorner\ \ulcorner Q \urcorner$$

$$\ulcorner P \rightarrow Q \urcorner = imp\ \ulcorner P \urcorner\ \ulcorner Q \urcorner$$

Formal statement accounts for variables:

If $P \in Prop$, and if for each propositional variable x in P , we have $x : o \in \Gamma$, then $\Gamma \vdash \ulcorner P \urcorner : o$. Proof by **induction**.

Faithfulness of Bijection

Define $\ulcorner \cdot \urcorner^{-1}$

$$\ulcorner x \urcorner^{-1} = x \quad \text{for } x \text{ a variable}$$

$$\ulcorner \text{not } P \urcorner^{-1} = \neg \ulcorner P \urcorner^{-1}$$

$$\ulcorner \text{and } P \text{ } Q \urcorner^{-1} = \ulcorner P \urcorner^{-1} \wedge \ulcorner Q \urcorner^{-1}$$

$$\ulcorner \text{imp } P \text{ } Q \urcorner^{-1} = \ulcorner P \urcorner^{-1} \rightarrow \ulcorner Q \urcorner^{-1}$$

Faithfulness of Bijection

Define $\ulcorner \cdot \urcorner^{-1}$

$$\ulcorner x \urcorner^{-1} = x \quad \text{for } x \text{ a variable}$$

$$\ulcorner \text{not } P \urcorner^{-1} = \neg \ulcorner P \urcorner^{-1}$$

$$\ulcorner \text{and } P \text{ } Q \urcorner^{-1} = \ulcorner P \urcorner^{-1} \wedge \ulcorner Q \urcorner^{-1}$$

$$\ulcorner \text{imp } P \text{ } Q \urcorner^{-1} = \ulcorner P \urcorner^{-1} \rightarrow \ulcorner Q \urcorner^{-1}$$

For **bijection**, should have $\ulcorner \ulcorner P \urcorner \urcorner^{-1} = P$ and $\ulcorner \ulcorner t \urcorner^{-1} \urcorner = t$.

Former is **trivial**, but what about latter?

$\lceil t \rceil^{-1}$ Is not Total

Example: For $t = \text{not } ((\lambda x^o. x) a)$, we have $a : o \vdash t : o$

$$\begin{array}{c}
 \dfrac{a : o, x : o \vdash x : o}{a : o \vdash \lambda x^o. x : o \rightarrow o} \text{abs} \quad a : o \vdash a : o \\
 \dfrac{a : o \vdash \text{not} : o \rightarrow o \quad \dfrac{a : o \vdash (\lambda x^o. x) a : o}{a : o \vdash \text{not } ((\lambda x^o. x) a) : o} \text{app}}{a : o \vdash \text{not } ((\lambda x^o. x) a) : o} \text{app}
 \end{array}$$

But $\lceil t \rceil^{-1}$ is undefined!

Normal Forms

If $t : o$, then $t =_{\beta\eta} t'$, for $t' : o$ a **canonical** ($\beta\eta$ -long) normal form, e.g.

$$\begin{aligned} \text{not } ((\lambda x^o. x) a) &=_{\beta\eta} \text{not } a \\ \text{not} &=_{\beta\eta} \lambda x^o. \text{not } x \\ \text{imp } (\text{not } ((\lambda x^o. x) a)) &=_{\beta\eta} \lambda x^o. \text{imp } (\text{not } a) x \end{aligned}$$

Bijection Theorem

The encoding $\ulcorner \cdot \urcorner$ is a bijection between propositional formulae **with variables in Γ** and canonical terms t' , where $\Gamma \vdash t' : o$.

Bijection Theorem

The encoding $\ulcorner \cdot \urcorner$ is a bijection between propositional formulae **with variables in Γ** and canonical terms t' , where $\Gamma \vdash t' : o$.

Proof: Based on **normalization**

$$\frac{\frac{x : \sigma \vdash e : \tau}{\vdash \lambda x^\sigma. e : \sigma \rightarrow \tau} \text{abs} \quad \vdash e' : \sigma}{\vdash (\lambda x^\sigma. e) e' : \tau} \text{app} \quad \Rightarrow \quad \vdash e[x \leftarrow e'] : \tau$$

Bijection Theorem

The encoding $\ulcorner \cdot \urcorner$ is a bijection between propositional formulae **with variables in Γ** and canonical terms t' , where $\Gamma \vdash t' : o$.

Proof: Based on **normalization**

$$\frac{\frac{x : \sigma \vdash e : \tau}{\vdash \lambda x^\sigma . e : \sigma \rightarrow \tau} \text{abs} \quad \vdash e' : \sigma}{\vdash (\lambda x^\sigma . e) e' : \tau} \text{app} \quad \Rightarrow \quad \vdash e[x \leftarrow e'] : \tau$$

Corollary: If $t : o$ then $t =_{\beta\eta} t'$ and $\ulcorner t' \urcorner^{-1} \in Prop$ for some canonical t' .

Representing Syntax of First-Order Logic

In *Prop*, we only have the syntactic category of formulae (propositions), represented in λ^{\rightarrow} by the type *o*.

Representing Syntax of First-Order Logic

In *Prop*, we only have the **syntactic category** of formulae (propositions), represented in λ^{\rightarrow} by the type *o*.

In **first-order** logic, we also have the **syntactic category** of terms. For representation in λ^{\rightarrow} , we now introduce type *i*, so $\mathcal{B} = \{i, o\}$.

Representing Syntax of First-Order Logic

In *Prop*, we only have the syntactic category of formulae (propositions), represented in λ^{\rightarrow} by the type o .

In first-order logic, we also have the syntactic category of terms. For representation in λ^{\rightarrow} , we now introduce type i , so $\mathcal{B} = \{i, o\}$.

Just like $\Gamma \vdash a : o$ means that a represents a proposition, $\Gamma \vdash t : i$ means that t represents a term.

Example: First-Order Arithmetic (FOA)

Following fragment of FOA is our **object level language**:

Terms $T ::= x \mid 0 \mid sT \mid T + T \mid T \times T$

Formulae $F ::= T = T \mid \neg F \mid F \wedge F \mid F \rightarrow F$

Example: First-Order Arithmetic (FOA)

Following fragment of FOA is our **object level language**:

Terms $T ::= x \mid 0 \mid sT \mid T + T \mid T \times T$

Formulae $F ::= T = T \mid \neg F \mid F \wedge F \mid F \rightarrow F$

In λ^{\rightarrow} (on metalevel), we define **signature** $\Sigma = \Sigma_T \cup \Sigma_F$:

$$\Sigma_T = \langle \text{zero} : i, \text{succ} : i \rightarrow i, \text{plus} : i \rightarrow i \rightarrow i, \\ \text{times} : i \rightarrow i \rightarrow i \rangle$$

$$\Sigma_F = \langle \text{eq} : i \rightarrow i \rightarrow o, \text{not} : o \rightarrow o, \text{and} : o \rightarrow o \rightarrow o, \dots \rangle$$

Example: $\ulcorner x + s0 \urcorner =$

Example: First-Order Arithmetic (FOA)

Following fragment of FOA is our **object level language**:

Terms $T ::= x \mid 0 \mid sT \mid T + T \mid T \times T$

Formulae $F ::= T = T \mid \neg F \mid F \wedge F \mid F \rightarrow F$

In λ^{\rightarrow} (on metalevel), we define **signature** $\Sigma = \Sigma_T \cup \Sigma_F$:

$$\Sigma_T = \langle \text{zero} : i, \text{succ} : i \rightarrow i, \text{plus} : i \rightarrow i \rightarrow i, \\ \text{times} : i \rightarrow i \rightarrow i \rangle$$

$$\Sigma_F = \langle \text{eq} : i \rightarrow i \rightarrow o, \text{not} : o \rightarrow o, \text{and} : o \rightarrow o \rightarrow o, \dots \rangle$$

Example: $\ulcorner x + s0 \urcorner = \text{plus } x (\text{succ zero})$.

Quantifiers in First-Order Syntax

Along the same lines, one might suggest

$$all : var \rightarrow o \rightarrow o, \quad \text{so } \ulcorner \forall x. P \urcorner = all\ x \ulcorner P \urcorner$$

But this approach has some problems:

Quantifiers in First-Order Syntax

Along the same lines, one might suggest

$$all : var \rightarrow o \rightarrow o, \quad \text{so } \ulcorner \forall x. P \urcorner = all\ x \ulcorner P \urcorner$$

But this approach has some problems:

- Variables are also **terms**, so “ $var \subseteq i$ ”? No subtyping!

Quantifiers in First-Order Syntax

Along the same lines, one might suggest

$$all : var \rightarrow o \rightarrow o, \quad \text{so } \ulcorner \forall x. P \urcorner = all\ x \ulcorner P \urcorner$$

But this approach has some problems:

- Variables are also **terms**, so “ $var \subseteq i$ ”? No subtyping!
- all is not a **binding operator** in λ^{\rightarrow} . E.g.,
 $(p(x) \wedge \forall x. q(x))[x \leftarrow a]$ cannot be modeled as
 $(and\ (p\ x)(all\ x\ (q\ x)))[x \leftarrow a]$.

Higher-Order Abstract Syntax (HOAS)

Example, full FOA: $F ::= \dots \forall x. A \mid \exists x. A$

$\Sigma = \Sigma_T \cup \Sigma_F \cup \Sigma_Q$:

$$\Sigma_Q = \langle all : (i \rightarrow o) \rightarrow o, exists : (i \rightarrow o) \rightarrow o \rangle$$

Higher-Order Abstract Syntax (HOAS)

Example, full **FOA**: $F ::= \dots \forall x. A \mid \exists x. A$

$\Sigma = \Sigma_T \cup \Sigma_F \cup \Sigma_Q$:

$$\Sigma_Q = \langle all : (i \rightarrow o) \rightarrow o, exists : (i \rightarrow o) \rightarrow o \rangle$$

Extend the definition of $\ulcorner \cdot \urcorner$:

$$\ulcorner \forall x. P \urcorner = all (\lambda x^i. \ulcorner P \urcorner)$$

$$\ulcorner \exists x. P \urcorner = exists (\lambda x^i. \ulcorner P \urcorner)$$

Faithfulness and adequacy as before.

Examples

$$\ulcorner \forall x. x = x \urcorner = \text{all}(\lambda x^i. \text{eq } x \ x)$$

$$\ulcorner \forall x. \exists y. \neg(x + x = y) \urcorner = \\ \text{all}(\lambda x^i. \text{exists}(\lambda y^i. \text{not}(\text{eq}(\text{plus } x \ x) \ y))))$$

Examples

$$\ulcorner \forall x. x = x \urcorner = all(\lambda x^i. eq\ x\ x)$$

$$\ulcorner \forall x. \exists y. \neg(x + x = y) \urcorner = \\ all(\lambda x^i. exists(\lambda y^i. not\ (eq\ (plus\ x\ x)\ y)))$$

Example derivation (all but one steps use rule *app*).

$$\frac{\frac{\frac{x : i \vdash eq : i \rightarrow i \rightarrow o \quad x : i \vdash x : i}{x : i \vdash eq\ x : i \rightarrow o} \quad x : i \vdash x : i}{x : i \vdash eq\ x\ x : o} \quad \text{abs}}{\vdash all : (i \rightarrow o) \rightarrow o} \quad \vdash \lambda x^i. eq\ x\ x : i \rightarrow o$$

$$\vdash all(\lambda x^i. eq\ x\ x) : o$$

Order

Order of a type: For type τ written $\tau_1 \rightarrow \dots \rightarrow \tau_n$, right associated, $\tau_n \in \mathcal{B}$:

- $Ord(\tau) = 0$ if $\tau \in \mathcal{B}$, i.e., if $n = 1$;
- $Ord(\tau) = 1 + \max(Ord(\tau_i))$,

Order

Order of a type: For type τ written $\tau_1 \rightarrow \dots \rightarrow \tau_n$, **right associated**, $\tau_n \in \mathcal{B}$:

- $Ord(\tau) = 0$ if $\tau \in \mathcal{B}$, i.e., if $n = 1$;
- $Ord(\tau) = 1 + \max(Ord(\tau_i))$,

Intuition: “**functions as arguments**”.

A type of order 1 is **first-order**, of order 2 **second-order** etc.

A type of order > 1 is called **higher order**.

Why “Higher Order”?

Constants representing **propositional operators** (logical symbols) or **non-logical symbols** are first-order (hence **first-order** syntax):

$$\textit{and} : o \rightarrow o \rightarrow o$$

Why “Higher Order”?

Constants representing **propositional operators** (logical symbols) or **non-logical symbols** are first-order (hence **first-order** syntax):

$$\textit{and} : o \rightarrow o \rightarrow o$$

Variable binding operators are higher-order (hence **higher-order** syntax):

$$\textit{all} : (i \rightarrow o) \rightarrow o$$

Exercise: Summation Operator

What is the order of the summation operator Σ ?

Exercise: Summation Operator

What is the order of the summation operator \sum ?

$$sum : i \rightarrow i \rightarrow (i \rightarrow i) \rightarrow i$$

$$\ulcorner \sum_{x=0}^n (x + 2) \urcorner =$$

Exercise: Summation Operator

What is the order of the summation operator \sum ?

$$sum : i \rightarrow i \rightarrow (i \rightarrow i) \rightarrow i$$

$$\ulcorner \sum_{x=0}^n (x + 2) \urcorner = sum\ zero\ n\ (\lambda x^i. plus\ x\ (succ\ succ\ zero))$$

So the order is 2.

Why “Abstract”?

HOAS looks quite different from the concrete object level syntax and hence “abstracts” from this object level syntax. More specifically, different object level **binding** operators are represented by a combination of a **constant** (*all*, *exists*) and the **generic** λ -operator.

Thanks to this technique, standard operations on syntax need no **special encoding**, but are supported implicitly by λ^{\rightarrow} . We will now see this.

Binding

Binding on the object level and metalevel coincide.

So in $\forall x. P$, all occurrences of x in P are bound, and likewise, in $all(\lambda x^i. \lceil P \rceil)$, all occurrences of x in $\lceil P \rceil$ are bound.

This provides support for substitution.

Substitution

Recall rules for \forall :

$$\frac{\forall x. P(x)}{P(t)} \forall\text{-}E$$

Substitution

Recall rules for \forall :

$$\frac{\forall x. P(x)}{P(t)} \forall\text{-}E \quad \rightsquigarrow \quad \frac{all\ P}{P(t)} \forall\text{-}E$$

Substitution

Recall rules for \forall :

$$\frac{\forall x. P(x)}{P(t)} \forall\text{-}E \quad \rightsquigarrow \quad \frac{all\ P}{P(t)} \forall\text{-}E$$

$$\frac{\forall x. x = x}{x = x[x \leftarrow 0]} \forall\text{-}E$$

Now apply substitution. . .

Substitution

Recall rules for \forall :

$$\frac{\forall x. P(x)}{P(t)} \forall\text{-}E \quad \rightsquigarrow \quad \frac{all\ P}{P(t)} \forall\text{-}E$$

$$\frac{\forall x. x = x}{0 = 0} \forall\text{-}E$$

Now apply substitution. . .

Substitution

Recall rules for \forall :

$$\frac{\forall x. P(x)}{P(t)} \forall\text{-}E \quad \rightsquigarrow \quad \frac{all\ P}{P(t)} \forall\text{-}E$$

$$\frac{\forall x. x = x}{0 = 0} \forall\text{-}E \quad \rightsquigarrow \quad \frac{all\ (\lambda x^i. eq\ x\ x)}{(\lambda x^i. eq\ x\ x)\ zero} \forall\text{-}E$$

Now apply substitution. . .

Now apply β -reduction. . .

Substitution

Recall rules for \forall :

$$\frac{\forall x. P(x)}{P(t)} \forall\text{-}E \quad \rightsquigarrow \quad \frac{all\ P}{P(t)} \forall\text{-}E$$

$$\frac{\forall x. x = x}{0 = 0} \forall\text{-}E \quad \rightsquigarrow \quad \frac{all\ (\lambda x^i. eq\ x\ x)}{eq\ zero\ zero} \forall\text{-}E$$

Now apply substitution. . .

Now apply β -reduction. . .

We now understand “marked positions in a formula”.

Equivalence under Bound Variable Renaming

On the object level, formulae are equivalent under renaming of bound variables:

$$(\forall x. P \leftrightarrow \forall y. P[x \leftarrow y])$$

Equivalence under Bound Variable Renaming

On the object level, formulae are equivalent under renaming of bound variables:

$$(\forall x. P \leftrightarrow \forall y. P[x \leftarrow y])$$

Likewise, on the metalevel, formulae obtained by bound variable renaming are α -equivalent:

$$all(\lambda x^i. P) =_{\alpha} all(\lambda y^i. P[x \leftarrow y])$$

Summary of Encoding Syntax

Object Language

Metalanguage

Syntactic category

Term, Prop

Summary of Encoding Syntax

Object Language

Metalanguage

Syntactic category

Type declaration $\{i, o\} \in \mathcal{B}$

Term, Prop

Variable x

Summary of Encoding Syntax

Object Language

Metalanguage

Syntactic category

Type declaration $\{i, o\} \in \mathcal{B}$

Term, *Prop*

Variable x

Variable x

Non-logical symb. $+$

Summary of Encoding Syntax

Object Language

Metalanguage

Syntactic category

Type declaration $\{i, o\} \in \mathcal{B}$

Term, *Prop*

Variable x

Variable x

Non-logical symb. $+$

1st-order constant $plus : i \rightarrow i \rightarrow i$

Logical symbol \wedge

Summary of Encoding Syntax

Object Language

Metalanguage

Syntactic category

Type declaration $\{i, o\} \in \mathcal{B}$

Term, *Prop*

Variable x

Variable x

Non-logical symb. $+$

1st-order constant $plus : i \rightarrow i \rightarrow i$

Logical symbol \wedge

1st-order constant $and : o \rightarrow o \rightarrow o$

Binding operator \forall

Summary of Encoding Syntax

Object Language

Metalanguage

Syntactic category

Type declaration $\{i, o\} \in \mathcal{B}$

$Term, Prop$

Variable x

Variable x

Non-logical symb. $+$

1st-order constant $plus : i \rightarrow i \rightarrow i$

Logical symbol \wedge

1st-order constant $and : o \rightarrow o \rightarrow o$

Binding operator \forall

2nd-order constant

$all : (i \rightarrow o) \rightarrow o$

Meaningful expr.

$a \wedge b \in Prop$

Summary of Encoding Syntax

Object Language	Metalanguage
Syntactic category $Term, Prop$	Type declaration $\{i, o\} \in \mathcal{B}$
Variable x	Variable x
Non-logical symb. $+$	1st-order constant $plus : i \rightarrow i \rightarrow i$
Logical symbol \wedge	1st-order constant $and : o \rightarrow o \rightarrow o$
Binding operator \forall	2nd-order constant $all : (i \rightarrow o) \rightarrow o$
Meaningful expr. $a \wedge b \in Prop$	Member of type $(and\ a\ b) : o$



More Detailed Explanations

Representing Syntax and Semantics

In the following, we will distinguish between the **object logic** and the **metalogic**. We have already seen this kind of distinction **before**.

The object logic, or user-defined theory if you like, has a syntax and has a notion of **proof**. Both must be represented in the metalogic. This is what this lecture and a **later lecture** are about.

Core

By the core we mean the syntax and proof rules of the metalogic. These should be simple, so that one can be reasonably confident that the implementation is correct.

Shared Support for Automation

There are some general techniques involved in automating the search for a proof that work for various object logics. It is therefore useful to implement these techniques on a higher level, rather than considering each object logic individually.

Conceptual Framework

By implementing various object logics within the same metalogic, we can compare the object logics in a more formal way.

Metalayer

Having a logic and a metalogic can be very mind-boggling. We already experienced that when working with Isabelle, it is sometimes confusing to know whether we are at the level of a particular theory, or at the level of general Isabelle syntax, or at the level of ML, the programming language that Isabelle is implemented in.

Assumptions

Designing a metalogic is a bold endeavor.

How are we supposed to know that the metalogic is expressive enough to encode any object logic someone might invent?

There is probably no general satisfactory answer to this question.

In fact, we make assumptions that object logics are of a certain kind.

This is related to the nature of implication. Roughly speaking, we assume logics and proof systems for which the **deduction theorem** holds, i.e., for which $A \vdash B$ (B is derivable under assumption A) holds if and only if $\vdash A \rightarrow B$ ($A \rightarrow B$ is derivable without any assumption).

There are logics (modal, relevance logics) for which the theorem does not hold [BM00].

Syntax Encoding

$\phi \in Prop$ iff $\ulcorner \phi \urcorner \in o$ means: The object level formula ϕ is a well-formed (according to the syntactic rules of the object logic) proposition if and only if its encoding in the metalogic, written $\ulcorner \phi \urcorner$, has type o .

Why First-Order?

We will now show an approach to representing the syntax an object logic in λ^{\rightarrow} . This approach is called **first-order syntax**. For the moment, it is better not to worry about why the approach is called that. This will become clear **later**.

Which Fragment?

We consider here the fragment of propositional logic containing the **logical symbols** $\neg, \wedge, \rightarrow$, and we call it *Prop*. We chose this small fragment because it is sufficient for our purposes, namely to demonstrate how encoding syntax in λ^{\rightarrow} works. It would be trivial to adapt everything in the sequel to include \vee or \perp .

Metalevel Constants

Now the object/meta distinction starts becoming mind-boggling!

We declare

$$\Sigma = \langle not : o \rightarrow o, and : o \rightarrow o \rightarrow o, imp : o \rightarrow o \rightarrow o \rangle,$$

and so on the level of our **metallogic** λ^{\rightarrow} , *not*, *and*, and *imp* are **constants**. However, these constants **represent** the **logical symbols** of the object logic.

Note the types of the constants:

not has type $o \rightarrow o$, so it takes a proposition and returns a proposition.
and and *imp* have type $o \rightarrow o \rightarrow o$, so each takes **two** propositions and returns a proposition.

Metalevel Variables

We identify metalevel variables and object level propositional variables. Hence Γ should contain expressions of the form $a : o$, where a is a λ^{\rightarrow} variable, representing a propositional variable. Note that under this agreement, Γ should **not** contain expressions like, e.g., $a : o \rightarrow o$.

Intuition for $a : o \vdash \text{imp} (\text{not } a) a : o$

$a : o \vdash \text{imp} (\text{not } a) a : o$ is a **judgement** in λ^{\rightarrow} , which may or may not be provable.

If we set up everything correctly and if $a : o \vdash \text{imp} (\text{not } a) a : o$ is provable, then the judgement represents the fact $\neg a \rightarrow a$ is a proposition. In this sense, we could then say that derivability in λ^{\rightarrow} captures the syntax of *Prop*, i.e., it can distinguish a legal proposition from a “non-proposition”.

Note that this has nothing to do with the question of whether it is a **true** proposition! So far, we are only talking about the representation of **syntax**.

Intuition for $a : o \vdash \text{not } (\text{imp } a) a : o$

$a : o \vdash \text{not } (\text{imp } a) a : o$ is a judgement in λ^{\rightarrow} which may or may not be provable.

If we set up everything correctly and if $a : o \vdash \text{not } (\text{imp } a) a : o$ is provable, then the judgement represents the fact that $(\rightarrow a) \neg a$ is a proposition.

However, you may observe that $(\rightarrow a) \neg a$ is gibberish. In fact, there is no formal sense whatsoever in saying that $\text{not } (\text{imp } a) a$ corresponds to $(\rightarrow a) \neg a$.

We will see that $a : o \vdash \text{not } (\text{imp } a) a : o$ isn't provable, and this reflects the fact that there is no proposition represented by $\text{not } (\text{imp } a) a$.

Non-existence of Proofs

Generally, it is difficult to prove that a proof of a given judgement within a given **proof system** does not exist, since there are infinitely many possible proofs and it is not obvious to predict how big an existing proof might be.

However, under certain conditions, there are techniques for simplifying proofs. In fact, there may be **normal form** proofs, i.e., proofs simplified as much as possible. One can then argue: if a proof of a certain judgement exists, it must be no bigger than a certain size. By searching through all proofs smaller than this size, one can prove that no proof exists.

In this lecture, we do not go into the details of this topic [[GLT89](#), [Pra65](#)].

Bijection

In general mathematical terminology, a **bijection** between A and B is a mapping $f : A \rightarrow B$ such that for all $a, a' \in A$, where $a \neq a'$, we have $f(a) \neq f(a')$, and for each $b \in B$, there exists an $a \in A$ such that $f(a) = b$.

For a bijection f , the inverse f^{-1} is always defined, and we have $f(f^{-1}(b)) = b$ for all $b \in B$ and $f^{-1}(f(a)) = a$ for all $a \in A$.

Proof of Adequacy

If $P \in Prop$, and if for each propositional variable x in P , we have $x : o \in \Gamma$, then $\Gamma \vdash \ulcorner P \urcorner : o$.

Proof: By structural induction on $Prop$.

Base case: P is a propositional variable.

Then $\ulcorner P \urcorner = P$, and so if $P : o \in \Gamma$, then we have $\Gamma \vdash \ulcorner P \urcorner : o$ by rule *hyp*.

Induction step: Suppose the claim holds for $P \in Prop$ and $Q \in Prop$.

Consider the propositional formula $\neg P$. We have $\ulcorner \neg P \urcorner = \text{not } \ulcorner P \urcorner$.

Assume that for each propositional variable x in P , we have $x : o \in \Gamma$.

By the induction hypothesis, $\Gamma \vdash \ulcorner P \urcorner : o$. Moreover $\Gamma \vdash \text{not} : o \rightarrow o$ by rule *assum*, and so $\Gamma \vdash \text{not } \ulcorner P \urcorner : o$ by rule *app*.

Now consider the propositional formula $P \wedge Q$. We have

$\ulcorner P \wedge Q \urcorner = \text{and } \ulcorner P \urcorner \ulcorner Q \urcorner$. Assume that for each propositional variable x

in P or Q , we have $x : o \in \Gamma$. By the induction hypothesis, $\Gamma \vdash \ulcorner P \urcorner : o$ and $\Gamma \vdash \ulcorner Q \urcorner : o$. Moreover $\Gamma \vdash \text{and} : o \rightarrow o \rightarrow o$ by rule *assum*, and so $\Gamma \vdash \text{and} \ulcorner P \urcorner \ulcorner Q \urcorner : o$ by two applications of rule *app*.

The case $P \rightarrow Q$ is completely analogous.

Why $\ulcorner \ulcorner P \urcorner \urcorner^{-1} = P$?

By the definition of *Prop* and the definition of $\ulcorner \cdot \urcorner$, it is clear that $\ulcorner P \urcorner$ is defined for all $P \in Prop$. It is very easy to show by induction on *Prop* that $\ulcorner \ulcorner P \urcorner \urcorner^{-1} = P$.

Here is an example of a proof by induction on *Prop*.

Obviously, everything we say here depends on the particular fragment of propositional logic, but in an inessential way. It would be trivial to adapt to other fragments.

Canonical $\beta\eta$ -Long Normal Form

A **canonical $\beta\eta$ -long normal form** of a λ -term is obtained by applying first β -reduction as long as possible, and then η -expansion (η -conversion applied from right to left) as long as possible.

You may think: Why should the process of η -expansion ever stop? Can't I always replace a λ -term M with $\lambda x. (Mx)$? Since we are in the **typed** λ -calculus, the answer is no! Consider this example:

not can be expanded to $\lambda x. not\ x$ since *not* is of **function type**: it **has type** $o \rightarrow o$. Therefore, *not* x can be **assigned a type**:

$$\frac{\frac{\Gamma \vdash not : o \rightarrow o}{\Gamma, x : o \vdash not\ x : o} \text{abs}}{\Gamma \vdash \lambda x. not\ x : o \rightarrow o} \text{app}$$

But now, we cannot continue, say by expanding to $\lambda y. (\lambda x. not\ x) y$.

$\lambda x. not\ x$ is not of function type, and so it is impossible to assign a type to $(\lambda x. not\ x)\ y$.

Effectively, when a term of type $\tau_1 \rightarrow \tau_n \rightarrow \tau$ is η -expanded, it will have the form $\lambda x_1 x_2 \dots x_n. e$.

Normal forms are **unique**.

Variables in Γ

Saying that a propositional formula has variables in Γ is an abuse of terminology, i.e., it isn't exactly true, but it is trusted that the reader can guess the exact formulation.

What we mean is: a propositional formula such that for each propositional variable x occurring in the formula, we have $x : o \in \Gamma$.

Simplifying Proofs

What this picture says is that if the left hand side is a fragment from a proof tree, deriving the judgement $\vdash (\lambda x^\sigma. e)e' : \tau$, then there exists a proof of the judgement $\vdash e[x \leftarrow e'] : \tau$.

Be aware however that our argument here is very sketchy. We do not go into the details in this course.

$$t : o$$

Simply writing $t : o$ is again a bit sloppy. We should write: $\Gamma \vdash t : o$ for some Γ containing only expressions of the form $x : o$, where x is a propositional variable in $Prop$.

Two Times First-Order!

In the [previous section](#), we have seen how we can use **first-order syntax** (of λ^{\rightarrow}) to represent the syntax of an object logic, then *Prop*. We [haven't really understood yet](#) why we speak of **first-order** syntax, but note that the notion “first-order” refers to λ^{\rightarrow} , i.e., the metalevel.

We will now consider first-order logic as **object** language. So we will now attempt to represent the syntax of **first-order** logic (the object language) using **first-order** λ^{\rightarrow} syntax (the metalanguage). To avoid confusion, it is best to imagine that it is a mere coincidence that both the object and the [metalanguage](#) are described as “first-order”. Of course there are reasons why both languages are called like that, but it is best to understand this separately for both levels. We will come back to this.

Specifying a First-Order Language

With this grammar, we specify a certain language of a fragment (since quantifiers, \forall , and \perp are missing) of first-order logic.

Alternatively, we could say that $\mathcal{F} = \{0, s, +, \times\}$ and $\mathcal{P} = \{=\}$.

However, the way we defined first-order logic, the language thus obtained would also include quantifiers, \forall , and \perp . For the moment we want to restrict ourselves to the fragment given by the grammar for FOA.

\mathcal{S}

s is a unary **prefix** function, so s applied to T is written sT .

Some Intuition for FOA

We have defined

$$\Sigma_T = \langle \textit{zero} : i, \textit{succ} : i \rightarrow i, \textit{plus} : i \rightarrow i \rightarrow i, \textit{times} : i \rightarrow i \rightarrow i \rangle$$

Some Intuition for FOA

We have defined

$$\Sigma_T = \langle \textit{zero} : i, \textit{succ} : i \rightarrow i, \textit{plus} : i \rightarrow i \rightarrow i, \textit{times} : i \rightarrow i \rightarrow i \rangle$$

$\textit{zero} : i$ means: viewed on the object level, 0 is a term.

$\textit{plus} : i \rightarrow i \rightarrow i$ means: viewed on the object level, \textit{plus} is a function that takes **two** terms and returns a term. $\textit{eq} : i \rightarrow i \rightarrow o$ means: viewed on the object level, $=$ is a predicate that takes **two** terms and returns a proposition.

On the metalevel (level of λ^{\rightarrow}), \textit{zero} , \textit{plus} and \textit{eq} are constants. Note that we could also formalize them as variables.

Recall that we encoded the **non-logical** symbols of an object logic as constants. It would however be possible to set up the encoding in such a

way that the non-logical symbols are encoded as variables, so we would have a context Γ_T instead of our Σ_T . This is in line with [Perlis' epigram](#). We will sometimes take this approach in the exercises as the encoding of LF in Isabelle makes it more straightforward to play around with different Γ 's than with different Σ 's.

Definition of $\ulcorner \cdot \urcorner$

We extend the definition of $\ulcorner \cdot \urcorner$ as follows:

$$\ulcorner x \urcorner = x$$

$$\ulcorner 0 \urcorner = \textit{zero}$$

$$\ulcorner s \ t \urcorner = \textit{succ} \ulcorner t \urcorner$$

$$\ulcorner r + t \urcorner = \textit{plus} \ulcorner r \urcorner \ulcorner t \urcorner$$

$$\ulcorner r \times t \urcorner = \textit{times} \ulcorner r \urcorner \ulcorner t \urcorner$$

Note that here, on the object level, x is a first-order variable (a variable is a **term**), and hence on the metalevel, it has type i .

Subtypes?

In first-order logic, variables are not a **syntactic category** of their own, but rather they are a “sub-category” of **terms**. Therefore one should expect that *var* should be a “subtype” of *i*, that is to say, every term of type *var* is automatically also of type *i*. However, there is no such notion in λ^{\rightarrow} .

$$(and\ (p\ x)(all\ x\ (q\ x)))[x \leftarrow a]$$

There is a notion of **substitution** in λ^{\rightarrow} , hence on the metalevel. But *all* is just a constant like any other on the level of λ^{\rightarrow} , and hence $(and\ (p\ x)(all\ x\ (q\ x)))[x \leftarrow a] = (and\ (p\ a)(all\ a\ (q\ a)))$, and not $(and\ (p\ a)(all\ x\ (q\ x)))$ as **one should expect**.

That is to say, the standard operation of substitution, which exists on the metalevel, is of no use for implementing substitution on the object level. Instead, substitution on the object level must be “programmed explicitly”.

Note that the following question arises: on the λ^{\rightarrow} level, should the terms of type *var* be variables or constants?

One could imagine that they are variables. This means that the **signature** Σ would not contain any constants of type *var* or $\dots \rightarrow var$. The only terms of type *var* would be variables. In this case, a λ^{\rightarrow} term like

$(\text{and } (p\ x)(\text{all } x\ (q\ x)))$ could only be typed in a context Γ containing $x : \text{var}$.

Alternatively, one could imagine that they are constants. The signature **signature** Σ would contain expressions of the form $x : \text{var}$, where x would be a λ^{\rightarrow} constant. One thing that isn't nice about this approach is that Σ cannot be an infinite sequence, and so we would have to fix a finite set of variables that can be represented in λ^{\rightarrow} .

In either case, the operation of substitution on the metalevel is of no use for implementing substitution on the object level.

$$all(\lambda x^i. eq\ x\ x : i \rightarrow o)$$

Some intuition: a proposition is represented by a term of type o . Now a term of type $i \rightarrow o$ represents a proposition where some positions are marked in a special way. For example, in $\lambda x^i. eq\ x\ x$, the positions where x occurs are marked in a special way, by virtue of the fact that the λ in front of the expression binds the x . This “marking” allows us to “insert” other terms in place of x . We will see this [soon](#).

all is a constant which can be applied to a term of type $i \rightarrow o$.

Faithfulness and Adequacy

Terms and formulae are represented by (canonical) members of i and o .
The principle is similar as for *Prop*.

Intuition for Order

A term of first-order type is a function taking (an arbitrary number of) arguments all of which must be of base type.

A term of second-order type is a function taking (an arbitrary number of) arguments some of which may be functions (of first order type).

A term of third-order type is a function taking (an arbitrary number of) arguments some of which may be functions, which again take functions (of first order type) as arguments.

...

Obviously, it would be wrong to think of the order as “number of arrows in a type”. Instead, one can think of order as the “nesting depth of arrows in a type”.

Sometimes, the notion “second-order” is used in the context of type theories for quite a different concept, but we will avoid that other use

here.

Propositional vs. First-Order Variables

Although **propositional variables** and **first-order variables** are quite different concepts, for both the representation in λ^{\rightarrow} used λ^{\rightarrow} variables. Technically however, there is a difference between the representations of **propositional variables** and **first-order variables**. In particular, propositional variables are represented as λ^{\rightarrow} variables of type o , and first-order variables are represented as λ^{\rightarrow} variables of type i .

Resolution

Three Sections on Deduction Techniques

After encoding **syntax**, the next topic in the theory is encoding **proofs** (very challenging!). But before, we look at some more practical issues:

- Resolution
- Proof search
- Term rewriting

We will explain many techniques relevant for Isabelle, but not in extreme detail and rigor. We want to understand better how Isabelle works, but not provide a formal proof that she works correctly, or be able to rebuild her.

Resolution

Resolution is the basic mechanism for transforming proof states in Isabelle in order to construct a proof.

It involves **unifying** a certain part of the current goal (state) with a certain part of a rule, and replacing that part of the current goal.

We have already explained this in the labs and you have been working with it all the time, but now we want to understand it more thoroughly.

We look at several variants of resolution.

Resolution (rtac, as in Prolog)

ϕ_1, \dots, ϕ_n are current subgoals and ψ is original goal.
Isabelle displays

Level ... (n subgoals)

ψ

1. ϕ_1

\vdots

$n.$ ϕ_n

$$\frac{\phi_1 \quad \dots \quad \phi_i \quad \dots \quad \phi_n}{\psi}$$

Resolution (rtac, as in Prolog)

$$\frac{\phi_1 \quad \dots \quad \phi_i \quad \dots \quad \phi_n}{\psi}$$

$$\frac{\alpha_1 \dots \alpha_m}{\beta}$$

ϕ_1, \dots, ϕ_n are current subgoals and ψ is original goal. Isabelle displays

Level ... (n subgoals)

ψ

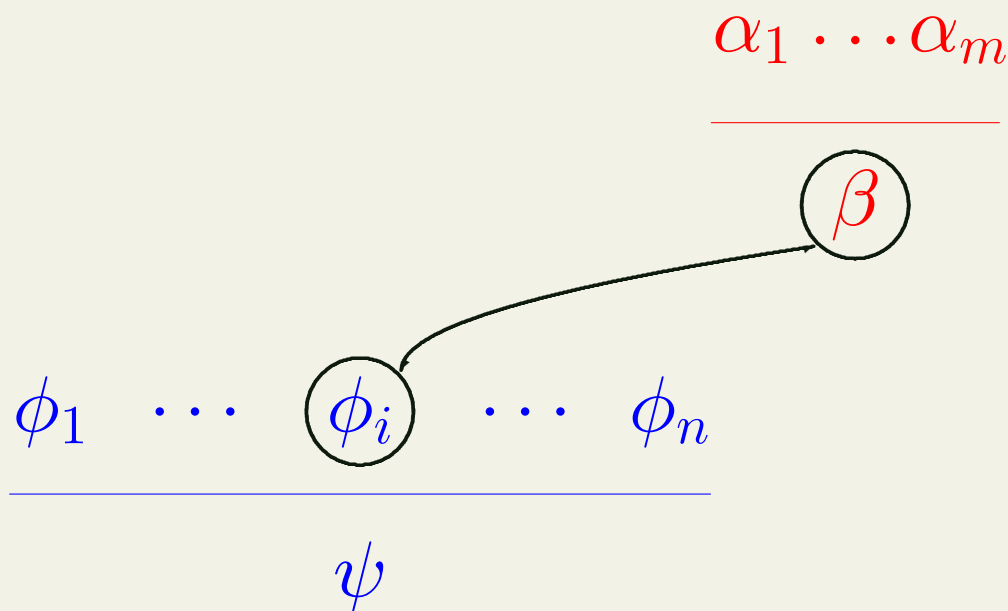
1. ϕ_1

⋮

$n.$ ϕ_n

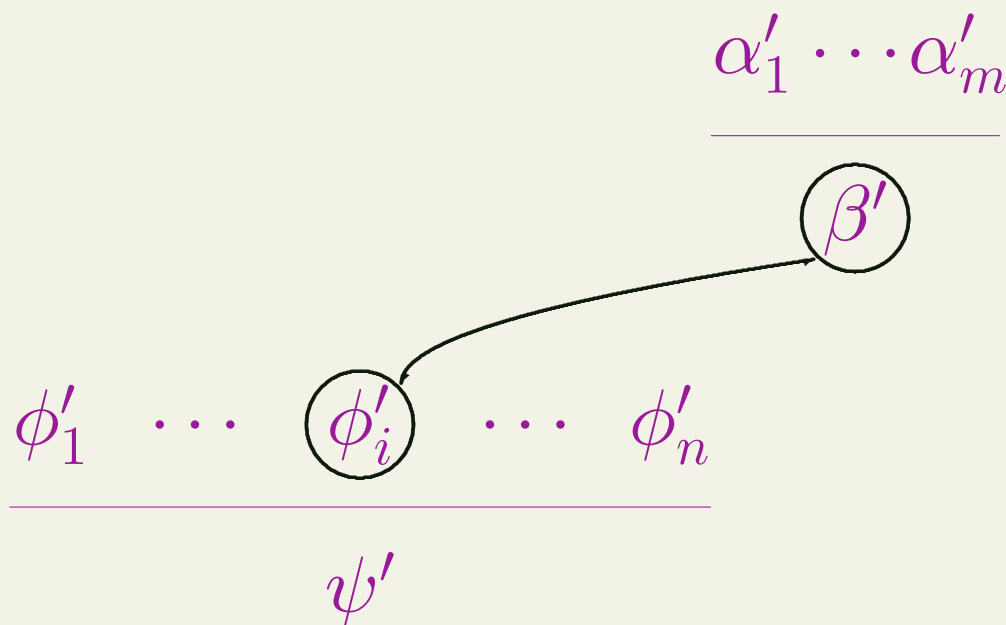
$[[\alpha_1; \dots; \alpha_m]] \implies \beta$ is rule.

Resolution (rtac, as in Prolog)



Simple scenario where ϕ_i has no premises. Now β must be unifiable with selected subgoal ϕ_i .

Resolution (rtac, as in Prolog)



Simple scenario where ϕ_i has no premises. Now β must be unifiable with selected subgoal ϕ_i .

We apply the unifier ($'$)

Resolution (rtac, as in Prolog)

Simple scenario where ϕ_i has no premises. Now β must be unifiable with selected subgoal ϕ_i .

$$\frac{\phi'_1 \cdots \alpha'_1 \cdots \alpha'_m \cdots \phi'_n}{\psi'}$$

We apply the unifier (')

We replace ϕ'_i by the premises of the rule.

Resolution (with Lifting over Parameters)

$$\frac{\phi_1 \quad \cdots \quad \bigwedge x. \phi_i \quad \cdots \quad \phi_n}{\psi}$$

Now suppose the i 'th (selected) subgoal is preceded by \bigwedge (metalevel universal quantifier).

Resolution (with Lifting over Parameters)

$$\alpha_1 \quad \dots \quad \alpha_m$$

$$\beta$$

$$\phi_1 \quad \dots \quad \bigwedge x. \phi_i \quad \dots \quad \phi_n$$

$$\psi$$

Rule

Resolution (with Lifting over Parameters)

$$\bigwedge x. \alpha_1[x] \cdots \bigwedge x. \alpha_m[x]$$

$$\bigwedge x. \beta[x]$$

$$\phi_1 \quad \cdots \quad \bigwedge x. \phi_i \quad \cdots \quad \phi_n$$

$$\psi$$

Rule is **lifted** over x : Apply $[?X \leftarrow ?X(x)]$.

Resolution (with Lifting over Parameters)

$$\begin{array}{c}
 \Lambda x. \alpha_1[x] \cdots \Lambda x. \alpha_m[x] \\
 \hline
 \Lambda x. \beta[x] \\
 \swarrow \\
 \phi_1 \quad \cdots \quad \Lambda x. \phi_i \quad \cdots \quad \phi_n \\
 \hline
 \psi
 \end{array}$$

Rule is **lifted** over x : Apply $[?X \leftarrow ?X(x)]$.

As before, β must be unifiable with ϕ_i ;

Resolution (with Lifting over Parameters)

$$\begin{array}{c}
 \frac{\bigwedge x. \alpha'_1[x] \cdots \bigwedge x. \alpha'_m[x]}{\bigwedge x. \beta'[x]} \\
 \\
 \frac{\phi'_1 \cdots \bigwedge x. \phi'_i \cdots \phi'_n}{\psi'}
 \end{array}$$

Rule is **lifted** over x : Apply $[?X \leftarrow ?X(x)]$.

As before, β must be unifiable with ϕ_i ; apply the **unifier**.

Resolution (with Lifting over Parameters)

$$\frac{\phi'_1 \cdots \wedge x.\alpha'_1[x] \cdots \wedge x.\alpha'_m[x] \cdots \phi'_n}{\psi'}$$

Rule is **lifted** over x : Apply $[?X \leftarrow ?X(x)]$.

As before, β must be unifiable with ϕ_i ; apply the **unifier**.

We replace ϕ'_i by the premises of the rule. $\alpha'_1, \dots, \alpha'_m$ are preceded by $\wedge x$.

Resolution (with Lifting over Assumptions)

$$\begin{array}{ccccccc}
 & & [\phi_{i1} \cdots \phi_{ik_i}] & & & & \\
 & & \vdots & & & & \\
 \phi_1 & \cdots & \phi_i & \cdots & \phi_n & & \\
 \hline
 & & \psi & & & &
 \end{array}$$

Now, suppose the i 'th (selected) subgoal has assumptions $\phi_{i1}, \dots, \phi_{ik_i}$.

$$\begin{array}{c}
 \alpha_1 \quad \dots \quad \alpha_m \\
 \hline
 [\phi_{i1} \cdots \phi_{ik_i}] \\
 \vdots \\
 \phi_1 \quad \dots \quad \phi_i \quad \dots \quad \phi_n \\
 \hline
 \psi
 \end{array}$$

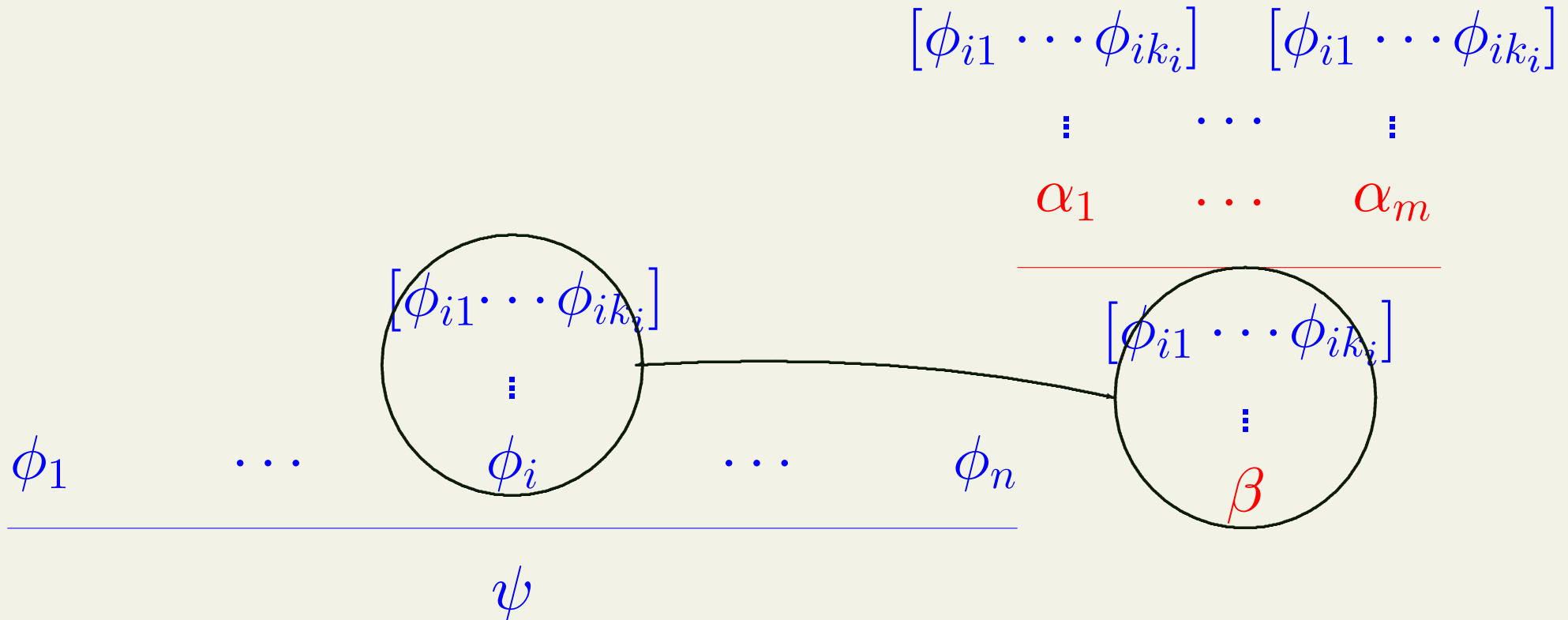
Smaus: CSMR; WS04/05

Resolution (with Lifting over Assumptions)

$$\begin{array}{c}
 \begin{array}{c}
 [\phi_{i1} \cdots \phi_{ik_i}] \\
 \vdots \\
 \alpha_1 \quad \cdots \quad \alpha_m
 \end{array} \\
 \hline
 \begin{array}{c}
 [\phi_{i1} \cdots \phi_{ik_i}] \\
 \vdots \\
 \beta
 \end{array}
 \end{array}
 \quad
 \begin{array}{c}
 [\phi_{i1} \cdots \phi_{ik_i}] \\
 \vdots \\
 \phi_1 \quad \cdots \quad \phi_i \quad \cdots \quad \phi_n \\
 \hline
 \psi
 \end{array}$$

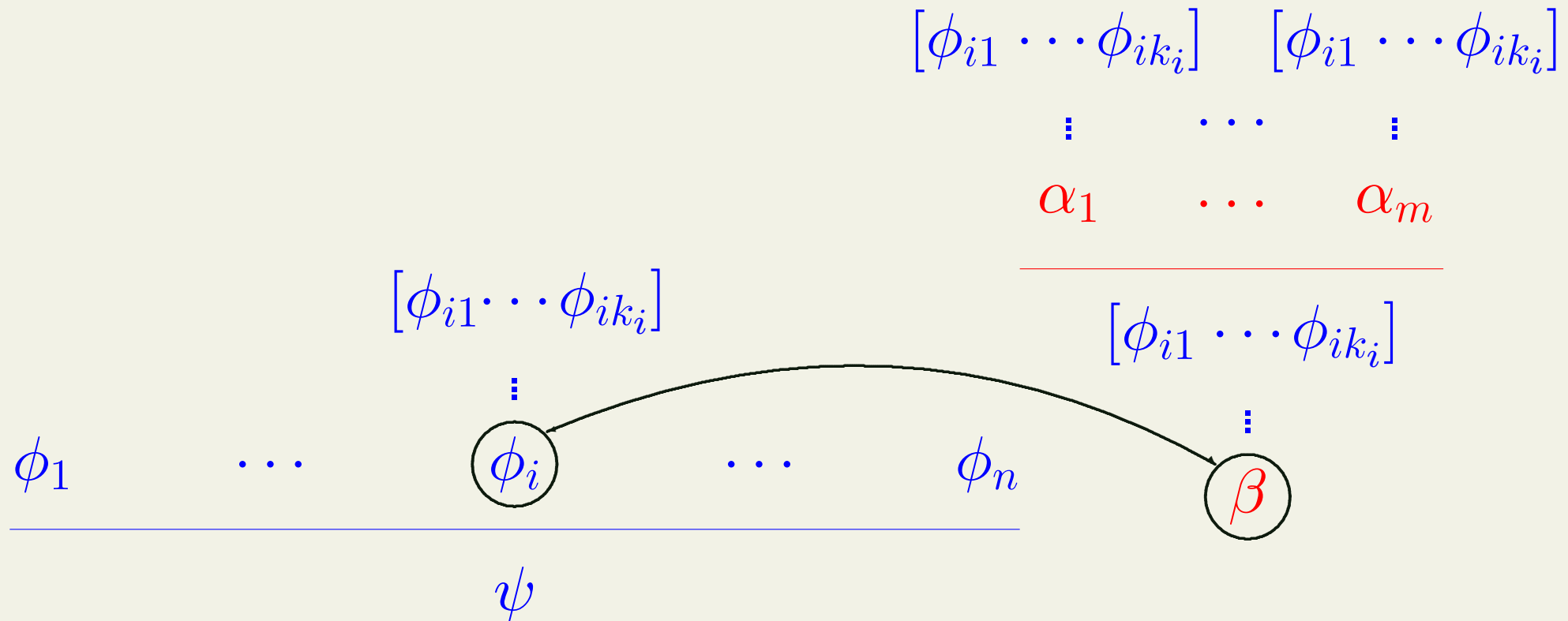
Rule must be lifted over assumptions. No unification so far!

Resolution (with Lifting over Assumptions)



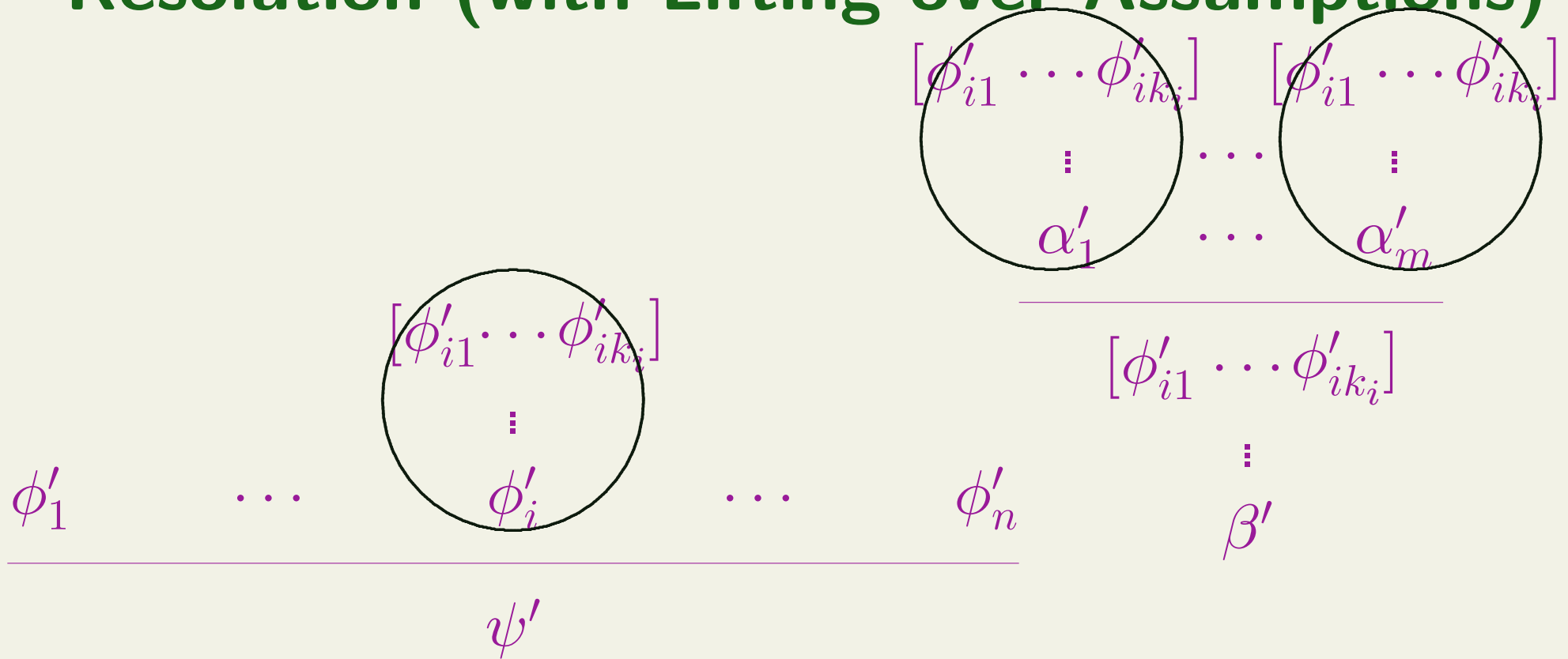
Now, subgoal and rule conclusion (below the bar) are **unifiable**.

Resolution (with Lifting over Assumptions)



Now, subgoal and rule conclusion (below the bar) are **unifiable**.
 Non-trivially, β must be unifiable with ϕ_i .

Resolution (with Lifting over Assumptions)



We apply the **unifier**.

Resolution (with Lifting over Assumptions)

$$\begin{array}{c}
 [\phi'_{i1} \cdots \phi'_{ik_i}] \quad [\phi'_{i1} \cdots \phi'_{ik_i}] \\
 \vdots \quad \cdots \quad \vdots \\
 \phi'_1 \cdots \phi'_{i-1} \quad \alpha'_1 \quad \cdots \quad \alpha'_m \quad \phi'_{i+1} \cdots \phi'_n \\
 \hline
 \psi'
 \end{array}$$

We replace the subgoal.

Rule Premises Containing \implies

$$[\phi'_{i1} \cdots \phi'_{ik_i}]$$

$$\vdots$$

$$\phi'_1 \cdots \quad \alpha'_j \quad \cdots \phi'_n$$

$$\psi'$$

What if some α'_j has the form $[\gamma_1; \dots; \gamma_l] \implies \delta$?

Rule Premises Containing \implies

$$[\phi'_{i1} \cdots \phi'_{ik_i}]$$

$$\vdots$$

$$\phi'_1 \cdots [\gamma_1; \dots; \gamma_l] \implies \delta \cdots \phi'_n$$

$$\psi'$$

What if some α'_j has the form $[\gamma_1; \dots; \gamma_l] \implies \delta$?

Is this what we get?

Rule Premises Containing \implies

$$[\phi'_{i_1} \cdots \phi'_{i_{k_i}}; \gamma'_1 \cdots \gamma'_l]$$

$$\vdots$$

$$\phi'_1 \cdots \delta' \cdots \phi'_n$$

$$\psi'$$

What if some α'_j has the form $[\gamma_1; \dots; \gamma_l] \implies \delta$?

Is this what we get?

Well, we write $:$ for \implies , and use

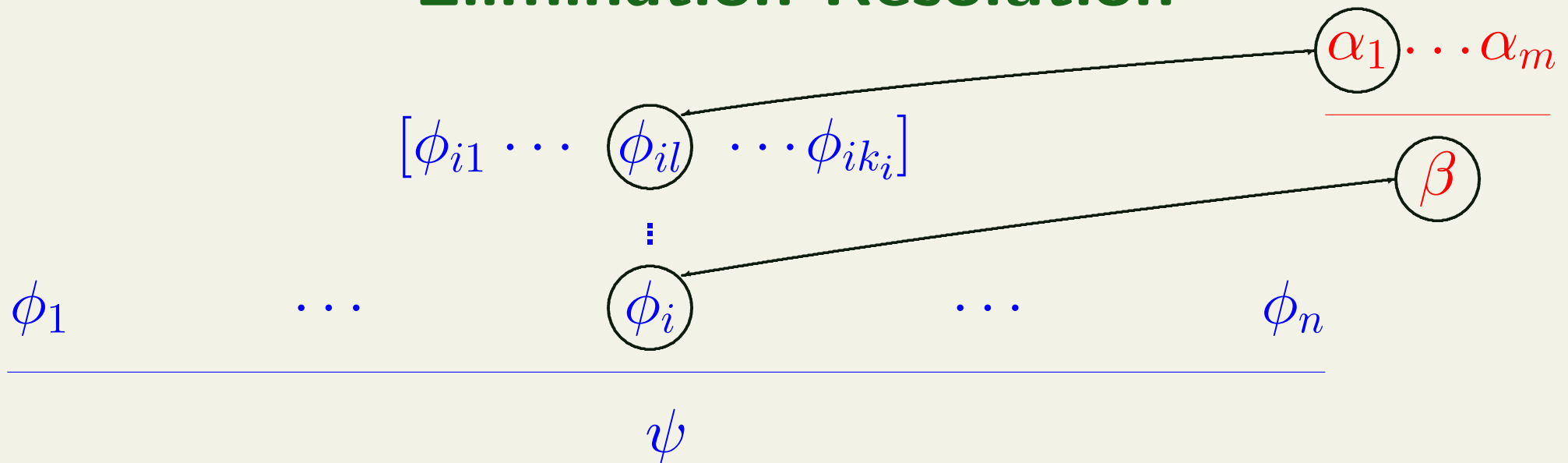
$$A \implies B \implies C \equiv [A; B] \implies C.$$

Elimination-Resolution

$$\begin{array}{c}
 \alpha_1 \cdots \alpha_m \\
 \hline
 \beta \\
 \\
 \begin{array}{ccccccc}
 & & [\phi_{i1} \cdots \phi_{il} \cdots \phi_{ik_i}] & & & & \\
 & & \vdots & & & & \\
 \phi_1 & \cdots & \phi_i & \cdots & \phi_n & & \\
 \hline
 & & \psi & & & &
 \end{array}
 \end{array}$$

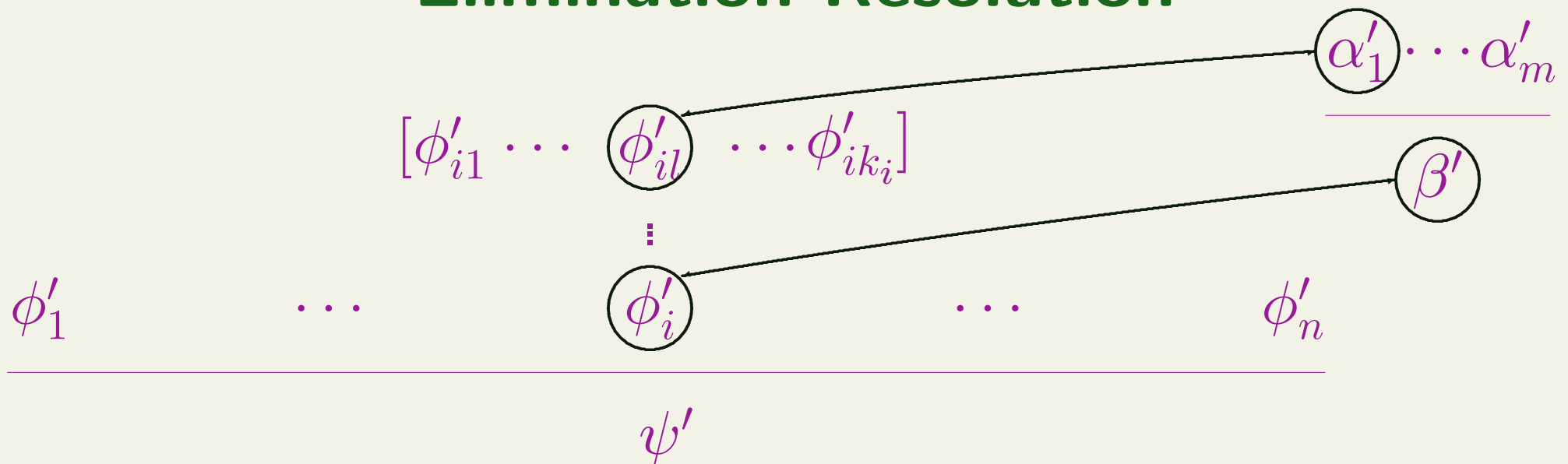
Same scenario as before

Elimination-Resolution



Same scenario as before, but now β must be unifiable with ϕ_i , and α_1 must be unifiable with ϕ_{il} , for some l .

Elimination-Resolution



Same scenario as before, but now β must be unifiable with ϕ_i , and α_1 must be unifiable with ϕ_{il} , for some l .

Apply the **unifier**.

Elimination-Resolution

$$\begin{array}{ccc}
 [\phi'_{i1} \cdots \phi'_{i,l-1}, \phi'_{i,l+1} \cdots \phi'_{ik_i}] & & [\phi'_{i1} \cdots \phi'_{i,l-1}, \phi'_{i,l+1} \cdots \phi'_{ik_i}] \\
 \vdots & & \vdots \\
 \phi'_1 \cdots \phi'_{i-1} \alpha'_2 & \cdots & \alpha'_m \phi'_{i+1} \cdots \phi'_n
 \end{array}$$

$$\psi'$$

Same scenario as before, but now β must be unifiable with ϕ_i , and α_1 must be unifiable with ϕ_{il} , for some l .

Apply the **unifier**.

We replace ϕ'_i by the premises of the rule **except** first premise. $\alpha'_2, \dots, \alpha'_m$ inherit the assumptions of ϕ'_i , **except** ϕ'_{il} .

Destruct-Resolution

 α

$$[\phi_{i1} \cdots \phi_{il} \cdots \phi_{ik_i}] \quad \frac{\alpha}{\beta}$$

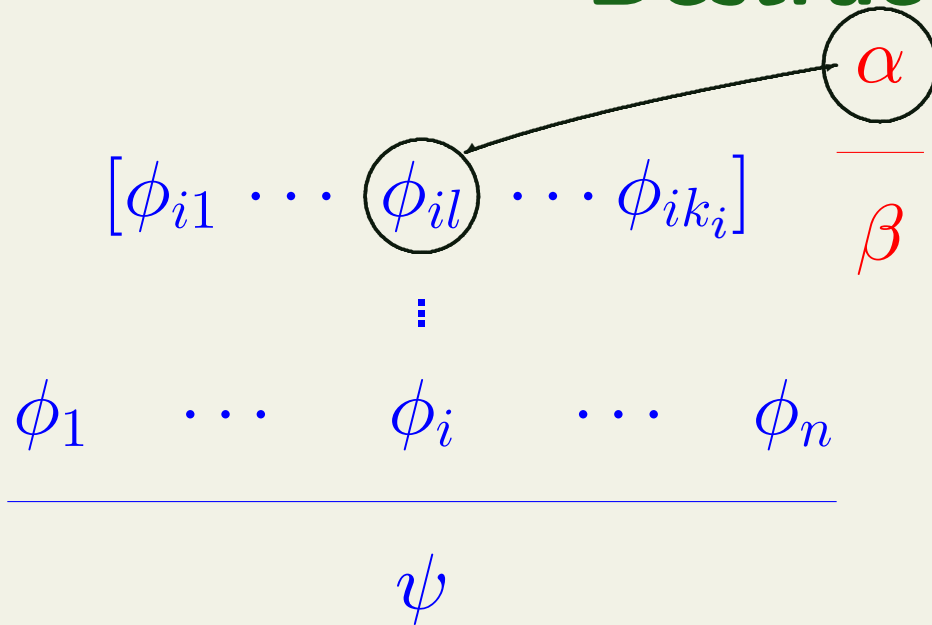
 β
 \vdots

$$\phi_1 \cdots \phi_i \cdots \phi_n$$

 ψ

Simple rule

Destruct-Resolution



Simple rule, and α must be unifiable with ϕ_{il} , for some l .

Destruct-Resolution

$$\begin{array}{c}
 \begin{array}{c}
 \alpha' \\
 \circlearrowleft
 \end{array} \\
 \swarrow \\
 [\phi'_{i1} \cdots \phi'_{il} \cdots \phi'_{ik_i}] \quad \frac{\quad}{\beta'} \\
 \vdots \\
 \phi'_1 \quad \cdots \quad \phi'_i \quad \cdots \quad \phi'_n \\
 \hline
 \psi'
 \end{array}$$

Simple rule, and α must be unifiable with ϕ_{il} , for some l .

We apply the **unifier**.

Destruct-Resolution

$$\begin{array}{c}
 [\phi'_{i1} \cdots \beta' \cdots \phi'_{ik_i}] \\
 \vdots \\
 \phi'_1 \cdots \phi'_i \cdots \phi'_n \\
 \hline
 \psi'
 \end{array}$$

Simple rule, and α must be unifiable with ϕ_{il} , for some l .

We apply the **unifier**.

We **replace** premise ϕ'_{il} with the **conclusion** of the rule.

Summary on Resolution

- Build proof resembling **sequent style notation**;
- technically: replace **goals** with rule **premises**, or goal **premises** with rule **conclusions**;
- metavariables and **unification** to obtain appropriate instance of rule, delay commitments;
- lifting over **parameters** and **assumptions**;
- various techniques to manipulate premises or conclusions, as convenient: **rtac**, **etac**, **dtac**.



More Detailed Explanations

Prolog

Prolog is a logic programming language [Apt97].

The computation mechanism of Prolog is resolution of a current goal (corresponding to our ϕ_1, \dots, ϕ_n) with a Horn clause (corresponding to our $[\alpha_1; \dots; \alpha_m] \implies \beta$).

Simple ϕ_i

ϕ_i is the selected subgoal. In Isabelle, the number i of the selected subgoal is always one of the arguments of a tactic. One writes:

by (*tactic rule i*);

We assume here that ϕ_i is a formula, i.e., it contains no \implies (metalevel implication). The form of the other subgoals $\phi_1, \dots, \phi_{i-1}, \phi_{i+1}, \dots, \phi_n$ is arbitrary.

Prime (')

In all illustrations that follow, we use ' to suggest the application of the appropriate unifier.

Metalevel Universal Quantification

\bigwedge is the metalevel universal quantification (also written $!!$). If a goal is preceded by $\bigwedge x$, this means that Isabelle must be able to prove the subgoal in a way which is independent from x , i.e., without instantiating x .

Lifting over Parameters

The metavariables of the rule are made dependent on x . That is to say, each metavariable $?X$ is replaced by a $?X(x)$. You may also say that $?X$ is now a **Skolem** function of x .

This process is called **lifting the rule over the parameter x** .

We denote by $\rho[x]$ the result of lifting ρ over x .

Non-unifiability

The selected subgoal is $[[\phi_{i1}, \dots, \phi_{ik_i}]] \implies \phi_i$ where $\phi_{i1}, \dots, \phi_{ik_i}, \phi_i$ are object-level formulae. So the selected subgoal is not an object-level formula, but it has \implies as “top-level constructor” and is hence a formula in the metalogic.

Moreover, β is a formula. It is clear that an object-level formula cannot be unifiable with a formula in the metalogic having \implies as “top-level constructor”.

Lifting over Assumptions

Each premise of the rule, as well as the conclusion of the rule, are preceded by the assumptions $[\phi_{i1}, \dots, \phi_{ik_i}]$ of the current subgoals. Actually, the rule

$$\begin{array}{c}
 [\phi_{i1} \cdots \phi_{ik_i}] \quad [\phi_{i1} \cdots \phi_{ik_i}] \\
 \vdots \quad \cdots \quad \vdots \\
 \alpha_1 \quad \cdots \quad \alpha_m \\
 \hline
 [\phi_{i1} \cdots \phi_{ik_i}] \\
 \vdots \\
 \beta
 \end{array}$$

may look different from any rules you have seen so far, but it can be formally derived from the rule:

$$\frac{\alpha_1 \quad \dots \quad \alpha_m}{\beta}$$

The derived rule should be read as: If for all $j \in \{1, \dots, m\}$, we can derive α_j from $\phi_{i1}, \dots, \phi_{ik_i}$, then we can derive β from $\phi_{i1}, \dots, \phi_{ik_i}$.

Unifiability

Still assuming that ϕ_i and β are unifiable.

A Trivial Unification

Both the subgoal and the conclusion of the lifted rule are preceded by assumptions $\phi_{i1}, \dots, \phi_{ik_i}$. Hence the assumption list of the subgoal and the assumption list of the rule are trivially unifiable since they are identical.

Folding Assumptions

Generally, Isabelle makes no distinction between

$$\llbracket \psi_1; \dots; \psi_n \rrbracket \Longrightarrow \llbracket \mu_1; \dots; \mu_k \rrbracket \Longrightarrow \phi$$

and

$$\llbracket \psi_1; \dots; \psi_n; \mu_1; \dots; \mu_k \rrbracket \Longrightarrow \phi$$

and displays the second form. Semantically, this corresponds to the equivalence of $A_1 \wedge \dots \wedge A_n \rightarrow B$ and $A_1 \rightarrow \dots \rightarrow A_n \rightarrow B$.

We have seen this in the exercises.

Same as Resolution

So the scenario looks as for resolution with **lifting over assumptions**. However, this time we do not show the lifting over assumptions in our animation.

The Rationale of Elimination-Resolution

Elimination-resolution is used to **eliminate a connective in the premises**.

For example, if the current goal is

$$\frac{\begin{array}{c} [A \wedge B] \\ \vdots \\ B \end{array}}{A \wedge B \rightarrow B}$$

and the rule is

$$\frac{\begin{array}{c} [P; Q] \\ \vdots \\ R \end{array} \quad \frac{P \wedge Q}{R}}{\wedge-E}$$

then the result of elimination resolution is

$$\frac{\begin{array}{c} [A; B] \\ \vdots \\ B \end{array}}{A \wedge B \rightarrow B}$$

Effectively, the interplay between elimination rules and elimination-resolution is such that one “does not throw any information away”. Before we had the assumption $A \wedge B$. This was replaced by the components A and B as separate assumptions.

The Rationale of Destruct-Resolution

Destruct-resolution is used to **eliminate a connective in the premises**. The difference compared to **elimination-resolution** can be seen in the following example. Unlike elimination-resolution, destruct-resolution “throws information away”.

For example, if the current goal is

$$\frac{\begin{array}{c} [A \wedge B] \\ \vdots \\ B \end{array}}{A \wedge B \rightarrow B}$$

and the rule is

$$\frac{P \wedge Q}{Q} \text{conjunct2}$$

then the result of destruct-resolution is

$$\frac{\begin{array}{c} [B] \\ \vdots \\ B \end{array}}{A \wedge B \rightarrow B}$$

If we had instead used rule

$$\frac{P \wedge Q}{P} \text{conjunct2}$$

the result would have been

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \wedge B \rightarrow B}$$

and we would be stuck. We accidentally “threw away” the assumption B .

Automation by Proof Search

Outline of this Part

- Proof search (à la tableaux proving) and backtracking
- Classifying rules
- Proof procedures

Proof Search and Backtracking

- Need for more automation
- Some aspects in proof construction are highly non-deterministic:
 - unification: which unifier to choose?
 - resolution: where to apply a rule (which 'subgoal')?
 - which rule to apply?
- How to organize proof-search technically?

Organizing Proof Search Conceptually

Organize proof search as a **tree of theorems** (thm's).

A sketch of an exemplary proof search:

Organizing Proof Search Conceptually

Organize proof search as a **tree** of **theorems** (thm's).

A sketch of an exemplary proof search:

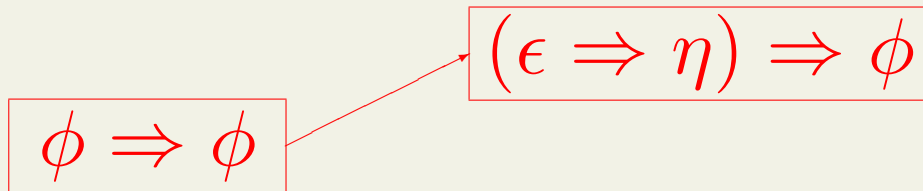
$$\phi \Rightarrow \phi$$

Goal ϕ will create the proof state $\phi \Rightarrow \phi$.

Organizing Proof Search Conceptually

Organize proof search as a **tree** of **theorems** (thm's).

A sketch of an exemplary proof search:

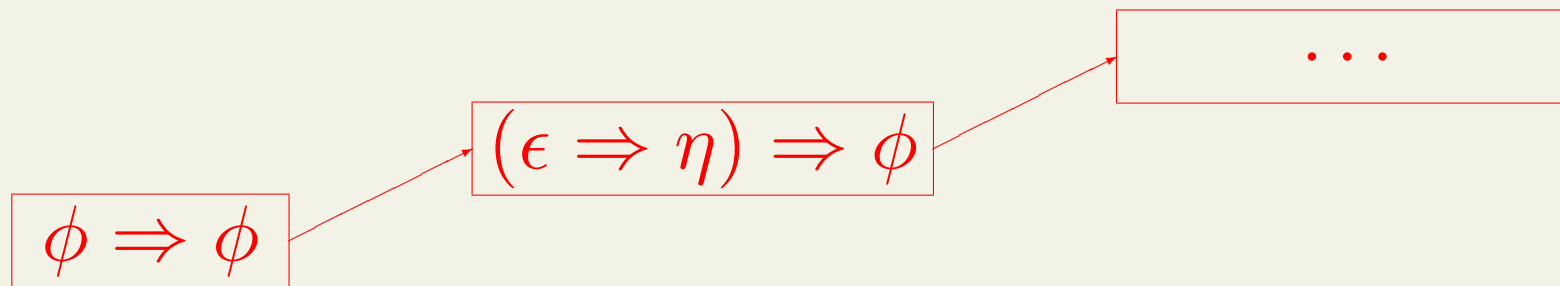


One tactic step (by ...) transforms into proof state $(\epsilon \Rightarrow \eta) \Rightarrow \phi$.

Organizing Proof Search Conceptually

Organize proof search as a **tree of theorems** (thm's).

A sketch of an exemplary proof search:

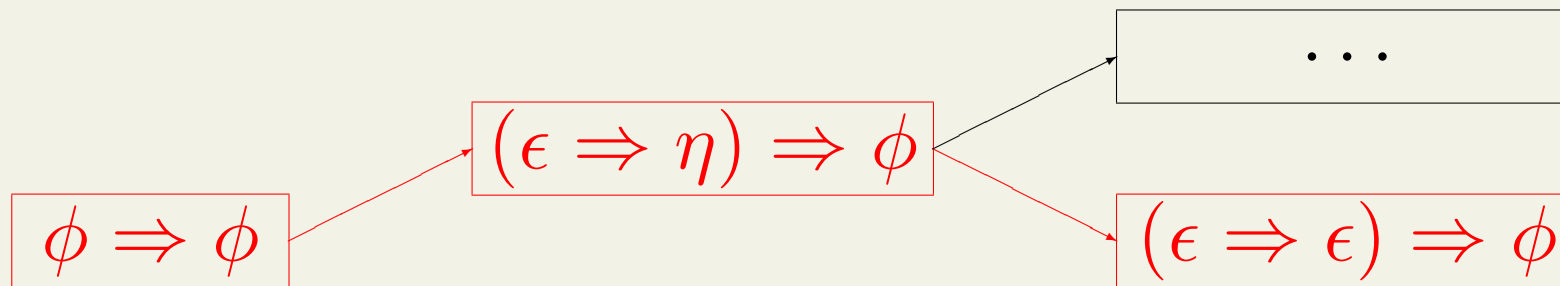


Next tactic step yields dead-end (no tactic application possible).

Organizing Proof Search Conceptually

Organize proof search as a **tree of theorems** (thm's).

A sketch of an exemplary proof search:

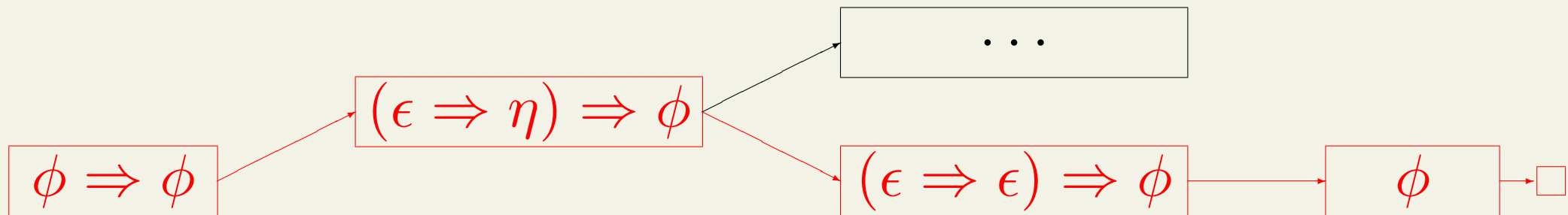


`back()`; tries an alternative successor of $(\epsilon \Rightarrow \eta) \Rightarrow \phi$.

Organizing Proof Search Conceptually

Organize proof search as a **tree of theorems** (thm's).

A sketch of an exemplary proof search:

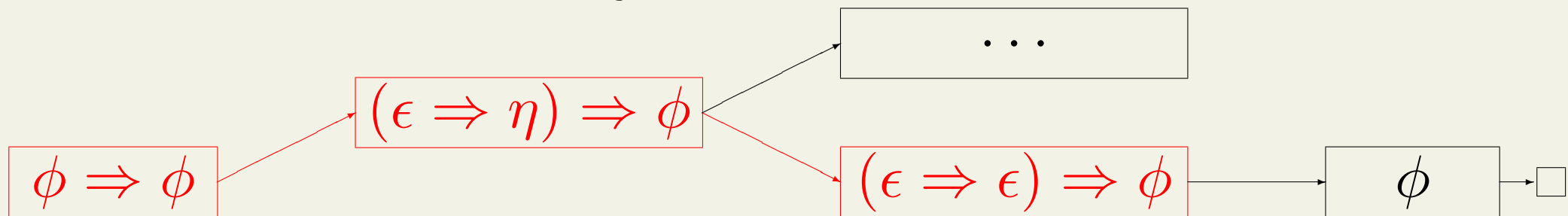


Now $(\epsilon \Rightarrow \epsilon) \Rightarrow \phi$ is solvable using **atac**. **qed**.

Organizing Proof Search Conceptually

Organize proof search as a **tree of theorems** (thm's).

A sketch of an exemplary proof search:

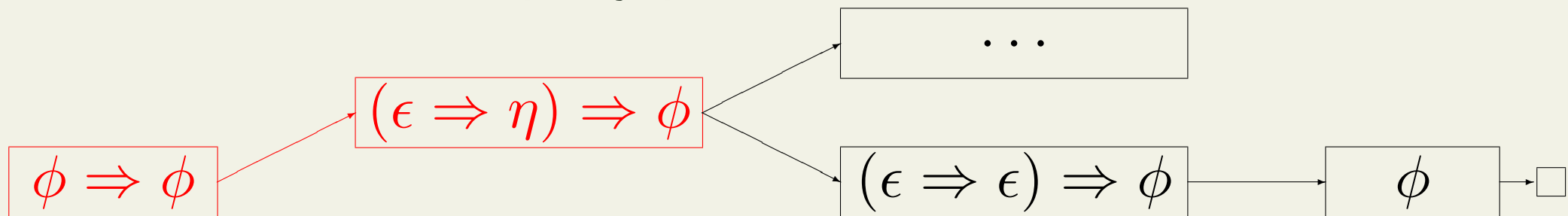


Use `undo()`; three times to go to previous proof states.

Organizing Proof Search Conceptually

Organize proof search as a **tree of theorems** (thm's).

A sketch of an exemplary proof search:

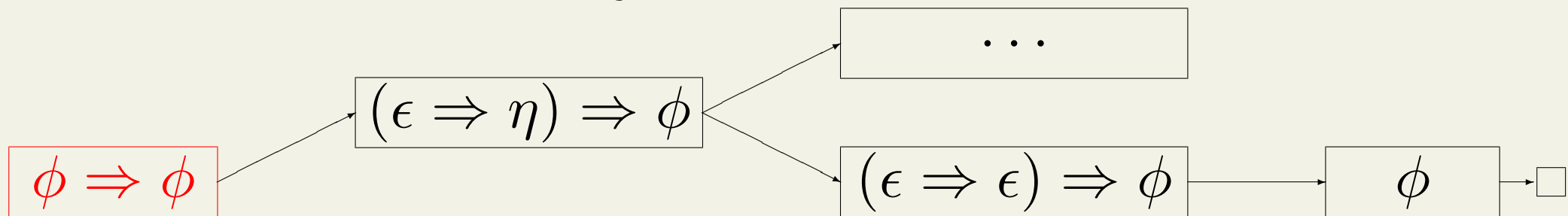


Use `undo()`; three times to go to previous proof states.

Organizing Proof Search Conceptually

Organize proof search as a **tree of theorems** (thm's).

A sketch of an exemplary proof search:

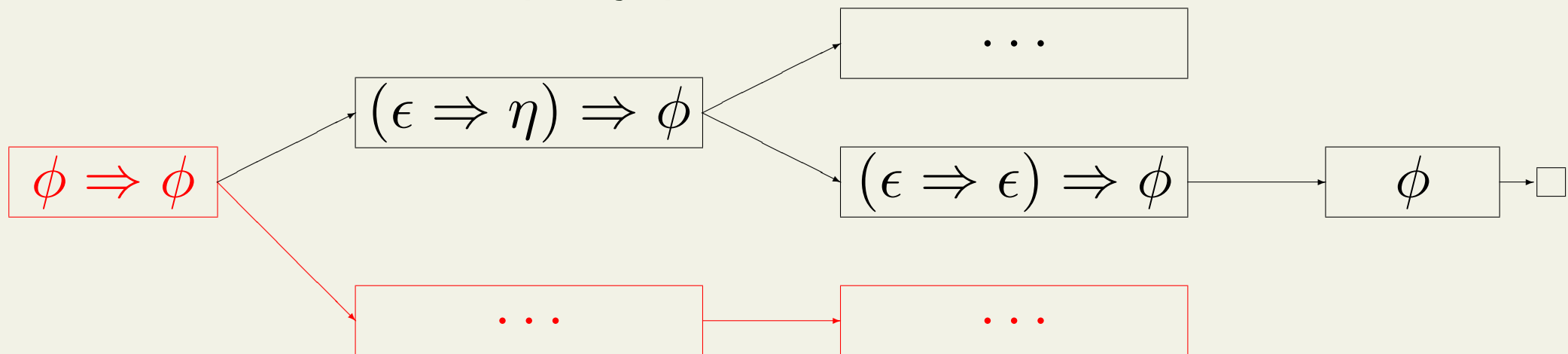


Use `undo()`; three times to go to previous proof states.

Organizing Proof Search Conceptually

Organize proof search as a **tree of theorems** (thm's).

A sketch of an exemplary proof search:

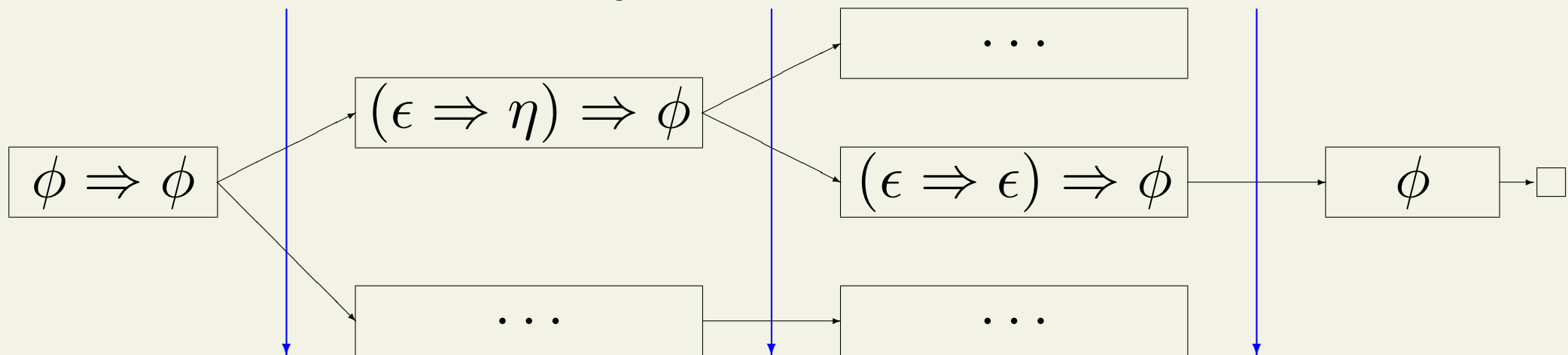


Use `back()`; to try alternative successor. . . .

Organizing Proof Search Conceptually

Organize proof search as a **tree of theorems** (thm's).

A sketch of an exemplary proof search:



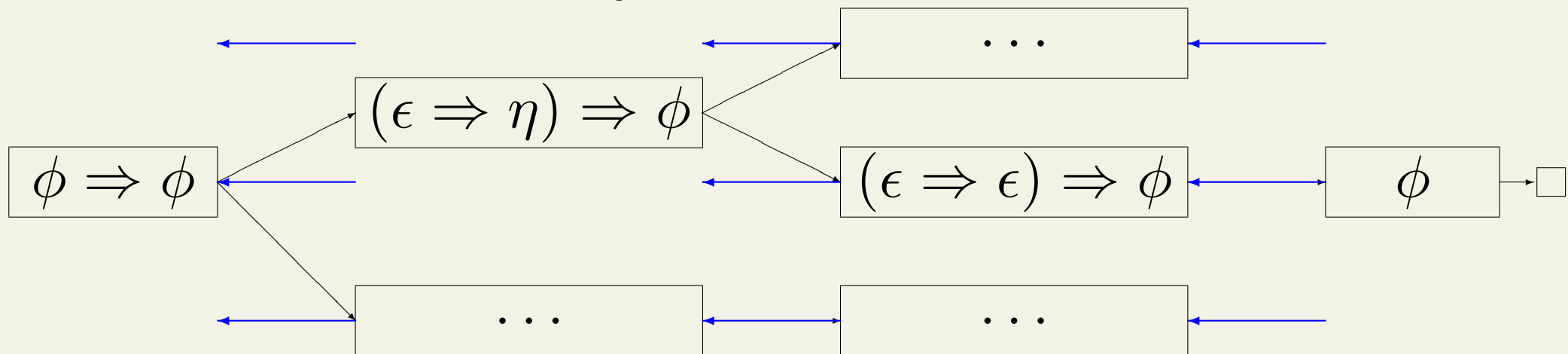
Summary:

back(); to try alternative successors (\Rightarrow different unifiers).

Organizing Proof Search Conceptually

Organize proof search as a **tree of theorems** (thm's).

A sketch of an exemplary proof search:



Summary:

back(); to try **alternative successors** (\Rightarrow different unifiers).

undo(); to go to previous proof state.

Problems

The search space of proof search can be thought of as such a tree, but it cannot be implemented like this straightaway:

- Branching of the tree infinite in general (**HO-unification**).
- **Explicit tree representation** expensive in time and space.

As an **aside**, it is also possible to understand proof search more abstractly. But we are interested in the operational aspects.

Organizing Proof Search Operationally

Organize proof search as a function on theorems (thm's)

$$\text{type tactic} = \text{thm} \rightarrow \text{thm seq}$$

where `seq` is the type constructor for infinite lists.

This allows us to have:

- THEN
- ORELSE
- REPEAT
- INTLEAVE, BREADTHFIRST, DEPTHFIRST, . . .

Classifying Rules

How to Organize Proof Rules?

Observation: Some rules can always be applied **blindly** in backward reasoning, e.g. $\rightarrow-I$ or $\wedge-I$. Others are problematic, e.g. $\wedge-EL$ or $\wedge-ER$ (you do not know which to apply to get rid of a \wedge in the premises).

Classifying Rules

How to Organize Proof Rules?

Observation: Some rules can always be applied **blindly** in backward reasoning, e.g. $\rightarrow-I$ or $\wedge-I$. Others are problematic, e.g. $\wedge-EL$ or $\wedge-ER$ (you do not know which to apply to get rid of a \wedge in the premises).

But proof rules can be tailored to be applied blindly.

In the following we will explain this using **sequent style notation**.

Review: Sequent Notation

$$\Gamma \vdash A \quad (\text{where } A \in \Gamma) \quad \frac{\Gamma \vdash B}{A, \Gamma \vdash B} \text{weaken}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge\text{-}I \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge\text{-}EL \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge\text{-}ER$$

$$\frac{A, \Gamma \vdash B}{\Gamma \vdash A \rightarrow B} \rightarrow\text{-}I \quad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \rightarrow\text{-}E$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee\text{-}IL \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee\text{-}IR$$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \vee\text{-}E$$

Example: $\wedge-E'$

In the **sequent calculus**, one writes $\wedge-E$ as:

$$\frac{A, B, \Gamma \vdash C}{A \wedge B, \Gamma \vdash C} \wedge-E'$$

This mimics the effect of using $\wedge-E$ (`conjE` of Isabelle), here written in **sequent style notation**,

$$\frac{\Gamma \vdash A \wedge B \quad A, B, \Gamma \vdash C}{\Gamma \vdash C} \wedge-E$$

in combination with **etac**. We now derive $\wedge-E'$.

Deriving $\wedge\text{-}E'$ from $\wedge\text{-}E$

$$\frac{\Gamma \vdash A \wedge B \quad A, B, \quad \Gamma \vdash C}{\Gamma \vdash C} \wedge\text{-}E$$

Deriving $\wedge-E'$ from $\wedge-E$

$$\frac{A \wedge B, \Gamma \vdash A \wedge B \quad A, B, A \wedge B, \Gamma \vdash C}{A \wedge B, \Gamma \vdash C} \wedge-E$$

If we replace Γ with $A \wedge B, \Gamma$ (just **instantiation**),

Deriving $\wedge-E'$ from $\wedge-E$

$$\frac{A, B, A \wedge B, \Gamma \vdash C}{A \wedge B, \Gamma \vdash C} \wedge-E$$

If we replace Γ with $A \wedge B, \Gamma$ (just **instantiation**), then one part holds by the **assumption rule**,

Deriving $\wedge-E'$ from $\wedge-E$

$$\frac{\frac{A, B, \Gamma \vdash C}{A, B, A \wedge B, \Gamma \vdash C} \text{weaken}}{A \wedge B, \Gamma \vdash C} \wedge-E$$

If we replace Γ with $A \wedge B, \Gamma$ (just **instantiation**), then one part holds by the **assumption rule**, and we can apply **weakening**.

Deriving $\wedge\text{-}E'$ from $\wedge\text{-}E$

$$\frac{A, B, \Gamma \vdash C}{A \wedge B, \Gamma \vdash C} \wedge\text{-}E'$$

A Proof by Blind Rule Application

$$\vdash (\rho \wedge \phi) \rightarrow \psi \rightarrow \phi$$

The topmost connective is \rightarrow , which asks for \rightarrow -I.

A Proof by Blind Rule Application

$$\frac{\rho \wedge \phi \vdash \psi \rightarrow \phi}{\vdash (\rho \wedge \phi) \rightarrow \psi \rightarrow \phi} \rightarrow -I$$

The topmost connective is \rightarrow , which asks for $\rightarrow -I$.

Again $\rightarrow -I$.

A Proof by Blind Rule Application

$$\frac{\frac{\rho \wedge \phi, \psi \vdash \phi}{\rho \wedge \phi \vdash \psi \rightarrow \phi} \rightarrow -I}{\vdash (\rho \wedge \phi) \rightarrow \psi \rightarrow \phi} \rightarrow -I$$

The topmost connective is \rightarrow , which asks for $\rightarrow -I$.

Again $\rightarrow -I$.

To decompose the assumption $\rho \wedge \phi$, use $\wedge -E'$.

A Proof by Blind Rule Application

$$\begin{array}{c}
 \dfrac{\rho, \phi, \psi \vdash \phi}{\rho \wedge \phi, \psi \vdash \phi} \wedge\text{-}E' \\
 \dfrac{\rho \wedge \phi, \psi \vdash \phi}{\rho \wedge \phi \vdash \psi \rightarrow \phi} \rightarrow\text{-}I \\
 \dfrac{\rho \wedge \phi \vdash \psi \rightarrow \phi}{\vdash (\rho \wedge \phi) \rightarrow \psi \rightarrow \phi} \rightarrow\text{-}I
 \end{array}$$

The topmost connective is \rightarrow , which asks for $\rightarrow\text{-}I$.

Again $\rightarrow\text{-}I$.

To decompose the assumption $\rho \wedge \phi$, use $\wedge\text{-}E'$.

The proof can now be completed by the assumption rule.

Safe and Unsafe Rules

Combined tactics rely on **classification** of rules, maintained in Isabelle data structure `claset`, and accessed by functions of type `claset * thm list → claset`.

Class:	To add use function:
Safe introduction rules	<code>addSIs</code>
Safe elimination rules	<code>addSEs</code>
Unsafe introduction rules	<code>addIs</code>
Unsafe elimination rules	<code>addEs</code>

Adapting Rules for Automated Proof Search

As seen for \wedge - E , rules must be suitably adapted in order to be useful in automated proof search.

$$\frac{}{\vdash (\alpha \rightarrow \beta) \vee (\beta \rightarrow \alpha)} \vee\text{-swap}$$

Adapting Rules for Automated Proof Search

As seen for \wedge -E, rules must be suitably adapted in order to be useful in automated proof search.

$$\frac{\frac{}{\neg(\alpha \rightarrow \beta) \vdash \beta \rightarrow \alpha} \rightarrow\text{-I}}{\vdash (\alpha \rightarrow \beta) \vee (\beta \rightarrow \alpha)} \vee\text{-swap}$$

Neither \vee -IL nor \vee -IR would work here. Uses classical logic.

Adapting Rules for Automated Proof Search

As seen for \wedge - E , rules must be suitably adapted in order to be useful in automated proof search.

$$\begin{array}{c}
 \frac{}{\neg(\alpha \rightarrow \beta), \beta \vdash \alpha} \rightarrow\text{-swap}E \\
 \frac{}{\neg(\alpha \rightarrow \beta) \vdash \beta \rightarrow \alpha} \rightarrow\text{-I} \\
 \hline
 \vdash (\alpha \rightarrow \beta) \vee (\beta \rightarrow \alpha) \quad \vee\text{-swap}
 \end{array}$$

Adapting Rules for Automated Proof Search

As seen for \wedge -E, rules must be suitably adapted in order to be useful in automated proof search.

$$\begin{array}{c}
 \frac{\neg\alpha, \alpha, \beta \vdash \beta}{\neg(\alpha \rightarrow \beta), \beta \vdash \alpha} \rightarrow\text{-swapE} \\
 \frac{\neg(\alpha \rightarrow \beta), \beta \vdash \alpha}{\neg(\alpha \rightarrow \beta) \vdash \beta \rightarrow \alpha} \rightarrow\text{-I} \\
 \frac{\neg(\alpha \rightarrow \beta) \vdash \beta \rightarrow \alpha}{\vdash (\alpha \rightarrow \beta) \vee (\beta \rightarrow \alpha)} \vee\text{-swap}
 \end{array}$$

Principle: Emulate sequent calculus with derived rules.

Handling Quantifiers

Can derive $\forall\text{-}E'$ (\equiv allE) using $\forall\text{-}E$ (\equiv spec):

$$\frac{\forall x.A(x) \quad \begin{array}{c} [A(x) \\ \vdots \\ B \end{array}]}{B} \forall\text{-}E'$$

What is the difference to $\exists\text{-}E$?

Handling Quantifiers

Can derive $\forall\text{-}E'$ (\equiv **allE**) using $\forall\text{-}E$ (\equiv **spec**):

$$\frac{\forall x.A(x) \quad \begin{array}{c} [A(x) \\ \vdots \\ B \end{array}]}{B} \forall\text{-}E'$$

What is the **difference** to $\exists\text{-}E$?

Problem: $\forall x.A(x)$ may still be needed.

Handling Quantifiers

Can derive $\forall\text{-}E'$ (\equiv `allE`) using $\forall\text{-}E$ (\equiv `spec`):

$$\frac{\forall x.A(x) \quad \begin{array}{c} [A(x), \forall x.A(x)] \\ \vdots \\ B \end{array}}{B} \forall\text{-dup}E$$

What is the **difference** to $\exists\text{-}E$?

Problem: $\forall x.A(x)$ may still be needed.

Principle: Introduce **duplicating** rules. Turns search **infinite**!

Check out `allE` and `all_dupE` in `IFOL_lemmas.ML`!

Proof Procedures (Simplified)

Tactics in **Isabelle** are performed **in order**:

1. REPEAT (*rtac safe_I_rules* OTHERWISE *etac safe_E_rules*)
2. canonize: propagate " $x = t$ " throughout subgoal
3. *rtac unsafe_I_rules* OTHERWISE *etac unsafe_E_rules*
4. *atac*

Combined Proof Search Tactics

- `step_tac : claset → int → tactic`
(just safe steps)
- `fast_tac : claset → int → tactic`
(safe and unsafe steps in depth-first strategy)
- `best_tac : claset → int → tactic`
(safe and unsafe steps in breadth-first strategy)
- `slow_tac : claset → int → tactic`
(like `fast_tac`, but with backtracking `atac`'s)
- `blast_tac : claset → int → tactic`
(like `fast_tac`, but often more powerful)

Summary on Automated Proof Search

- Proof search can be organized as a **tree of theorems**.
- Calculi can be set up to facilitate proof search (although this must be done by specialists).
- Combined with **search strategies**, powerful automatic procedures arise. Can prove well-known hard problems such as $((\exists y. \forall x. J(y, x) \vee \neg J(x, x)) \rightarrow \neg(\forall x. \exists y. \forall z. J(z, y) \vee \neg J(z, x)))$
- Unfortunately, failure is **difficult to interpret**. ►

More Detailed Explanations

Sequent Calculus/Tableau

Tableau proving is a derivation system [Fit96].

It turns out that the language of tableaux is equivalent to the **sequent calculus** (recall our use of **sequent style notation**) [Pau97]. The techniques Isabelle uses for automating proofs can thereby be understood as tableau proving [Pau97].

Need for Automation

We have seen in the exercises that doing a proof step by step is very tedious and often involves difficult guessing or alternatively, backtracking. We cannot hope to prove anything about realistic systems if proving simple theorems is so tedious.

Efficiency considerations are important for automation. The non-determinacy in proof search obviously leads to inefficiencies as many possibilities have to be explored.

Which Subgoal?

We have seen in the exercises (and also in [the lecture](#)) that one can choose the subgoal to which one wants to apply a rule.

Technical Organiztion

We have seen in the [previous lecture](#) that resolution transforms a proof state into a new proof state. But how does one organize all those potential proof states in order to find [proofs](#)?

A Tree of Theorems

We have seen in the [previous lecture](#) that resolution transforms a proof state into a new proof state. Since in general, a proof state has several **successor** states (states that can be obtained by one resolution step), conceptually one obtains a **tree** where the children of a state are the successors.

$$\phi \Longrightarrow \phi?$$

The initial proof state is $\phi \Longrightarrow \phi$. Isabelle will display this as

Level 1 : (1 subgoal)

ϕ

1. ϕ

It might contradict your intuition and experience with Isabelle to think that the initial proof state is $\phi \Longrightarrow \phi$. Shouldn't it be just ϕ ? However, this seeming contradiction can be resolved.

The way Isabelle displays the proof state focuses on **what has to be proven**, the subgoals. The proof state should be read as: if I have proven ϕ (the ϕ occurring after the 1.), I am done.

Technically, the proof state is an **Isabelle theorem** (`thm`), i.e. something which Isabelle regards as true. Now of course, she cannot initially regard

ϕ as true, as ϕ is what is to be proven. But she can regard $\phi \implies \phi$ as true. The aim of a proof search is to transform $\phi \implies \phi$ (ϕ can be shown if I assume ϕ) into ϕ (ϕ can be shown if I assume nothing).

Isabelle

For more details on Isabelle technicalities, you should consult the reference manual [[Pau04](#)].

Alternative Successors

Note that when there are no more successors (you cannot go down) anymore, `back();` will go to the previous proof state, i.e., go to the left one level (just like `undo();`), and **then** try alternative successors.

Isabelle Theorems

Technically, a proof state is an **Isabelle theorem**, (thm), i.e. something which **Isabelle** regards as true.

Explicit Tree Representation Expensive

Obviously, an infinite tree cannot be represented explicitly. But even if the tree is finite, it is generally expensive to represent it explicitly. In particular, the tree may contain many failing branches and only few successful ones, which begs the question if representing the unsuccessful branches cannot be avoided somehow.

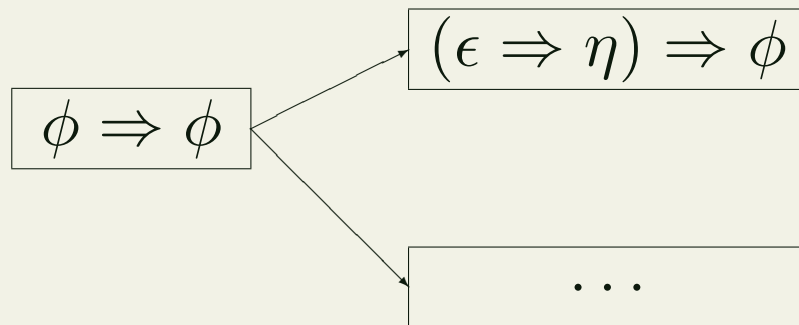
Explicit Tree Representation not Abstract

The explicit tree representation is not very abstract in that each node has a defined order of the children (first successor, second successor, . . .). This order is an artefact of the order in which **unifiers are enumerated** by the unification algorithm used. It is inessential for the proofs that are contained in the tree.

As a more abstract understanding of proof search, one can organize proof search as a **relation on theorems** (thm's)

$$prooftrees = \mathcal{P}(\text{thm} \times \text{thm})$$

More precisely, one can look at a fragment of a tree of theorems as **before**, e.g.:



One could say that each tactic application (with a particular rule) gives rise to a relations on theorems. That is to say, ϕ and ϕ' are in the relation if ϕ' is a successor proof state of ϕ .

This is **abstract** in that there is no **order** among the successors of a proof state.

Also, one does not represent a tree explicitly.

Advantage: we have an abstract algebra.

- $PT_1 \circ PT_2$: sequential composition (“then”).

Given two relations between **thm's**, PT_1 and PT_2 , we define

composition $PT_1 \circ PT_2$ as the relation

$$\{(\phi, \psi) \mid \text{there is } \eta \text{ such that } (\phi, \eta) \in PT_1 \text{ and } (\eta, \psi) \in PT_2\}$$

- $PT_1 \cup PT_2$: alternative of proof attempts (“or”)

The **union** of two relations is defined as usual for sets. If PT_1 and PT_2 each model the application of a particular tactic, then $PT_1 \cup PT_2$ models the application of “first tactic **or** second tactic”.

- PT^* : reflexive transitive closure (“repeat ”)

- $(\phi \Rightarrow \phi, \phi) \in PT^* \quad \equiv \quad \text{“there is a proof for } \phi\text{”}$

PT^* is inductively defined as the smallest set where

- $(\phi, \phi) \in PT^*$ for all ϕ ;
- if $(\phi, \eta) \in PT$ and $(\eta, \psi) \in PT^*$ then $(\phi, \psi) \in PT^*$.

So if PT models the application of a particular tactic, then PT^*

models the application of that tactic arbitrarily many times.

However, this also has some disadvantages:

- Union \cup is difficult to implement (needs comparison with all previous results since one wants to avoid duplicates).
- More **operational**, strategic interpretations of union \cup are desirable (try this — then that, interleave attempts in PT_1 with attempts in PT_2 , and so forth).

A Function on Theorems

This way of understanding and organizing proof search is not so **abstract**, but rather operational. Instead of saying that ϕ and ϕ' are in a relation, one says that ϕ' is in the sequence returned by the tactic applied to ϕ . There is an **order** among the successors of a proof state. One still does not represent a tree explicitly, although conceptually, proof search is about exploring **this tree**.

Infinite Lists

For any type τ , the type τ seq (recall the [notation](#)) is the type of (possibly) infinite lists of elements of type τ . This is of course an abstract datatype. There should be functions to return the head and the tail of such an infinite list.

An **abstract datatype** is a type whose terms cannot be represented explicitly and accessed directly, but only via certain functions for that type.

Tacticals

- THEN
- ORELSE
- REPEAT
- INTLEAVE, BREADTHFIRST, DEPTHFIRST, . . .

are called **tacticals**.

Tacticals are operations on tactics. They play an important role in automating proofs in [Isabelle](#). The most basic tacticals are THEN and ORELSE. Both of those tacticals are of type `tactic * tactic → tactic` and are written infix: tac_1 THEN tac_2 applies tac_1 and then tac_2 , while tac_1 ORELSE tac_2 applies tac_1 if possible and otherwise applies tac_2 [[Pau04](#), Ch. 4].

$\wedge\text{-}E$

In `Isabelle` notation, it looks as follows:

$$\llbracket P \& Q; \llbracket P; Q \rrbracket \Longrightarrow R \rrbracket \Longrightarrow R$$

(see `IFOL_lemmas.ML`).

Blind Application of $\wedge-E'$

See now that we first derived the rule $\wedge-E'$, which is a rule that can be used blindly to decompose a conjunction in the assumptions. This was not something ad-hoc to prove this particular formula. The rule $\wedge-E'$ should be used generally instead of $\wedge-EL$ or $\wedge-EL$, because it has the advantage that it can be applied blindly.

The essential point about being able to apply a rule blindly is that the application **does not throw any information away**. This is indeed the case for $\wedge-E'$. We remove the assumption $\phi \wedge \psi$, but we get the two conjuncts ϕ and ψ as assumptions instead.

The rule $\wedge-E'$ mimics the effect of using $\wedge-E$ in combination with **etac**, which you best see by trying such a proof in Isabelle.

claset

`claset` is an abstract datatype. Overloading notation, `claset` is also an ML unit function which will return a term of that datatype when applied to `()`, namely, the current classifier set.

A classifier set determines which rules are safe and unsafe introduction, respectively elimination rules. The current classifier set is a classifier set used by default in certain tactics.

The current classifier set can be accessed via special functions for that purpose.

Accessing the claset

The functions `addSIs`, `addSEs`, `addIs`, `addEs` are all of type `claset * thm list → claset`. They add rules to the current classifier set. For example, `addSIs` adds a rule as **safe introduction rule**.

Emulating the Sequent Calculus

The sequent calculus works with expressions of the form

$A_1, \dots, A_n \vdash B_1, \dots, B_m$ which should be interpreted as: under the assumptions A_1, \dots, A_n , at least one of B_1, \dots, B_m can be proven. So as a formula, this would be $A_1 \wedge \dots \wedge A_n \rightarrow B_1 \vee \dots \vee B_m$.

In Isabelle (and the proof trees we have seen, e.g., in this lecture), we only have sequents with one formula to the right of the \vdash . We have said that we use **sequent notation**.

The important point to note here is that in the sequent calculus, one can shift a formula from left to right or vice versa, but one has to **negate** it, or more precisely, turn A into $\neg A$ and $\neg A$ into A . This is called **swapping** and is an important technique for **combined tactics**.

The sequent calculus inherently relies on **classical** reasoning [Pau04, Ch. 11].

Deriving allE

You should do it in Isabelle. The rule is:

$$\llbracket \text{ALL } x. P(x); P(x) \Longrightarrow R \rrbracket \Longrightarrow R$$

Name Confusion

As you may have noticed earlier, there is a confusion between the names of proof rules as we present them for the theory and the names used in Isabelle. For example, rule \rightarrow -E is called `mp` in Isabelle. This confusion concerns elimination rules.

There is however a good reason for these choices. In traditional presentations of logic, one sets up the simplest possible elimination rules for the connectives which naturally arise from the meaning of those connectives. This is what we have done as well. However, as we see in this lecture, these rules cannot be applied blindly and are thus not very suitable for automation. Therefore, **combined tactics** in Isabelle use derived rules such as \wedge -E (called `conjE` in Isabelle).

Since this is of such central importance for Isabelle, one prefers to have the obvious names `conjE`, `allE` etc. for the rules that are actually used

in “advanced” applications of Isabelle.

$\exists-E$

The rule

$$\frac{\exists x.A(x) \quad \begin{array}{c} [A(x)] \\ \vdots \\ B \end{array}}{B} \exists-E$$

was derived [previously](#) (but in Isabelle, it is a basic rule in [IFOL.ML](#)). It is

$$\llbracket \text{ALL } x. P(x); !!x. P(x) \Longrightarrow R \rrbracket \Longrightarrow R$$

Note that the rule [allE](#) ($\forall-E'$) is

$$\llbracket \text{ALL } x. P(x); P(x) \Longrightarrow R \rrbracket \Longrightarrow R$$

The difference is that the former rule contains a metalevel universal quantifier. In terms of paper-and-pencil proofs, $\exists-E$ has the side

condition that x must not occur free in any assumption on which B (see tree!) depends. There is no such side condition for \forall - E' .

Difference between $\forall-E'$ and $\exists-E$

The difference between

$$\frac{\frac{\exists x.A(x)}{B} \quad \begin{array}{c} [A(x)] \\ \vdots \\ B \end{array}}{B} \exists-E$$

and

$$\frac{\frac{\forall x.A(x)}{B} \quad \begin{array}{c} [A(x)] \\ \vdots \\ B \end{array}}{B} \forall-E'$$

is that the first rule has a side condition: x must not occur free in any assumption on which B depends. See also what this means in terms of Isabelle.

The Rule \vee -swap

The rule \vee -swap is

$$\frac{\neg A, \Gamma \vdash B}{\Gamma \vdash A \vee B} \vee\text{-swap}$$

To derive it you need **classical** reasoning, as the rule exploits the equivalence of $A \rightarrow B$ and $\neg A \vee B$.

This is a derived rule which is explicitly contained in the Isabelle classifier set as the **classical introduction rule for \vee** . It is called `disjCI` (check out `FOL_lemmas1.ML`)!

The Rule \rightarrow -swapE

The rule \rightarrow -swapE is

$$\frac{A, \neg C, \Gamma \vdash B}{\neg(A \rightarrow B), \Gamma \vdash C} \rightarrow\text{-swapE}$$

To derive it you need **classical** reasoning, as the rule exploits the equivalence of $\neg(A \rightarrow B)$ and $A \wedge \neg B$.

This is a standard technique in Isabelle, based on **swapping**. For dealing with negated formulas in the premises of the current subgoal, **introduction** rules are combined with swap using etac.

Generally, we have a formula $\neg(A \circ B)$ in the premises, where \circ is some binary connective. Swapping will put $(A \circ B)$ in the conclusion and put the old conclusion into the premises after negating it. Afterwards, an introduction rule for \circ will be used [Pau04, Section 11.2].

Duplicating Rules

You should recall that elimination rules are used in combination with *etac*. Using *allE* will eliminate the quantifier.

You should try a proof of the formula $(\forall x.P(x)) \rightarrow (P(a) \wedge P(b))$ in Isabelle to convince yourself that this is a problem since the quantified formula $\forall x.P(x)$ is needed twice as an assumption, with two different instantiations of x .

The duplicating rule \forall -*dupE* has the effect that the universally quantified formula will still remain as an assumption.

Infinite Proof Search

Given only the rules so far (in combination with the appropriate tactics, `rtac` and `etac`, and `swapping`), excluding $\forall\text{-dup}E$, the proof search would be finite.

The rule $\forall\text{-dup}E$ is responsible for making the proof search infinite. This can be no surprise however, as first-order logic is undecidable [And02], and so there can be no automatic procedure for proving all true first-order formulas.

Isabelle Files

These files should be contained in your [Isabelle](#) distribution. In Freiburg they are located at:

```
/usr/local/isabelle/Isabelle/src/FOL/
```


Proof Procedures

Tactics in **Isabelle** are performed **in order**:

1. REPEAT (**rtac** *safe_I_rules* ORELSE **etac** *safe_E_rules*);
2. canonize: propagate " $x = t$ " . . . throughout subgoal;
3. **rtac** *unsafe_I_rules* ORELSE **etac** *unsafe_E_rules*;
4. **atac**.

One elementary proof step consists of trying a safe introduction rule with **rtac**, or, if that is not possible, a safe elimination rule with **etac**. This will be repeated as long as possible.

Then in the current subgoal, any assumption of the form $x = t$ (where x is a metavariable) will be propagated throughout the subgoal, i.e., all occurrences of x will be replaced by t .

Then Isabelle will try **one** application of an unsafe introduction rule with

rtac, or, if that is not possible, an unsafe elimination rule with **etac**. Finally, she will use **atac**. Note that **atac** is unsafe. In general, there are several premises in a subgoal and **atac** may unify the conclusion of the subgoal with the wrong premise.

Failure in Classical Reasoning

`fast_tac`, `blast_tac` just tell you that the tactic failed, but not why. And it would be difficult to do that, since backtracking means that all attempts failed. This can have several reasons: a rule is missing, a rule has been **classified** wrongly, the **search strategy** was not adequate for the problem, enumeration of **unifiers** in a bad order. Or a combination thereof. Or it might be that **too many unsafe steps** are needed, since `fast_tac` limits their number.

Term Rewriting

Outline of this Part

- Higher-order rewriting
- Extensions: Ordered, pattern, congruence, splitting rewriting
- Organizing simplification rules

In this context, a term is a λ -term, since we use the λ -calculus to encode object logics.

Higher-Order Rewriting

Motivation:

- Simplification is a very important part of deduction, e.g.:

$$0 + (x + 0) = x$$

$$[a, b, d] @ [a, b] = [a, b, d, a, b]$$

- Based on **rewrite rules** as in functional programming:

$$x + 0 = x,$$

$$0 + x = x$$

$$[] @ X = X,$$

$$(x :: X) @ Y = x :: (X @ Y)$$

Term Rewriting: Foundation

- Recall: An **equational theory** consists of **rules**

$$\begin{array}{c}
 \frac{}{x = x} \textit{refl} \quad \frac{x = y}{y = x} \textit{sym} \quad \frac{x = y \quad y = z}{x = z} \textit{trans} \\
 \\
 \frac{x = y \quad P(x)}{P(y)} \textit{subst}
 \end{array}$$

- plus additional (possibly conditional) rules of the form

$$\phi_1 = \psi_1, \dots, \phi_n = \psi_n \Rightarrow \phi = \psi.$$

The additional rules can be interpreted as **rewrite rules**, i.e. they are applied from **left to right**.

Incomplete Decision Procedure for Goal $e = e'$

To decide if $e = e'$ in an equational theory:

1. **stop** if the goal is **solved**, i.e., $e \equiv e'$ (syntactical equality)

Incomplete Decision Procedure for Goal $e = e'$

To decide if $e = e'$ in an equational theory:

1. **stop** if the goal is **solved**, i.e., $e \equiv e'$ (syntactical equality)
2. make a rewrite step:
 - (a) pick a subterm t in $e(t)$

Incomplete Decision Procedure for Goal $e = e'$

To decide if $e = e'$ in an equational theory:

1. **stop** if the goal is **solved**, i.e., $e \equiv e'$ (syntactical equality)
2. make a rewrite step:
 - (a) pick a subterm t in $e(t)$
 - (b) for a rewrite rule $\phi = \psi$, **match** (unify)
 ϕ against t , i.e., find θ such that $\phi\theta = t$

Incomplete Decision Procedure for Goal $e = e'$

To decide if $e = e'$ in an equational theory:

1. **stop** if the goal is **solved**, i.e., $e \equiv e'$ (syntactical equality)
2. make a rewrite step:
 - (a) pick a subterm t in $e(t)$
 - (b) for a rewrite rule $\phi = \psi$, **match** (unify)
 ϕ against t , i.e., find θ such that $\phi\theta = t$
 - (d) replace $e(t)$ by $e(\psi\theta)$

Incomplete Decision Procedure for Goal $e = e'$

To decide if $e = e'$ in an equational theory:

1. **stop** if the goal is **solved**, i.e., $e \equiv e'$ (syntactical equality)
2. make a rewrite step:
 - (a) pick a subterm t in $e(t)$
 - (b) for a rewrite rule $\phi = \psi$, **match** (unify)
 ϕ against t , i.e., find θ such that $\phi\theta = t$
 - (d) replace $e(t)$ by $e(\psi\theta)$
3. **goto** 1

Incomplete Decision Procedure for Goal $e = e'$

To decide if $e = e'$ in an equational theory:

1. **stop** if the goal is **solved**, i.e., $e \equiv e'$ (syntactical equality)
2. make a rewrite step:
 - (a) pick a subterm t in $e(t)$ (resp. $e'(t)$)
 - (b) for a rewrite rule $\phi = \psi$, **match** (unify)
 ϕ against t , i.e., find θ such that $\phi\theta = t$
 - (d) replace $e(t)$ by $e(\psi\theta)$ (resp. $e'(t)$ by $e'(\psi\theta)$)
3. **goto** 1

Incomplete Decision Procedure for Goal $e = e'$

To decide if $e = e'$ in an equational theory:

1. **stop** if the goal is **solved**, i.e., $e \equiv e'$ (syntactical equality)
2. make a rewrite step:
 - (a) pick a subterm t in $e(t)$ (resp. $e'(t)$)
 - (b) for a rewrite rule $\phi_1 = \psi_1, \dots, \phi_n = \psi_n \implies \phi = \psi$, **match** (unify) ϕ against t , i.e., find θ such that $\phi\theta = t$
 - (c) **solve** $(\phi_1 = \psi_1, \dots, \phi_n = \psi_n)\theta$
 - (d) replace $e(t)$ by $e(\psi\theta)$ (resp. $e'(t)$ by $e'(\psi\theta)$)
3. **goto** 1

Incomplete Decision Procedure for Goal $e = e'$

To decide if $e = e'$ in an equational theory:

1. **stop** if the goal is **solved**, i.e., $e \equiv e'$ (syntactical equality)
2. make a rewrite step:
 - (a) pick a subterm t in $e(t)$ (resp. $e'(t)$)
 - (b) for a rewrite rule $\phi_1 = \psi_1, \dots, \phi_n = \psi_n \implies \phi = \psi$, **match** (unify) ϕ against t , i.e., find θ such that $\phi\theta = t$
 - (c) **solve** $(\phi_1 = \psi_1, \dots, \phi_n = \psi_n)\theta$
 - (d) replace $e(t)$ by $e(\psi\theta)$ (resp. $e'(t)$ by $e'(\psi\theta)$)
3. **goto** 1

This procedure + the rules define a **term rewriting system**.

Term Rewriting is Non-Trivial

- There are two major problems: this decision procedure may fail because:
 - it diverges (the rules are **not terminating**), e.g. $x + y = y + x$ or $x = y \implies x = y$;

Term Rewriting is Non-Trivial

- There are two major problems: this decision procedure may fail because:
 - it diverges (the rules are **not terminating**), e.g. $x + y = y + x$ or $x = y \implies x = y$;
 - rewriting does not yield a unique normal form (the rules are not **confluent**), e.g. rules $a = b$, $a = c$.

Term Rewriting is Non-Trivial

- There are two major problems: this decision procedure may fail because:
 - it diverges (the rules are **not terminating**), e.g. $x + y = y + x$ or $x = y \implies x = y$;
 - rewriting does not yield a unique normal form (the rules are not **confluent**), e.g. rules $a = b$, $a = c$.
- Providing criteria for terminating and confluent rule sets is an active research area (see [BN98, KIo93], RTA, . . .).

Extensions of Rewriting

- Symmetric rules are problematic, e.g. ACI:

$$(x + y) + z = x + (y + z) \quad (A)$$

$$x + y = y + x \quad (C)$$

$$x + x = x \quad (I)$$

Extensions of Rewriting

- Symmetric rules are problematic, e.g. ACI:

$$(x + y) + z = x + (y + z) \quad (\text{A})$$

$$x + y = y + x \quad (\text{C})$$

$$x + x = x \quad (\text{I})$$

- Idea: apply only if replaced term gets smaller w.r.t. some term ordering. In example, if $(y + x)\theta$ is smaller than $(x + y)\theta$.
- **Ordered rewriting** solves rewriting modulo ACI, using derived rules (exercise).

Extension: HO-Pattern Rewriting

Rules such as $F(G\ c) = \dots$ lead to highly ambiguous matching and hence inefficiency.

Solution: restrict l.h.s. of a rule to higher-order patterns.

Extension: HO-Pattern Rewriting

Rules such as $F(G\ c) = \dots$ lead to highly ambiguous matching and hence inefficiency.

Solution: restrict l.h.s. of a rule to higher-order patterns.

A term t is a HO-pattern if

- it is in β -normal form; and
- any free F in t occurs in a subterm $F\ x_1 \dots x_n$ where the x_i are η -equivalent to distinct bound variables.

Extension: HO-Pattern Rewriting

Rules such as $F(G\ c) = \dots$ lead to highly ambiguous matching and hence inefficiency.

Solution: restrict l.h.s. of a rule to **higher-order patterns**.

A term t is a **HO-pattern** if

- it is in β -normal form; and
- any **free** F in t occurs in a subterm $F\ x_1 \dots x_n$ where the x_i are η -equivalent to distinct bound variables.

Matching (unification) is decidable, **unitary** ('unique') and efficient algorithms exist.

HO-Pattern Rewriting (Cont.)

A rule $\dots \Rightarrow \phi = \psi$ is a **HO-pattern rule** if:

- the left-hand side ϕ is a HO-pattern;
- all **free** variables in ψ occur also in ϕ ; and
- ϕ is **constant-head**, i.e. of the form $\lambda x_1..x_m.c p_1 \dots p_n$ (where c is a **constant**, $m \geq 0$, $n \geq 0$).

HO-Pattern Rewriting (Cont.)

A rule $\dots \Rightarrow \phi = \psi$ is a **HO-pattern rule** if:

- the left-hand side ϕ is a HO-pattern;
- all **free** variables in ψ occur also in ϕ ; and
- ϕ is **constant-head**, i.e. of the form $\lambda x_1..x_m.c p_1 \dots p_n$ (where c is a **constant**, $m \geq 0$, $n \geq 0$).

Example: $(\forall x.Px \wedge Qx) = (\forall x.Px) \wedge (\forall x.Qx)$

Result: HO-pattern allows for very effective quantifier reasoning.

Extensions Related to if – then – else

The if-then-else construct will play an important role later. It asks for special rewrite rules.

Extension: Congruence Rewriting

Problem :

$\text{if } A \text{ then } P \text{ else } Q = \text{if } A \text{ then } P' \text{ else } Q$
where $P = P'$ under condition A

is not a rule.

Solution in Isabelle: explicitly admit this extra class of rules
(congruence rewriting)

$\llbracket A \Longrightarrow P = P' \rrbracket \Longrightarrow$
 $\text{if } A \text{ then } P \text{ else } Q = \text{if } A \text{ then } P' \text{ else } Q$

Extension: Splitting Rewriting

Problem:

$$P(\text{if } A \text{ then } x \text{ else } y) = \text{if } A \text{ then } (P x) \text{ else } (P y)$$

is not a HO-pattern rule (since it is not **constant-head**).

Solution in **Isabelle**: explicitly admit this extra class of rules (**case splitting**).

Organizing Simplification Rules

- Standard (HO-pattern conditional ordered rewrite) rules;
- congruence rules;
- splitting rules.

Isabelle data structure: `simpset`. Some operations:

- `addsimps : simpset * thm list → simpset`
- `delsimps : simpset * thm list → simpset`
- `addcongs : simpset * thm list → simpset`
- `addsplits : simpset * thm list → simpset`

Commutativity can be added without losing termination.

How to Apply the Simplifier?

Several **versions** of the simplifier:

- `simp_tac : simpset → int → tactic`
- `asm_simp_tac : simpset → int → tactic`
(includes assumptions into simpset)
- `asm_full_simp_tac : simpset → int → tactic`
(rewrites assumptions, and includes them into simpset)

Using **global** simplifier sets: `Simp_tac`, `Asm_simp_tac`, `Asm_full_simp_tac`.

Summary on Term Rewriting

Simplifier is a powerful proof tool for

- conditional equational formulas
- ACI-rewriting
- quantifier reasoning
- congruence rewriting
- automatic proofs by case splitting

Fortunately, failure is quite easy to interpret.

Summary on Last Three Sections

- Although Isabelle is an **interactive** proof construction, it is a flexible environment with powerful **automated** proof procedures.
- For **classical** logic and set theory, **tableau-like** procedures like `blast_tac` and `fast_tac` decide many tautologies.
- For equational theories (datatypes, evaluating functional programs, but also **higher-order logic**) `simp_tac` decides many tautologies (and is fairly easy to control). ►

More Detailed Explanations

$$0 + (x + 0) = x$$

Simplifying $0 + (x + 0)$ to x is something you have learned in school. It is justified by the usual semantics of arithmetic expressions. Here, however, we want to see more formally how such simplification works, rather than why it is justified.

Lists

Lists are a common datatype in functional programming. $[a, b, d, a, b]$ is a list. Actually, this notation is **syntactic sugar** for $a :: (b :: (d :: (a :: (b :: []))))$. Here, $[]$ is the empty list and $::$ is a term constructor taking an element and a list and returning a list. $@$ stands for list concatenation.

Intuitively, it is clear that $[a, b, d]$ concatenated with $[a, b]$ yields $[a, b, d, a, b]$.

Term constructor is usual terminology in functional programming. In first-order logic, we would speak of a **function symbol**. In the λ -calculus, we would speak of a (special kind of) constant (this will become clear later).

Functional Programming

For example, the lines

$$\begin{aligned} [] @ X &= X \\ (x :: X) @ Y &= x :: (X @ Y) \end{aligned}$$

define the list concatenation function @.

Rewrite Rules

An equational theory is a formalism based on equational rules of the form $\phi_1 = \psi_1, \dots, \phi_n = \psi_n \implies \phi = \psi$.

A term rewriting system (to be defined shortly) is another formalism, based of **rewrite rules**. They also have the form

$\phi_1 = \psi_1, \dots, \phi_n = \psi_n \implies \phi = \psi$, but they have a different flavor in that $=$ must be interpreted as a directed symbol. One could also write \rightsquigarrow instead of $=$ to emphasize this.

Matching

Given two terms s and t , a **unifier** is a substitution θ such that $s\theta = t\theta$.
A **match** is a substitution which only instantiates one of s or t , so $s\theta = t$ or $s = t\theta$ (one should usually clarify in the given context which of the terms is instantiated).

Solve $(\phi_1 = \psi_1, \dots, \phi_n = \psi_n)\theta$

This means that the procedure is called recursively for the conditions of the rewrite rule.

Term Rewriting System

The procedure defines a **term rewriting system** [BN98, KIo93].

Equational theories, term rewriting systems, propositional logic, first-order logic, different versions of the λ -calculus — with all those different formalisms playing a role here, we must agree on some terminology. In particular, the words **term**, **function**, **predicate**, **constant** and **variable** are used somewhat differently in the different formalisms. Our point of reference for the terminology is the λ -calculus as it is **built into Isabelle** for representing object logics. In particular:

- A **term** is a λ -term; object-level formulae (including equations) as well as object-level terms are all represented as λ -terms, and so for example, when we rewrite an **equation**, we rewrite a term.
- One could say that a **function** is any λ -term of functional type, i.e., of type containing at least one \rightarrow . Apart from that, there may be

function symbols in some object logic. On the metalevel (and hence also for the purpose of term rewriting), these would be **constants**.

- There may be **predicate symbols** in some object logic. On the metalevel (and hence also for the purpose of term rewriting), these would be **constants**.
- A **constant** is a λ -term consisting of just one symbol from a set *Const*. **Constants** of the λ -calculus may be used to represent connectives, **quantifiers**, functions, predicates or any other symbols that an object logic may contain.
- The notion of variable is that of the metalevel, and so we usually mean “variables including metavariables”.

Nevertheless, some confusion may arise wherever we use the terminology from the point of view of an object logic.

See the following example:

The following is an example rewrite sequence, using the [rules](#) for [lists](#). The picked subterm which is being replaced is underlined in each step:

$$\begin{aligned}
 & \underline{(a :: (b :: (d :: [])))} @ (a :: (b :: [])) &= [a, b, d, a, b] &\rightsquigarrow \\
 & a :: \underline{((b :: (d :: [])) @ (a :: (b :: [])))} &= [a, b, d, a, b] &\rightsquigarrow \\
 & a :: (b :: \underline{((d :: []) @ (a :: (b :: [])))}) &= [a, b, d, a, b] &\rightsquigarrow \\
 & a :: (b :: (d :: \underline{([] @ (a :: (b :: [])))})) &= [a, b, d, a, b] &\rightsquigarrow \\
 & a :: (b :: (d :: (a :: (b :: [])))) &= [a, b, d, a, b] &\rightsquigarrow
 \end{aligned}$$

Note that we are done now, as the right-hand side is identical to the left-hand side, modulo the use of [syntactic sugar](#).

Note that generally, a term rewriting sequence rewrites arbitrary terms. Here we only rewrite equations. From the point of view of term rewriting, an equation is just a [special case](#) of a term.

One could also imagine that object-level function and predicate symbols

are represented as variables, as is done in LF. Recall Perlis' epigram.

$$a = b, a = c$$

For a rewriting system consisting of rules $a = b$, $a = c$, one cannot rewrite $b = c$ to prove the equality, although it holds:

$$\frac{\frac{a = b}{b = a} \text{sym} \quad a = c}{b = c} \text{trans}$$

Term Ordering

The biggest problem for term rewriting is **(non-)termination**. For some crucial rules, this problem is solved by **ordered** term rewriting. A term ordering is any **partial order** between **ground** (i.e., not containing free variables) terms.

One can define a term ordering by giving some function, called **norm**, from ground terms to natural numbers. Then a term is smaller than another term if the number assigned to the first term is smaller than the number assigned to the second term.

ACI

ACI stands for **associative**, **commutative** and **idempotent**. In

$$(x + y) + z = x + (y + z) \quad (\text{A})$$

$$x + y = y + x \quad (\text{C})$$

$$x + x = x \quad (\text{I})$$

the constant $+$ is written **infix**.

How Ordered Rewriting Solves ACI

Consider an equational theory consisting only of those rules (apart from *refl*, *sym*, *trans*, *subst*). Apart from that, the language may contain arbitrary other constant symbols. For such a language, it is possible to give a term ordering that will assign more weight to the same term on the left-hand-side of a $+$ than on the right-hand side. We can base such a term ordering on a *norm*. For example, the inductive definition of a norm $|_$ might include the line:

$$|s + t| := 2|s| + |t|$$

This means that if $|s| > |t|$, then $|s + t| = 2|s| + |t| > 2|t| + |s| = |t + s|$. This has two effects:

- Applications of (A) or (I) always decrease the weight of a term

(provided the weight of s is > 0):

$$\begin{aligned} |(s + t) + r| &= 2|s + t| + |r| = 4|s| + 2|t| + |r| > \\ 2|s| + 2|t| + |r| &= 2|s| + |t + r| = |s + (t + r)|. \end{aligned}$$

- Applications of (C) are only possible if the left-hand side is heavier than the right-hand side.

We haven't worked out here how the norm should be defined for the other symbols of the language. This would have to depend on that language.

The notation $|-|$ (the argument is **between the bars**) is used in standard mathematics for the absolute value of a number and is standard for **norms** as well.

Ambiguous Matching

For higher-order rewriting, it is very problematic to have rules containing terms of the form $F(G\ c)$ on the left-hand side, where F and G are free variables and c is a constant or bound variable. The reason can be seen in an example: Suppose you want to rewrite the term $f(g(h(i\ c)))$ where f, g, h, i are all constants. There are four unifiers of $F\ (G\ c)$ and $f(g(h(i\ c)))$:

$$\begin{aligned} &\{F \leftarrow f, G \leftarrow (\lambda x. g(h(i\ x)))\}, \\ &\{F \leftarrow (\lambda x. f(g\ x)), G \leftarrow (\lambda x. h(i\ x))\}, \\ &\{(F \leftarrow \lambda x. f(g(h\ x))), G \leftarrow (\lambda x. i\ x)\}, \\ &\{(F \leftarrow \lambda x. f(g(h(i\ x)))), G \leftarrow (\lambda x. x)\}. \end{aligned}$$

This ambiguity makes such **TRSs** very inefficient.

\forall, \exists is a Constant

Further examples:

- $(\exists x.Px \vee Qx) = (\exists x.Px) \vee (\exists x.Qx)$
- $(\exists x.P \rightarrow Qx) = P \rightarrow (\exists x.Qx)$
- $(\exists x.Px \rightarrow Q) = (\forall x.Px) \rightarrow Q$

In these examples, you may assume that first-order logic is our object logic.

On the [metalevel](#), and hence also for the sake of term rewriting, \forall, \exists are [constants](#).

In the notation $(\forall x.Px \wedge Qx)$, the symbols P and Q are metavariables (as far as term rewriting is concerned, simply think: variables).

Actually, $(\forall x.Px \wedge Qx)$ mixes object and metalevel syntax in a way which is typical for Isabelle: $(\forall x.Px \wedge Qx)$ is a “pretty-printed” version

of ALL ($P \ \& \ Q$).

You may want to look at a theory file (say, `IFOL.thy`) to get a flavor of this. The principle was explained thoroughly [before](#).

Nested \implies

Rewrite rules have the form $\phi_1 = \psi_1, \dots, \phi_n = \psi_n \implies \phi = \psi$ (several equations imply one equation). It is not possible that any of the equations $\phi_1 = \psi_1, \dots, \phi_n = \psi_n$ again depend on some condition, as in

$$\text{if } A \text{ then } P \text{ else } Q = \text{if } A \text{ then } P' \text{ else } Q$$

where $P = P'$ under condition A

`simpset`

The `simpset` is an abstract datatype and at the same time an ML unit function for returning the current simplifier set. This is in analogy to the `classifier set`.

Accessing the simpset

These function manipulate the simplifier set, in analogy to the [classifier set](#).

Global simpset

`Simp_tac`, `Asm_simp_tac`, `Asm_full_simp_tac` work like their lower-case counterparts but use the current (global) simplifier set and hence do not take a simplifier set as first argument (e.g., `Simp_tac` has type `int → tactic`)

There are analogous capitalized versions for the tactics of the [classical reasoner](#).

Failure in Simplifier

When you use `simp_tac`, usually you can just look at the term that you get to understand which simplification has not worked although you think that it should have worked.

Isabelle's Metalogic

Representing Syntax and Proofs

- Previously, we have seen how the (polymorphically) typed λ -calculus can be used to represent the **syntax** of an object logic.
- Today, we will extend the λ -calculus to a **logic** (with formulae and inference rules): Isabelle's metalogic, which goes under the names of **\mathcal{M}** , **Pure**, **HOL**.

This lecture is based on Paulson's work [Pau89].

What Is Formality anyway?

- Ultimately, logic and formal reasoning have to resort to natural language. Proofs of, say, the soundness of a derivation system employ the usual mathematical rigor, but that's all. Imagine this for the situation that we just want to do reasoning in propositional logic and nothing else.
- We will now introduce a logic \mathcal{M} . Its proof system is small!

Proof Techniques = Meta-Theorems

- When constructing proofs, there are
 - aspects that are specific to certain logics and its **logical symbols**: the **proof rules**;
 - aspects that reflect **general principles** of proof building: making and discharging assumptions, **substitution**, **side conditions**, **resolution**.

It seems that the latter must be justified by complicated (and thus error-prone) explanations in natural language.

Proof Techniques = Meta-Theorems

- When constructing proofs, there are
 - aspects that are specific to certain logics and its **logical symbols**: the **proof rules**;
 - aspects that reflect **general principles** of proof building: making and discharging assumptions, **substitution**, **side conditions**, **resolution**.

It seems that the latter must be justified by complicated (and thus error-prone) explanations in natural language.

- Using a metalogic such as \mathcal{M} has two benefits:
 - Shared implementational support for the “general principles”;
 - to a wide extent, the “general principles” are formally derived in \mathcal{M} . This gives a high degree of confidence.

The Logic \mathcal{M}

We first introduce \mathcal{M} just like any other logic, without considering its special role as metalogic. Nonetheless, we use the qualification “meta” to avoid confusion [later](#).

Some variations are possible (mainly: polymorphism/type classes or not), but those are not so important for us.

The Logic \mathcal{M}

We first introduce \mathcal{M} just like any other logic, without considering its special role as metalogic. Nonetheless, we use the qualification “meta” to avoid confusion **later**.

Some variations are possible (mainly: polymorphism/type classes or not), but those are not so important for us.

Would you call λ^{\rightarrow} a **logic**?

The Logic \mathcal{M}

We first introduce \mathcal{M} just like any other logic, without considering its special role as metalogic. Nonetheless, we use the qualification “meta” to avoid confusion **later**.

Some variations are possible (mainly: polymorphism/type classes or not), but those are not so important for us.

Would you call λ^{\rightarrow} a **logic**?

So far, λ^{\rightarrow} is **not** a logic (no **connectives**, no **formulae**). We will now extend it to a logic.

Logic Based on λ^{\rightarrow}

Assume some \mathcal{B} where $bool \in \mathcal{B}$, and some signature Σ where

- $\Rightarrow: bool \rightarrow bool \rightarrow bool \in \Sigma$,
- $\equiv_{\sigma}: \sigma \rightarrow \sigma \rightarrow bool \in \Sigma$ for all types σ , and
- $\bigwedge_{\sigma}: (\sigma \rightarrow bool) \rightarrow bool \in \Sigma$ for all types σ .

We usually omit type subscript and write \equiv , \bigwedge .

\Rightarrow , \equiv , and \bigwedge are the logical symbols of \mathcal{M} . \Rightarrow and \equiv are written infix.

Terms of type $bool$ are called

Logic Based on λ^{\rightarrow}

Assume some \mathcal{B} where $bool \in \mathcal{B}$, and some signature Σ where

- $\Rightarrow: bool \rightarrow bool \rightarrow bool \in \Sigma$,
- $\equiv_{\sigma}: \sigma \rightarrow \sigma \rightarrow bool \in \Sigma$ for all types σ , and
- $\bigwedge_{\sigma}: (\sigma \rightarrow bool) \rightarrow bool \in \Sigma$ for all types σ .

We usually omit type subscript and write \equiv , \bigwedge .

\Rightarrow , \equiv , and \bigwedge are the logical symbols of \mathcal{M} . \Rightarrow and \equiv are written infix.

Terms of type $bool$ are called (meta-)formulae: types generalize syntactic categories.

Folding Assumptions

Lists of (meta-)formulae are denoted by Φ, Ψ, Ω . If Φ is the list $[\phi_1, \dots, \phi_n]$, then

$$\begin{aligned} [\phi_1, \dots, \phi_n] &\Rightarrow \psi, \text{ i.e.} \\ \Phi &\Rightarrow \psi \end{aligned}$$

abbreviates the meta-formula $\phi_1 \Rightarrow \dots \Rightarrow \phi_n \Rightarrow \psi$.

You have seen this in the exercises.

Note that $[\phi_1, \dots, \phi_n]$ on its own is not a term in \mathcal{M} !

Proof System for \mathcal{M}

The proof system will be presented in the style of **natural deduction**.

This is as formal as we get (for the metalogic): derivation trees in natural deduction style are authoritative.

The **judgements**, just like for **natural deduction proofs** in propositional logic or first-order logic, are **formulae**, i.e., terms of type *bool*. This is in contrast to **derivability judgements** or **type judgements**.

Rules for \Rightarrow

$$\frac{\begin{array}{c} [\phi] \\ \vdots \\ \psi \end{array}}{\phi \Rightarrow \psi} \Rightarrow -I \quad \frac{\phi \Rightarrow \psi \quad \phi}{\psi} \Rightarrow -E$$

Just like rules for \rightarrow !

Rules for \Rightarrow

$$\frac{\begin{array}{c} [\phi] \\ \vdots \\ \psi \end{array}}{\phi \Rightarrow \psi} \Rightarrow -I \quad \frac{\phi \Rightarrow \psi \quad \phi}{\psi} \Rightarrow -E$$

Just like rules for \rightarrow !

For layout reasons we sometimes use

$$\frac{\phi \quad \phi \Rightarrow \psi}{\psi} \Rightarrow -E$$

Rules for \bigwedge

Meta-universal-quantification is formalized in the style of higher-order abstract syntax; may write $\bigwedge x.\phi$ as syntactic sugar for $\bigwedge(\lambda x.\phi)$.

Note: quantification over terms of arbitrary type!

Rules for \bigwedge

Meta-universal-quantification is formalized in the style of **higher-order abstract syntax**; may write $\bigwedge x.\phi$ as **syntactic sugar** for $\bigwedge(\lambda x.\phi)$.

Note: quantification over terms of arbitrary type!

Rules:

$$\frac{\phi}{\bigwedge x.\phi} \bigwedge\text{-}I^* \quad \frac{\bigwedge x.\phi}{\phi[x \leftarrow b]} \bigwedge\text{-}E$$

Side (**eigenvariable**) condition *: x is not free in any assumption on which ϕ depends.

Just like **rules for \forall** .

Rules for \equiv : Equivalence Relation

$$\frac{}{a \equiv a} \equiv\text{-refl}$$

$$\frac{a \equiv b}{b \equiv a} \equiv\text{-symm}$$

$$\frac{a \equiv b \quad b \equiv c}{a \equiv c} \equiv\text{-trans}$$

Just like rules for $=$.

Rules for \equiv : λ (i.e., α, β, η) Conversions

$$\frac{}{(\lambda x.a) \equiv (\lambda y.a[x \leftarrow y])} \alpha^* \qquad \frac{}{(\lambda x.a)b \equiv (a[x \leftarrow b])} \beta$$

$$\frac{}{(\lambda x.f \ x) \equiv f} \eta^{**}$$

Side condition *: y is not free in a .

Side condition **: x is not free in f .

Just like rules for $=_{\alpha, \beta, \eta}$.

η is equivalent to **extensionality**.

Rules for \equiv : Abstraction, Combination

$$\frac{a \equiv b}{(\lambda x.a) \equiv (\lambda x.b)} \equiv\text{-}abstr^* \qquad \frac{f \equiv g \quad a \equiv b}{f a \equiv g b} \equiv\text{-}comb$$

Side (**eigenvariable**) condition *: x is not free in any assumption on which $a \equiv b$ depends.

As defined for \rightarrow_β **before**, \equiv is propagated into contexts.

Conversion is built into the proof system!

Recall that $e \equiv e'$ is decidable in λ^\rightarrow (\equiv -rules so far).

However, $e \equiv e'$ is not decidable in \mathcal{M} (see next slide).

Rules for \equiv : Introduction and Elimination

$$\frac{\begin{array}{c} [\phi] \\ \vdots \\ \psi \end{array} \quad \begin{array}{c} [\psi] \\ \vdots \\ \phi \end{array}}{\phi \equiv \psi} \equiv\text{-}I \qquad \frac{\phi \equiv \psi \quad \phi}{\psi} \equiv\text{-}E$$

What is the type of ϕ and ψ here?

Rules for \equiv : Introduction and Elimination

$$\frac{\begin{array}{c} [\phi] \\ \vdots \\ \psi \end{array} \quad \begin{array}{c} [\psi] \\ \vdots \\ \phi \end{array}}{\phi \equiv \psi} \equiv\text{-}I \qquad \frac{\phi \equiv \psi \quad \phi}{\psi} \equiv\text{-}E$$

What is the type of ϕ and ψ here? ϕ and ψ are **formulae**, hence *bool*.

What object-level connective does \equiv correspond to?

Rules for \equiv : Introduction and Elimination

$$\frac{\begin{array}{c} [\phi] \\ \vdots \\ \psi \end{array} \quad \begin{array}{c} [\psi] \\ \vdots \\ \phi \end{array}}{\phi \equiv \psi} \equiv\text{-}I \qquad \frac{\phi \equiv \psi \quad \phi}{\psi} \equiv\text{-}E$$

What is the type of ϕ and ψ here? ϕ and ψ are **formulae**, hence *bool*.

What object-level connective does \equiv correspond to? \leftrightarrow .

Encoding Syntax and Provability

We use **FOL** and its subset **propositional logic** (which we call here *Prop*) as exemplary object logic.

We already know how to **encode syntax**.

We will now see how to encode **proof rules** and mimic proofs of the object logic.

To encode a particular object logic L , we have to extend \mathcal{M} by extending the **type language**, the term language (the **signature**) and the proof rules. The thus extended logic will be called \mathcal{M}_L .

Encoding Syntax: Review

As before, $i, o \in \mathcal{B}$. Previously:

$$\Sigma \supseteq \langle \text{not} : o \rightarrow o, \text{and} : o \rightarrow o \rightarrow o, \text{imp} : o \rightarrow o \rightarrow o, \\ \text{all} : (i \rightarrow o) \rightarrow o, \text{exists} : (i \rightarrow o) \rightarrow o \rangle$$

Two types for truth values: o and $bool$.

We now need a more concise (**sweeter**) syntax or things will become hopelessly unreadable.

But this is also quite demanding: you should always be able to “unsugar” the syntax.

Encoding Syntax Readably

$$\begin{aligned}\Sigma \supseteq \langle & \perp : o, \\ & \wedge, \vee, \rightarrow : o \rightarrow o \rightarrow o, \\ & \forall, \exists : (i \rightarrow o) \rightarrow o, \\ & \text{true} : o \rightarrow \text{bool} \rangle.\end{aligned}$$

\rightarrow is both a constant declared in Σ and the **function type arrow**.

$\wedge, \vee, \rightarrow$ will be written **infix**, and we may write $\forall x.\phi$ for $\forall(\lambda x.\phi)$, and likewise for \exists .

true A is usually written $\llbracket A \rrbracket$.

Encoding the Rules

The **rules** of the object logic are encoded as **axioms** of the metalogic. These axioms are added to the proof system of \mathcal{M} (to obtain \mathcal{M}_L).

To avoid confusion, we will use distinctive terminology:

There is a **meta-rule** called $\Rightarrow\text{-}E$.

There is a similar **object rule** that we call the $\rightarrow\text{-}E$ rule.

It is encoded as a **meta-axiom** that we call the $\rightarrow\text{-}E$ axiom.

Encoding of the Rules of Propositional Logic

$\wedge AB.[A] \Rightarrow ([B] \Rightarrow [A \wedge B])$	$(\wedge-I)$
$\wedge AB.[A \wedge B] \Rightarrow [A]$	$(\wedge-EL)$
$\wedge AB.[A \wedge B] \Rightarrow [B]$	$(\wedge-ER)$
$\wedge AB.[A] \Rightarrow [A \vee B]$	$(\vee-IL)$
$\wedge AB.[B] \Rightarrow [A \vee B]$	$(\vee-IR)$
$\wedge ABC.[A \vee B] \Rightarrow$ $([A] \Rightarrow [C]) \Rightarrow ([B] \Rightarrow [C]) \Rightarrow [C]$	$(\vee-E)$
$\wedge AB.([A] \Rightarrow [B]) \Rightarrow [A \rightarrow B]$	$(\rightarrow-I)$
$\wedge AB.[A \rightarrow B] \Rightarrow [A] \Rightarrow [B]$	$(\rightarrow-E)$
$\wedge A.[\perp] \Rightarrow [A]$	$(\perp-E)$

Faithful Metalogics

For any object logic L , we define:

- \mathcal{M}_L is **sound for L** if, for every proof of $\llbracket B \rrbracket$ from assumptions $\llbracket A_1 \rrbracket, \dots, \llbracket A_m \rrbracket$ in \mathcal{M}_L , there is a proof of B from assumptions A_1, \dots, A_m in L .
- \mathcal{M}_L is **complete for L** if, for every proof of B from assumptions A_1, \dots, A_m in L , there is a proof of $\llbracket B \rrbracket$ from assumptions $\llbracket A_1 \rrbracket, \dots, \llbracket A_m \rrbracket$ in \mathcal{M}_L .
- \mathcal{M}_L is **faithful for L** if \mathcal{M}_L is sound and complete for L .

Using concepts of Prawitz [Pra65, Pra71], one can show by structural induction that \mathcal{M}_{Prop} is faithful for *Prop*.

An Example Proof

$$\frac{}{\wedge AB. \llbracket A \wedge B \rrbracket \Rightarrow \llbracket A \rrbracket} \wedge\text{-}EL$$

An Example Proof

$$\frac{\frac{\frac{}{\wedge AB. \llbracket A \wedge B \rrbracket}}{\Rightarrow \llbracket A \rrbracket}}{\wedge B. \llbracket P \wedge B \rrbracket \Rightarrow \llbracket P \rrbracket} \wedge\text{-}E$$

An Example Proof

$$\begin{array}{c}
 \frac{}{\wedge AB. \llbracket A \wedge B \rrbracket \Rightarrow \llbracket A \rrbracket} \wedge\text{-}EL \\
 \frac{}{\wedge B. \llbracket P \wedge B \rrbracket \Rightarrow \llbracket P \rrbracket} \wedge\text{-}E \\
 \frac{}{\llbracket P \wedge Q \rrbracket \Rightarrow \llbracket P \rrbracket} \wedge\text{-}E
 \end{array}$$

An Example Proof

$$\begin{array}{c}
\frac{\wedge AB. \llbracket A \wedge B \rrbracket}{\Rightarrow \llbracket A \rrbracket} \wedge\text{-}EL \\
\frac{\wedge B. \llbracket P \wedge B \rrbracket}{\Rightarrow \llbracket P \rrbracket} \wedge\text{-}E \\
\frac{\llbracket P \wedge Q \rrbracket \quad \frac{\llbracket P \wedge Q \rrbracket \Rightarrow \llbracket P \rrbracket}{\Rightarrow\text{-}E}}{\llbracket P \rrbracket} \wedge\text{-}E
\end{array}$$

An Example Proof

$$\begin{array}{c}
 \frac{}{\wedge AB. \llbracket A \wedge B \rrbracket \Rightarrow \llbracket A \rrbracket} \wedge\text{-}EL \\
 \frac{}{\wedge B. \llbracket P \wedge B \rrbracket \Rightarrow \llbracket P \rrbracket} \wedge\text{-}E \\
 \frac{\llbracket \llbracket P \wedge Q \rrbracket \rrbracket^1 \quad \llbracket P \wedge Q \rrbracket \Rightarrow \llbracket P \rrbracket}{\llbracket P \rrbracket} \wedge\text{-}E \\
 \frac{\llbracket P \rrbracket}{\llbracket P \wedge Q \rrbracket \Rightarrow \llbracket P \rrbracket} \Rightarrow\text{-}E^1
 \end{array}$$

An Example Proof

$$\begin{array}{c}
 \frac{}{\wedge AB.([A] \Rightarrow [B]) \Rightarrow [A \rightarrow B]} \rightarrow\text{-I} \\
 \frac{}{\wedge B.([P \wedge Q] \Rightarrow [B]) \Rightarrow [P \wedge Q \rightarrow B]} \wedge\text{-E} \\
 \frac{}{([P \wedge Q] \Rightarrow [P]) \Rightarrow [P \wedge Q \rightarrow P]} \wedge\text{-E}
 \end{array}
 \quad
 \begin{array}{c}
 \frac{}{\wedge AB.([A \wedge B] \Rightarrow [A])} \wedge\text{-EL} \\
 \frac{}{\wedge B.([P \wedge B] \Rightarrow [P])} \wedge\text{-E} \\
 \frac{[[P \wedge Q]]^1 \quad \frac{}{[P \wedge Q] \Rightarrow [P]} \wedge\text{-E}}{[P]} \Rightarrow\text{-E} \\
 \frac{}{[P \wedge Q] \Rightarrow [P]} \Rightarrow\text{-I}^1
 \end{array}$$

An Example Proof

$$\begin{array}{c}
 \frac{}{\wedge AB.([A] \Rightarrow [B]) \Rightarrow [A \rightarrow B]} \rightarrow -I \\
 \frac{}{\wedge B.([P \wedge Q] \Rightarrow [B]) \Rightarrow [P \wedge Q \rightarrow B]} \wedge -E \\
 \frac{}{([P \wedge Q] \Rightarrow [P]) \Rightarrow [P \wedge Q \rightarrow P]} \wedge -E \\
 \hline
 [P \wedge Q \rightarrow P] \Rightarrow -E
 \end{array}
 \quad
 \begin{array}{c}
 \frac{}{\wedge AB.([A \wedge B] \Rightarrow [A])} \wedge -EL \\
 \frac{}{\wedge B.([P \wedge B] \Rightarrow [P])} \wedge -E \\
 \frac{[[P \wedge Q]]^1 \quad [P \wedge Q] \Rightarrow [P]}{[P]} \wedge -E \\
 \frac{[P]}{[P \wedge Q] \Rightarrow [P]} \Rightarrow -I^1 \\
 \hline
 [P \wedge Q] \Rightarrow [P] \Rightarrow -E
 \end{array}$$

An Example Proof

$$\begin{array}{c}
\frac{}{\wedge AB.([A] \Rightarrow [B]) \Rightarrow [A \rightarrow B]} \rightarrow\text{-I} \\
\frac{}{\wedge B.([P \wedge Q] \Rightarrow [B]) \Rightarrow [P \wedge Q \rightarrow B]} \wedge\text{-E} \\
\frac{}{([P \wedge Q] \Rightarrow [P]) \Rightarrow [P \wedge Q \rightarrow P]} \wedge\text{-E} \\
\hline
[P \wedge Q \rightarrow P] \Rightarrow\text{-E}
\end{array}
\qquad
\begin{array}{c}
\frac{}{\wedge AB.[A \wedge B] \Rightarrow [A]} \wedge\text{-EL} \\
\frac{}{\wedge B.[P \wedge B] \Rightarrow [P]} \wedge\text{-E} \\
\frac{}{[P \wedge Q] \Rightarrow [P]} \wedge\text{-E} \\
\frac{[[P \wedge Q]]^1}{[P \wedge Q] \Rightarrow [P]} \Rightarrow\text{-E} \\
\frac{[P]}{[P \wedge Q] \Rightarrow [P]} \Rightarrow\text{-I}^1 \\
\hline
[P \wedge Q \rightarrow P] \Rightarrow\text{-E}
\end{array}$$

Example Proof Simplified

$$\begin{array}{c}
 \frac{}{\wedge AB.([A] \Rightarrow [B])} \rightarrow -I \\
 \frac{}{\Rightarrow [A \rightarrow B]} \\
 \hline
 \wedge B.([P \wedge Q] \Rightarrow [B]) \quad \wedge -E \\
 \frac{}{\Rightarrow [P \wedge Q \rightarrow B]} \\
 \hline
 \wedge -E \\
 \frac{}{([P \wedge Q] \Rightarrow [P])} \\
 \frac{}{\Rightarrow [P \wedge Q \rightarrow P]} \\
 \hline
 [P \wedge Q \rightarrow P] \quad \Rightarrow -E
 \end{array}$$

$$\begin{array}{c}
 \frac{}{\wedge AB.[A \wedge B]} \wedge -EL \\
 \frac{}{\Rightarrow [A]} \\
 \hline
 \wedge -E \\
 \frac{}{\wedge B.[P \wedge B]} \\
 \frac{}{\Rightarrow [P]} \\
 \hline
 \wedge -E \\
 \frac{}{[P \wedge Q] \Rightarrow [P]} \\
 \Rightarrow -E
 \end{array}$$

Remarks about Example Proof

- \wedge - EL and \rightarrow - E are not **object rules** but **meta-axioms**!
- The first, more complicated proof corresponds to the construction one would use to show that \mathcal{M}_{Prop} is **complete for $Prop$** .
- Proof fragments of the form

$$\frac{\phi \Rightarrow \psi \quad [\phi]}{\psi} \Rightarrow\text{-}E$$

$$\frac{\psi}{\phi \Rightarrow \psi} \Rightarrow\text{-}I$$

can be collapsed into $\phi \Rightarrow \psi$: **proof normalization**.

Reasoning with Resolution

In Isabelle, we mainly use **backwards reasoning**: we construct a proof tree starting from the root working to the leaves.

Reasoning with Resolution

In Isabelle, we mainly use **backwards reasoning**: we construct a proof tree starting from the root working to the leaves.

On the meta-level, this proof is in fact a **forwards proof**: working from the leaves to the root.

This is achieved by starting the proof of ψ with the **trivial meta-theorem** $\psi \Rightarrow \psi$ and using a technique called **resolution**.

The Resolution Rule

For any formulae $\psi_1, \dots, \psi_n, \psi, \phi_1, \dots, \phi_m, \phi$ where $FV(\phi_1, \dots, \phi_m, \phi) \subseteq \{x_1, \dots, x_k\}$, and $\phi\theta \equiv \psi_i$ for some $i \in \{1, \dots, n\}$, **resolution** is the following rule:

$$\frac{\bigwedge x_1 \dots x_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi \quad [\psi_1, \dots, \psi_n] \Rightarrow \psi}{[\psi_1, \dots, \psi_{i-1}, \phi_1\theta, \dots, \phi_m\theta, \psi_{i+1}, \dots, \psi_n] \Rightarrow \psi} \text{res}$$

Intuition: $\bigwedge x_1 \dots x_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi$ is a **meta-axiom** such as **\wedge -EL**, $[\psi_1, \dots, \psi_n] \Rightarrow \psi$ is the current goal (proof state).

Compare to **phrasing using \vee !**

We will now derive the rule.

Resolution as Derived Meta-Rule

$$\bigwedge x_1 \dots x_k. \\ [\phi_1, \dots, \phi_m] \Rightarrow \phi$$

Resolution as Derived Meta-Rule

$$\frac{\begin{array}{c} \bigwedge x_1 \dots x_k. \\ [\phi_1, \dots, \phi_m] \Rightarrow \phi \end{array}}{[\phi_1\theta, \dots, \phi_m\theta] \Rightarrow \phi\theta} \wedge\text{-}E$$

Resolution as Derived Meta-Rule

$$\begin{array}{c}
 \bigwedge x_1 \dots x_k. \\
 [\phi_1, \dots, \phi_m] \Rightarrow \phi \\
 \hline
 \phi_1\theta \quad \dots \quad \phi_m\theta \quad \frac{[\phi_1\theta, \dots, \phi_m\theta] \Rightarrow \phi\theta}{\text{}} \quad \textcolor{blue}{\wedge\text{-}E}
 \end{array}$$

Resolution as Derived Meta-Rule

$$\frac{\phi_1\theta \quad \dots \quad \phi_m\theta \quad \frac{\frac{\bigwedge x_1 \dots x_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi}{[\phi_1\theta, \dots, \phi_m\theta] \Rightarrow \phi\theta} \wedge\text{-}E}{\phi\theta} \Rightarrow\text{-}E$$

Resolution as Derived Meta-Rule

$$\begin{array}{c}
 \begin{array}{c}
 \bigwedge x_1 \dots x_k. \\
 [\phi_1, \dots, \phi_m] \Rightarrow \phi \\
 \hline
 \phi_1\theta \quad \dots \quad \phi_m\theta \quad [\phi_1\theta, \dots, \phi_m\theta] \Rightarrow \phi\theta \quad \text{---} \wedge\text{-}E \\
 \hline
 \phi\theta \quad \text{---} \Rightarrow\text{-}E
 \end{array}
 \end{array}
 \quad
 \begin{array}{c}
 [\psi_1, \dots, \psi_n] \\
 \Rightarrow \psi
 \end{array}$$

Resolution as Derived Meta-Rule

$$\begin{array}{c}
 \frac{\frac{\frac{\bigwedge x_1 \dots x_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi}{[\phi_1 \theta, \dots, \phi_m \theta] \Rightarrow \phi \theta} \quad \psi_1 \dots \psi_{i-1} \quad [\psi_1, \dots, \psi_n] \Rightarrow \psi}{\phi \theta} \quad \Rightarrow -E}{\phi \theta} \quad \wedge -E
 \end{array}$$

Resolution as Derived Meta-Rule

$$\begin{array}{c}
 \frac{\phi_1\theta \quad \dots \quad \phi_m\theta}{\phi\theta} \quad \frac{\frac{\frac{\bigwedge x_1 \dots x_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi}{[\phi_1\theta, \dots, \phi_m\theta] \Rightarrow \phi\theta} \quad \wedge\text{-}E}{\phi\theta} \quad \Rightarrow\text{-}E}{\phi\theta}
 \end{array}
 \quad
 \frac{\psi_1 \quad \dots \quad \psi_{i-1} \quad \frac{[\psi_1, \dots, \psi_n] \Rightarrow \psi}{\Rightarrow\psi}}{[\psi_i, \dots, \psi_n] \Rightarrow \psi} \Rightarrow\text{-}E$$

Resolution as Derived Meta-Rule

$$\begin{array}{c}
 \frac{\frac{\frac{\bigwedge x_1 \dots x_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi}{[\phi_1\theta, \dots, \phi_m\theta] \Rightarrow \phi\theta} \quad \frac{\psi_1 \dots \psi_{i-1} \quad [\psi_1, \dots, \psi_n] \Rightarrow \psi}{[\psi_i, \dots, \psi_n] \Rightarrow \psi}}{\phi\theta} \quad \frac{\psi_1 \dots \psi_{i-1} \quad [\psi_1, \dots, \psi_n] \Rightarrow \psi}{[\psi_i, \dots, \psi_n] \Rightarrow \psi}}{\psi_{i+1}, \dots, \psi_n \Rightarrow \psi} \\
 \text{Rules: } \wedge\text{-}E, \Rightarrow\text{-}E, \Rightarrow\text{-}E
 \end{array}$$

Resolution as Derived Meta-Rule

$$\begin{array}{c}
 \frac{\frac{\frac{\frac{\bigwedge x_1 \dots x_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi}{[\phi_1 \theta]^2 \dots [\phi_m \theta]^2} \quad \frac{[\phi_1 \theta, \dots, \phi_m \theta] \Rightarrow \phi \theta}{\Rightarrow -E} \quad \frac{\psi_1 \dots \psi_{i-1} \quad [\psi_1, \dots, \psi_n] \Rightarrow \psi}{\Rightarrow -E}}{[\psi_i, \dots, \psi_n] \Rightarrow \psi} \quad \frac{\phi \theta}{\Rightarrow -E}}{[\psi_{i+1}, \dots, \psi_n] \Rightarrow \psi} \quad \frac{[\phi_1 \theta, \dots, \phi_m \theta, \psi_{i+1}, \dots, \psi_n] \Rightarrow \psi}{\Rightarrow -I^2}
 \end{array}$$

$$\frac{\frac{\frac{\frac{\bigwedge x_1 \dots x_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi}{[\phi_1 \theta]^2 \dots [\phi_m \theta]^2} \quad \frac{[\psi_1]^1 \dots [\psi_{i-1}]^1 \quad [\psi_1, \dots, \psi_n] \Rightarrow \psi}{[\psi_i, \dots, \psi_n] \Rightarrow \psi} \Rightarrow -E}{[\phi_1 \theta, \dots, \phi_m \theta] \Rightarrow \phi \theta} \quad \frac{[\psi_1, \dots, \psi_n] \Rightarrow \psi}{[\psi_i, \dots, \psi_n] \Rightarrow \psi} \Rightarrow -E}{\phi \theta} \Rightarrow -E}{[\psi_{i+1}, \dots, \psi_n] \Rightarrow \psi} \Rightarrow -E}{[\phi_1 \theta, \dots, \phi_m \theta, \psi_{i+1}, \dots, \psi_n] \Rightarrow \psi} \Rightarrow -I^2}{[\psi_1, \dots, \psi_{i-1}, \phi_1 \theta, \dots, \phi_m \theta, \psi_{i+1}, \dots, \psi_n] \Rightarrow \psi} \Rightarrow -I^1$$

Deriving Resolution: Remarks

- We collapsed **iterated applications** of rules (denoted by **double** horizontal line).

Deriving Resolution: Remarks

- We collapsed **iterated applications** of rules (denoted by **double** horizontal line).
- This is not just a matter of simplicity. The derivation is **schematic** not just in the sense that the Greek letters could stand for **arbitrary formulae**; we don't even know **how many** formulae are involved (k, m, n, i could be any natural numbers).
- But for any concrete $\psi_1, \dots, \psi_n, \psi, \phi_1, \dots, \phi_m, \phi$, you could do the formal derivation in \mathcal{M} .

Dropping Outer Quantifiers

We adopt the convention that outer quantifiers in meta-formulae are dropped. E.g. $\llbracket A \rrbracket \Rightarrow \llbracket B \rrbracket \Rightarrow \llbracket A \wedge B \rrbracket$ instead of $\bigwedge AB. \llbracket A \rrbracket \Rightarrow \llbracket B \rrbracket \Rightarrow \llbracket A \wedge B \rrbracket$.

In addition: use **renaming** for **freshness**.

Then we can write the resolution rule as follows:

$$\frac{[\phi_1, \dots, \phi_m] \Rightarrow \phi \quad [\psi_1, \dots, \psi_n] \Rightarrow \psi}{[\psi_1, \dots, \psi_{i-1}, \phi_1\theta, \dots, \phi_m\theta, \psi_{i+1}, \dots, \psi_n] \Rightarrow \psi} \text{res}$$

where $\phi\theta \equiv \psi_i$.

We will now work with this **schematic form**.

Proof of $A \wedge B \rightarrow C \rightarrow A \wedge C$ (1)

Let's prove $A \wedge B \rightarrow (C \rightarrow A \wedge C)$ by resolution. We start like this:

$$\begin{array}{l}
 ([A_1] \Rightarrow [B_1]) \quad [A \wedge B \rightarrow (C \rightarrow A \wedge C)] \\
 \Rightarrow [A_1 \rightarrow B_1] \quad \Rightarrow [A \wedge B \rightarrow (C \rightarrow A \wedge C)] \\
 \hline
 ([A \wedge B] \Rightarrow [C \rightarrow A \wedge C]) \\
 \Rightarrow [A \wedge B \rightarrow (C \rightarrow A \wedge C)]
 \end{array}
 \text{res}$$

What to do next?

Proof of $A \wedge B \rightarrow C \rightarrow A \wedge C$ (1)

Let's prove $A \wedge B \rightarrow (C \rightarrow A \wedge C)$ by resolution. We start like this:

$$\begin{array}{l}
 ([A_1] \Rightarrow [B_1]) \quad [A \wedge B \rightarrow (C \rightarrow A \wedge C)] \\
 \Rightarrow [A_1 \rightarrow B_1] \quad \Rightarrow [A \wedge B \rightarrow (C \rightarrow A \wedge C)] \\
 \hline
 ([A \wedge B] \Rightarrow [C \rightarrow A \wedge C]) \quad \text{res} \\
 \Rightarrow [A \wedge B \rightarrow (C \rightarrow A \wedge C)]
 \end{array}$$

What to do next? Again resolution with \rightarrow -/.

Problem: the conclusion of \rightarrow -/ is not unifiable with $[A \wedge B] \Rightarrow [C \rightarrow A \wedge C]$.

Lifting over Assumptions

The rule for lifting an object rule (**meta-axiom**)
 $[\phi_1, \dots, \phi_m] \Rightarrow \phi$ over a list of assumptions Ψ is

$$\frac{[\phi_1, \dots, \phi_m] \Rightarrow \phi}{[\Psi \Rightarrow \phi_1, \dots, \Psi \Rightarrow \phi_m] \Rightarrow (\Psi \Rightarrow \phi)} \text{a-lift}$$

We will now derive it for **one** assumption, so $\Psi = [\psi]$.

Deriving Assumption Lifting for **one** Assumption

$$[\phi_1, \dots, \phi_m] \Rightarrow \phi \quad \frac{\psi \Rightarrow \phi_1 \quad \psi}{\phi_1} \Rightarrow -E \quad \dots \quad \frac{\psi \Rightarrow \phi_m \quad \psi}{\phi_m} \Rightarrow -E$$

Deriving Assumption Lifting for **one** Assumption

$$\begin{array}{c}
 [\phi_1, \dots, \phi_m] \Rightarrow \phi \quad \frac{\psi \Rightarrow \phi_1 \quad \psi}{\phi_1} \Rightarrow -E \quad \dots \quad \frac{\psi \Rightarrow \phi_m \quad \psi}{\phi_m} \Rightarrow -E \\
 \hline \hline
 \phi \Rightarrow -E
 \end{array}$$

Deriving Assumption Lifting for **one** Assumption

$$\begin{array}{c}
 \frac{[\phi_1, \dots, \phi_m] \Rightarrow \phi \quad \frac{\psi \Rightarrow \phi_1 \quad [\psi]^2}{\phi_1} \Rightarrow -E \quad \dots \quad \frac{\psi \Rightarrow \phi_m \quad [\psi]^2}{\phi_m} \Rightarrow -E}{\phi} \Rightarrow -E \\
 \hline
 \frac{\phi}{\psi \Rightarrow \phi} \Rightarrow -I^2
 \end{array}$$

Deriving Assumption Lifting for **one** Assumption

$$\begin{array}{c}
 \frac{[\phi_1, \dots, \phi_m] \Rightarrow \phi \quad \frac{[\psi \Rightarrow \phi_1]^1 \quad [\psi]^2}{\phi_1} \Rightarrow -E \quad \dots \quad \frac{[\psi \Rightarrow \phi_m]^1 \quad [\psi]^2}{\phi_m} \Rightarrow -E}{\phi} \Rightarrow -E \\
 \frac{\phi}{\psi \Rightarrow \phi} \Rightarrow -I^2 \\
 \hline
 [\psi \Rightarrow \phi_1, \dots, \psi \Rightarrow \phi_m] \Rightarrow (\psi \Rightarrow \phi) \Rightarrow -I^1
 \end{array}$$

This process can be repeated for any number of assumptions to get the general rule.

Proof of $A \wedge B \rightarrow (C \rightarrow A \wedge C)$ (2)

We do resolution using the \rightarrow -/ rule lifted over $A \wedge B$:

$$\begin{array}{c}
 \vdots \\
 \hline
 \begin{array}{ll}
 ([A \wedge B] \Rightarrow ([A_2] \Rightarrow [B_2])) & ([A \wedge B] \Rightarrow [C \rightarrow A \wedge C]) \\
 \Rightarrow ([A \wedge B] \Rightarrow [A_2 \rightarrow B_2]) & \Rightarrow [A \wedge B \rightarrow (C \rightarrow A \wedge C)]
 \end{array} \\
 \hline
 \text{res} \\
 \begin{array}{l}
 [A \wedge B] \Rightarrow [C] \Rightarrow [A \wedge C] \\
 \Rightarrow [A \wedge B \rightarrow (C \rightarrow A \wedge C)]
 \end{array}
 \end{array}$$

Proof of $A \wedge B \rightarrow (C \rightarrow A \wedge C)$ (2)

We do resolution using the \rightarrow -/ rule lifted over $A \wedge B$:

$$\begin{array}{c}
 \vdots \\
 \hline
 \begin{array}{ll}
 ([A \wedge B] \Rightarrow ([A_2] \Rightarrow [B_2])) & ([A \wedge B] \Rightarrow [C \rightarrow A \wedge C]) \\
 \Rightarrow ([A \wedge B] \Rightarrow [A_2 \rightarrow B_2]) & \Rightarrow [A \wedge B \rightarrow (C \rightarrow A \wedge C)]
 \end{array} \\
 \hline
 \text{res} \\
 [A \wedge B] \Rightarrow [C] \Rightarrow [A \wedge C] \\
 \Rightarrow [A \wedge B \rightarrow (C \rightarrow A \wedge C)]
 \end{array}$$

Before we proceed, we introduce the abbreviations

$$\psi = [A \wedge B \rightarrow (C \rightarrow A \wedge C)], \quad \Omega = [[A \wedge B], [C]]$$

Proof of $A \wedge B \rightarrow (C \rightarrow A \wedge C)$ (2)

We do resolution using the \rightarrow -/ rule lifted over $A \wedge B$:

$$\begin{array}{c}
 \vdots \\
 \hline
 \begin{array}{cc}
 ([A \wedge B] \Rightarrow ([A_2] \Rightarrow [B_2])) & ([A \wedge B] \Rightarrow [C \rightarrow A \wedge C]) \\
 \Rightarrow ([A \wedge B] \Rightarrow [A_2 \rightarrow B_2]) & \Rightarrow \psi
 \end{array} \\
 \hline
 \Omega \Rightarrow [A \wedge C] \quad \text{res} \\
 \Rightarrow \psi
 \end{array}$$

Before we proceed, we introduce the abbreviations

$$\psi = [A \wedge B \rightarrow (C \rightarrow A \wedge C)], \quad \Omega = [[A \wedge B], [C]]$$

Proof of $A \wedge B \rightarrow (C \rightarrow A \wedge C)$ (3)

We do resolution using the \wedge -/ rule lifted over Ω :

$$\begin{array}{c}
 (\Omega \Rightarrow \llbracket A_3 \rrbracket) \Rightarrow (\Omega \Rightarrow \llbracket B_3 \rrbracket) \\
 \Rightarrow (\Omega \Rightarrow \llbracket A_3 \wedge B_3 \rrbracket)
 \end{array}
 \quad
 \frac{\vdots}{(\Omega \Rightarrow \llbracket A \wedge C \rrbracket) \Rightarrow \psi}$$

$$(\Omega \Rightarrow \llbracket A \rrbracket) \Rightarrow (\Omega \Rightarrow \llbracket C \rrbracket) \Rightarrow \psi \quad \text{res}$$

At this point, Isabelle would display $\Omega \Rightarrow \llbracket A \rrbracket$ and $\Omega \Rightarrow \llbracket C \rrbracket$ as **two subgoals**.

The next step is to solve $\Omega \Rightarrow \llbracket C \rrbracket$ by assumption, but this must be formalized.

The Assumption Axiom

The assumption axiom is: for any $i \in \{1, \dots, m\}$

$$\frac{}{[\phi_1, \dots, \phi_m] \Rightarrow \phi_i} \text{assum}$$

It has a simple (**schematic**) derivation:

$$\frac{\frac{\frac{[\phi_i]^1}{[\phi_{i+1}, \dots, \phi_m] \Rightarrow \phi_i} \Rightarrow -/}{[\phi_i, \dots, \phi_m] \Rightarrow \phi_i} \Rightarrow -/{}^1}{[\phi_1, \dots, \phi_m] \Rightarrow \phi_i} \Rightarrow -/$$

Proof of $A \wedge B \rightarrow (C \rightarrow A \wedge C)$ (4)

We do resolution using the **assumption axiom**:

$$\frac{\Omega \Rightarrow [C] \quad \frac{\vdots}{(\Omega \Rightarrow [A]) \Rightarrow (\Omega \Rightarrow [C]) \Rightarrow \psi}}{(\Omega \Rightarrow [A]) \Rightarrow \psi} \text{res}$$

We used the correct instance of the assumption axiom.

Alternatively, we could have use the more generic

$$[A_1, A_2] \Rightarrow A_2.$$

What to do next? (Recall that $\Omega = [[A \wedge B], [C]]$.)

Proof of $A \wedge B \rightarrow (C \rightarrow A \wedge C)$ (4)

We do resolution using the **assumption axiom**:

$$\frac{\Omega \Rightarrow [C] \quad \frac{\vdots}{(\Omega \Rightarrow [A]) \Rightarrow (\Omega \Rightarrow [C]) \Rightarrow \psi}}{(\Omega \Rightarrow [A]) \Rightarrow \psi} \text{res}$$

We used the correct instance of the assumption axiom.

Alternatively, we could have use the more generic

$$[A_1, A_2] \Rightarrow A_2.$$

What to do next? (Recall that $\Omega = [[A \wedge B], [C]]$.)

Resolution with \wedge -EL.

Proof of $A \wedge B \rightarrow (C \rightarrow A \wedge C)$ (5)

Magically, we guess the right instance of \wedge - EL and lift it over Ω :

$$\frac{(\Omega \Rightarrow [A \wedge B]) \Rightarrow (\Omega \Rightarrow [A]) \quad \frac{\vdots}{(\Omega \Rightarrow [A]) \Rightarrow \psi}}{(\Omega \Rightarrow [A \wedge B]) \Rightarrow \psi} \text{res}$$

What to do next? (Recall that $\Omega = [[A \wedge B], [C]]$.)

Proof of $A \wedge B \rightarrow (C \rightarrow A \wedge C)$ (5)

Magically, we guess the right instance of \wedge - EL and lift it over Ω :

$$\frac{(\Omega \Rightarrow [A \wedge B]) \Rightarrow (\Omega \Rightarrow [A]) \quad \frac{\vdots}{(\Omega \Rightarrow [A]) \Rightarrow \psi}}{(\Omega \Rightarrow [A \wedge B]) \Rightarrow \psi} \text{res}$$

What to do next? (Recall that $\Omega = [[A \wedge B], [C]]$.) Prove the subgoal by assumption.

Proof of $A \wedge B \rightarrow (C \rightarrow A \wedge C)$ (6)

We do resolution using the **assumption axiom**:

$$\frac{\Omega \Rightarrow [A \wedge B] \quad \frac{\quad \quad \quad \vdots}{(\Omega \Rightarrow [A \wedge B]) \Rightarrow \psi} \quad \psi}{\psi} \text{res}$$

Recall that $\psi = [A \wedge B \rightarrow (C \rightarrow A \wedge C)]$. Done!

Getting Rid of the Magic

In one step, we had to guess the right instance of \wedge - EL .
This is not practical.

Solutions:

- Generalize the resolution rule to allow for instantiation of the current proof state and not just of meta-axioms.
- Derive

$$\bigwedge ABC. [A \wedge B] \Rightarrow ([A] \Rightarrow [B] \Rightarrow [C]) \Rightarrow [C]$$

which encodes the \wedge - E object rule.

The Whole Proof at a Glance

Compare proof in \mathcal{M}_{Prop} with corresponding proof in $Prop$:

$$\begin{array}{c}
 \frac{a. \quad \dots \Rightarrow \psi}{\psi} \quad \frac{\wedge-EL}{\dots \Rightarrow \psi} \\
 \frac{a. \quad \dots \Rightarrow \psi}{\dots \Rightarrow \psi} \quad \frac{\wedge-I}{\dots \Rightarrow \psi} \\
 \frac{\dots \Rightarrow \psi}{\dots \Rightarrow \psi} \quad \frac{\rightarrow-I}{\dots \Rightarrow \psi} \\
 \frac{\psi \Rightarrow \psi}{\psi \Rightarrow \psi} \quad \frac{\rightarrow-I}{\dots \Rightarrow \psi}
 \end{array}$$

$$\begin{array}{c}
 \frac{[A \wedge B]^1}{A} \wedge-EL \quad \frac{[C]^2}{A \wedge C} \wedge-I \\
 \frac{A \wedge C}{C \rightarrow A \wedge C} \rightarrow-I^2 \\
 \frac{C \rightarrow A \wedge C}{A \wedge B \rightarrow (C \rightarrow A \wedge C)} \rightarrow-I^1
 \end{array}$$

The Whole Proof at a Glance

Compare proof in \mathcal{M}_{Prop} with corresponding proof in $Prop$:

$$\begin{array}{c}
 \frac{a. \quad \frac{\frac{\frac{\frac{\frac{\psi \Rightarrow \psi}{\rightarrow -I}}{\dots \Rightarrow \psi}}{\rightarrow -I}}{\dots \Rightarrow \psi}}{\wedge -I}}{a. \quad \dots \Rightarrow \psi}}{\wedge -EL} \quad \dots \Rightarrow \psi \\
 \hline
 a. \quad \dots \Rightarrow \psi \\
 \hline
 \psi
 \end{array}
 \qquad
 \begin{array}{c}
 \frac{[A \wedge B]^1}{A} \wedge -EL \quad [C]^2 \\
 \hline
 \frac{A \wedge C}{C \rightarrow A \wedge C} \wedge -I \\
 \hline
 \frac{C \rightarrow A \wedge C}{A \wedge B \rightarrow (C \rightarrow A \wedge C)} \rightarrow -I^2 \\
 \hline
 A \wedge B \rightarrow (C \rightarrow A \wedge C) \rightarrow -I^1
 \end{array}$$

“The meta-level proof is the object level proof **upside-down**.”

Direction of the Implication

Is the direction of the implication reversed just because we go from the object to the meta-level?

Direction of the Implication

Is the direction of the implication reversed just because we go from the object to the meta-level?

No! The direction is reversed because we start from the trivial meta-theorem $\psi \Rightarrow \psi$, and the resolution steps modify the left-hand side of this meta-theorem.

How Can One Turn a Tree Upside-Down?

A proper tree has nodes where it **branches**. Also, in Isabelle proofs, we frequently have to prove several subgoals. So how is this branching reflected in the meta-proof?

How Can One Turn a Tree Upside-Down?

A proper tree has nodes where it **branches**. Also, in Isabelle proofs, we frequently have to prove several subgoals. So how is this branching reflected in the meta-proof?

A meta-formula of the form $\psi_1 \Rightarrow \dots \Rightarrow \psi_n \Rightarrow \psi$ corresponds to a branching point in the object level proof. It means that there are **subgoals** ψ_1, \dots, ψ_n . But in the derivation tree in \mathcal{M}_{Prop} , there is no branching.

How Can One Turn a Tree Upside-Down?

A proper tree has nodes where it **branches**. Also, in Isabelle proofs, we frequently have to prove several subgoals. So how is this branching reflected in the meta-proof?

A meta-formula of the form $\psi_1 \Rightarrow \dots \Rightarrow \psi_n \Rightarrow \psi$ corresponds to a branching point in the object level proof. It means that there are **subgoals** ψ_1, \dots, ψ_n . But in the derivation tree in \mathcal{M}_{Prop} , there is no branching.

In the construction of a meta-proof (just like in Isabelle), one is always free to choose which subgoal to solve next.

Interleaving is possible.

Quantification

We have to add the following meta-axioms to obtain \mathcal{M}_{FOL} :

$$\begin{aligned}\bigwedge F.(\bigwedge x.[F x]) &\Rightarrow [\forall x.F x] && (\forall-I) \\ \bigwedge Fy.[\forall x.F x] &\Rightarrow [F y] && (\forall-E) \\ \bigwedge Fy.[F y] &\Rightarrow [\exists x.F x] && (\exists-I) \\ \bigwedge FB.[\exists x.F x] &\Rightarrow (\bigwedge x.[F x] \Rightarrow [B]) \Rightarrow [B] && (\exists-E)\end{aligned}$$

Similarly as for *Prop*, one can show that \mathcal{M}_{FOL} is faithful for FOL.

Side condition checking is shifted to the meta-level.

We now consider resolution proofs for FOL.

Proof of $(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)$ (1)

$$\begin{array}{l}
 ([A_1] \Rightarrow [B_1]) \quad \quad \quad [(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)] \\
 \Rightarrow [A_1 \rightarrow B_1] \quad \Rightarrow [(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)] \\
 \hline
 \quad \quad \quad ([\forall z.G\ z] \Rightarrow [\forall z.G\ z \vee H\ z]) \\
 \quad \quad \Rightarrow [(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)]
 \end{array}
 \text{res}$$

What to do next?

Proof of $(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)$ (1)

$$\begin{array}{l}
 ([A_1] \Rightarrow [B_1]) \quad [(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)] \\
 \Rightarrow [A_1 \rightarrow B_1] \quad \Rightarrow [(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)] \\
 \hline
 ([\forall z.G\ z] \Rightarrow [\forall z.G\ z \vee H\ z]) \quad \text{res} \\
 \Rightarrow [(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)]
 \end{array}$$

What to do next? Resolution with \forall -/ lifted over assumption $[\forall z.G\ z]$.

Proof of $(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)$ (2)

$$\begin{array}{c}
 \vdots \\
 \hline
 \begin{array}{l}
 ([\forall z.G\ z] \Rightarrow (\bigwedge x.[F_1\ x])) \quad ([\forall z.G\ z] \Rightarrow [\forall z.G\ z \vee H\ z]) \\
 \Rightarrow ([\forall z.G\ z] \Rightarrow [\forall x.F_1\ x]) \quad \Rightarrow [(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)]
 \end{array} \\
 \hline
 \text{res} \\
 \begin{array}{l}
 ([\forall z.G\ z] \Rightarrow (\bigwedge z.[G\ z \vee H\ z])) \\
 \Rightarrow [(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)]
 \end{array}
 \end{array}$$

The substitution θ is $[F_1 \leftarrow \lambda w.G\ w \vee H\ w]$.

We suppress conversion, assuming terms are in normal form.

What to do next?

Proof of $(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)$ (2)

$$\begin{array}{c}
 \vdots \\
 \hline
 \begin{array}{l}
 (\llbracket \forall z.G\ z \rrbracket \Rightarrow (\bigwedge x.\llbracket F_1\ x \rrbracket)) \\
 \Rightarrow (\llbracket \forall z.G\ z \rrbracket \Rightarrow \llbracket \forall x.F_1\ x \rrbracket)
 \end{array}
 \quad
 \begin{array}{l}
 (\llbracket \forall z.G\ z \rrbracket \Rightarrow \llbracket \forall z.G\ z \vee H\ z \rrbracket) \\
 \Rightarrow \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket
 \end{array}
 \end{array}
 \quad \text{res}$$

$$\begin{array}{l}
 (\llbracket \forall z.G\ z \rrbracket \Rightarrow (\bigwedge z.\llbracket G\ z \vee H\ z \rrbracket)) \\
 \Rightarrow \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket
 \end{array}$$

The substitution θ is $[F_1 \leftarrow \lambda w.G\ w \vee H\ w]$.

We suppress **conversion**, assuming terms are in **normal form**.

What to do next? Resolution with \forall -*IL* after **lifting over assumption**. Problem: the conclusion of \forall -*IL* is not unifiable with $\bigwedge z.\llbracket G\ z \vee H\ z \rrbracket$.

Lifting over Parameters

Lifting over parameters seems easier to explain if outer \wedge 's are **not dropped**. The rule for lifting a **meta-axiom**

$\wedge y_1 \dots y_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi$ over a parameter z is

$$\frac{\wedge y_1 \dots y_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi}{\wedge f_1 \dots f_k. [\wedge z. \phi'_1, \dots, \wedge z. \phi'_m] \Rightarrow (\wedge z. \phi')} \text{ } p\text{-lift}$$

where $'$ stands for application of the substitution $[y_1 \leftarrow f_1 z, \dots, y_k \leftarrow f_k z]$.

We will now derive it.

Deriving Parameter Lifting for **one** Parameter

' stands for application of $[y_1 \leftarrow f_1(z), \dots, y_k \leftarrow f_k(z)]$.

$$\begin{array}{c}
 \frac{\bigwedge y_1 \dots y_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi}{[\phi'_1, \dots, \phi'_m] \Rightarrow \phi'} \quad \Lambda\text{-}E \quad \frac{\bigwedge z. \phi'_1}{\phi'_1} \quad \Lambda\text{-}E \quad \dots \quad \frac{\bigwedge z. \phi'_m}{\phi'_m} \quad \Lambda\text{-}E \\
 \hline
 \frac{\phi'}{\bigwedge z. \phi'} \quad \Lambda\text{-}I \quad \Rightarrow\text{-}E
 \end{array}$$

Deriving Parameter Lifting for **one** Parameter

' stands for application of $[y_1 \leftarrow f_1(z), \dots, y_k \leftarrow f_k(z)]$.

$$\begin{array}{c}
 \frac{\Lambda y_1 \dots y_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi}{[\phi'_1, \dots, \phi'_m] \Rightarrow \phi'} \quad \Lambda\text{-}E \quad \frac{[\Lambda z. \phi'_1]^1}{\phi'_1} \quad \Lambda\text{-}E \quad \dots \quad \frac{[\Lambda z. \phi'_m]^1}{\phi'_m} \quad \Lambda\text{-}E \\
 \hline
 \hline
 \frac{\phi'}{\Lambda z. \phi'} \quad \Lambda\text{-}I \\
 \hline
 \frac{[\Lambda z. \phi'_1, \dots, \Lambda z. \phi'_m] \Rightarrow \Lambda z. \phi'}{\Lambda f_1 \dots f_k. [\Lambda z. \phi'_1, \dots, \Lambda z. \phi'_m] \Rightarrow \Lambda z. \phi'} \quad \Rightarrow\text{-}I^1 \quad \Lambda\text{-}I
 \end{array}$$

Deriving Parameter Lifting for **one** Parameter

' stands for application of $[y_1 \leftarrow f_1(z), \dots, y_k \leftarrow f_k(z)]$.

$$\begin{array}{c}
 \frac{\Lambda y_1 \dots y_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi}{[\phi'_1, \dots, \phi'_m] \Rightarrow \phi'} \quad \Lambda\text{-}E \quad \frac{[\Lambda z. \phi'_1]^1}{\phi'_1} \quad \Lambda\text{-}E \quad \dots \quad \frac{[\Lambda z. \phi'_m]^1}{\phi'_m} \quad \Lambda\text{-}E \\
 \hline
 \hline
 \frac{\phi'}{\Lambda z. \phi'} \quad \Lambda\text{-}I \\
 \hline
 \frac{[\Lambda z. \phi'_1, \dots, \Lambda z. \phi'_m] \Rightarrow \Lambda z. \phi' \Rightarrow\text{-}I^1}{[\Lambda z. \phi'_1, \dots, \Lambda z. \phi'_m] \Rightarrow \Lambda z. \phi'} \quad \Lambda\text{-}I
 \end{array}$$

After parameter lifting, we **drop** outer quantifiers again.

Lifting \vee -IL

Lifting $\bigwedge AB. \llbracket A \rrbracket \Rightarrow \llbracket A \vee B \rrbracket$ (\vee -IL) over z gives

$$\bigwedge G_2 H_2. (\bigwedge z. \llbracket G_2 z \rrbracket) \Rightarrow (\bigwedge z. \llbracket G_2 z \vee H_2 z \rrbracket).$$

Lifting \vee -IL

Lifting $\bigwedge AB.[A] \Rightarrow [A \vee B]$ (\vee -IL) over z gives

$$(\bigwedge z.[G_2 z]) \Rightarrow (\bigwedge z.[G_2 z \vee H_2 z]).$$

We **drop** outer quantifiers and **lift over assumption** $[\forall z.G z]$ to obtain

$$([\forall z.G z] \Rightarrow \bigwedge z.[G_2 z]) \Rightarrow \\ ([\forall z.G z] \Rightarrow \bigwedge z.[G_2 z \vee H_2 z])$$

This rule will be applied in the next step.

Proof of $(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)$ (3)

$$\begin{array}{c}
 \vdots \\
 \hline
 \begin{array}{l}
 (\llbracket \forall z.G\ z \rrbracket \Rightarrow \bigwedge z.\llbracket G_2\ z \rrbracket) \Rightarrow \quad (\llbracket \forall z.G\ z \rrbracket \Rightarrow (\bigwedge z.\llbracket G\ z \vee H\ z \rrbracket)) \\
 (\llbracket \forall z.G\ z \rrbracket \Rightarrow \bigwedge z.\llbracket G_2\ z \vee H_2\ z \rrbracket) \quad \Rightarrow \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket
 \end{array} \\
 \hline
 \text{res}
 \end{array}$$

$$\begin{array}{l}
 (\llbracket \forall z.G\ z \rrbracket \Rightarrow \bigwedge z.\llbracket G\ z \rrbracket) \Rightarrow \\
 \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket
 \end{array}$$

What to do next?

Proof of $(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)$ (3)

$$\begin{array}{c}
 \vdots \\
 \hline
 \begin{array}{l}
 (\llbracket \forall z.G\ z \rrbracket \Rightarrow \bigwedge z.\llbracket G_2\ z \rrbracket) \Rightarrow \quad (\llbracket \forall z.G\ z \rrbracket \Rightarrow (\bigwedge z.\llbracket G\ z \vee H\ z \rrbracket)) \\
 (\llbracket \forall z.G\ z \rrbracket \Rightarrow \bigwedge z.\llbracket G_2\ z \vee H_2\ z \rrbracket) \quad \Rightarrow \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket
 \end{array} \\
 \hline
 \text{res} \\
 \begin{array}{l}
 (\llbracket \forall z.G\ z \rrbracket \Rightarrow \bigwedge z.\llbracket G\ z \rrbracket) \Rightarrow \\
 \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket
 \end{array}
 \end{array}$$

What to do next? Resolution with \forall -E lifted over z .

However, this cannot be guessed from looking at $\bigwedge z.\llbracket G\ z \rrbracket$, but rather from looking at premise $\llbracket \forall z.G\ z \rrbracket$.

Lifting of \forall -E over z

Lifting $\bigwedge Fy. [\forall x. F x] \Rightarrow [F y]$ (\forall -E) over parameter z gives

$$\bigwedge G_3 f_3. (\bigwedge z. [\forall x. (G_3 z)x]) \Rightarrow (\bigwedge z. [G_3 z(f_3 z)]).$$

We drop outer quantifiers and lift over assumption $[\forall z. G z]$ to obtain

$$([\forall z. G z] \Rightarrow \bigwedge z. [\forall x. (G_3 z)x]) \Rightarrow \\ ([\forall z. G z] \Rightarrow \bigwedge z. [G_3 z(f_3 z)])$$

This rule will be applied in the next step.

Proof of $(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)$ (4)

$$\begin{array}{c}
 \vdots \\
 \hline
 \frac{(\llbracket \forall z.G\ z \rrbracket \Rightarrow \bigwedge z.\llbracket \forall x.(G_3\ z)x \rrbracket) \Rightarrow (\llbracket \forall z.G\ z \rrbracket \Rightarrow \bigwedge z.\llbracket G_3\ z(f_3\ z) \rrbracket) \quad (\llbracket \forall z.G\ z \rrbracket \Rightarrow \bigwedge z.\llbracket G\ z \rrbracket) \Rightarrow \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket}{(\llbracket \forall z.G\ z \rrbracket \Rightarrow \bigwedge z.\llbracket \forall x.G\ x \rrbracket) \Rightarrow \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket} \text{res}
 \end{array}$$

The substitution θ is $[f_3 \leftarrow \lambda w.w, G_3 \leftarrow \lambda v w.G\ w]$.

We suppress conversion, assuming terms are in normal form.

What to do next?

Proof of $(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)$ (4)

$$\begin{array}{c}
 \vdots \\
 \hline
 \begin{array}{l}
 (\llbracket \forall z.G\ z \rrbracket \Rightarrow \bigwedge z. \llbracket \forall x.(G_3\ z)x \rrbracket) \Rightarrow \\
 (\llbracket \forall z.G\ z \rrbracket \Rightarrow \bigwedge z. \llbracket G_3\ z(f_3\ z) \rrbracket) \quad \quad \quad (\llbracket \forall z.G\ z \rrbracket \Rightarrow \bigwedge z. \llbracket G\ z \rrbracket) \Rightarrow \\
 \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket
 \end{array} \\
 \hline
 \text{res}
 \end{array}$$

$$\begin{array}{c}
 (\llbracket \forall z.G\ z \rrbracket \Rightarrow \bigwedge z. \llbracket \forall x.G\ x \rrbracket) \Rightarrow \\
 \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket
 \end{array}$$

The substitution θ is $[f_3 \leftarrow \lambda w.w, G_3 \leftarrow \lambda v w.G\ w]$.

We suppress conversion, assuming terms are in normal form.

What to do next? Since $z \notin FV(\forall x.G\ x)$, we can use the assumption axiom.

Modified Assumption Axiom

$$\begin{array}{c}
 \frac{[\forall z.G z] \Rightarrow [\forall x.G x] \quad [[\forall z.G z]]^1}{[\forall z.G z]} \Rightarrow -E \\
 \frac{[\forall z.G z]}{\bigwedge z. [\forall x.G x]} \wedge -I \\
 \frac{\bigwedge z. [\forall x.G x]}{[\forall z.G z] \Rightarrow \bigwedge z. [\forall x.G x]} \Rightarrow -I^1
 \end{array}$$

We used **conversion** to identify $\forall z.G z$ and $\forall x.G x$.
 This rule will be used in the next step.

Proof of $(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)$ (5)

$$\begin{array}{c}
 \vdots \qquad \qquad \qquad \vdots \\
 \hline
 \begin{array}{cc}
 \llbracket \forall z.G\ z \rrbracket \Rightarrow & (\llbracket \forall z.G\ z \rrbracket \Rightarrow \bigwedge z.\llbracket \forall x.G\ x \rrbracket) \Rightarrow \\
 \bigwedge z.\llbracket \forall x.G\ x \rrbracket & \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket
 \end{array} \\
 \hline
 \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket \quad \text{res}
 \end{array}$$

Done!

Remark on Step 2

Recall Step 2:

$$\begin{array}{c}
 \vdots \\
 \hline
 \begin{array}{ll}
 (\llbracket \forall z. G z \rrbracket \Rightarrow (\bigwedge x. \llbracket F_1 x \rrbracket)) & (\llbracket \forall z. G z \rrbracket \Rightarrow \llbracket \forall z. G z \vee H z \rrbracket) \\
 \Rightarrow (\llbracket \forall z. G z \rrbracket \Rightarrow \llbracket \forall x. F_1 x \rrbracket) & \Rightarrow \llbracket (\forall z. G z) \rightarrow (\forall z. G z \vee H z) \rrbracket
 \end{array} \\
 \hline
 \begin{array}{l}
 (\llbracket \forall z. G z \rrbracket \Rightarrow (\bigwedge z. \llbracket G z \vee H z \rrbracket)) \\
 \Rightarrow \llbracket (\forall z. G z) \rightarrow (\forall z. G z \vee H z) \rrbracket
 \end{array}
 \end{array}
 \quad \text{res}$$

One could have obtained $\bigwedge z. (\llbracket \forall z. G z \rrbracket \Rightarrow (\llbracket G z \vee H z \rrbracket))$ instead of $(\llbracket \forall z. G z \rrbracket \Rightarrow (\bigwedge z. \llbracket G z \vee H z \rrbracket))$ by **lifting \forall -/** in a different way.

Checking Side Conditions

To demonstrate how side conditions are checked, we show a proof attempt that **fails** due to a side condition.

Take $\exists y. \forall x. x = y$ in **FOL with equality**, so assume we have a **meta-axiom** for reflexivity:

$$\bigwedge z. \llbracket z = z \rrbracket \quad (refl)$$

Failed Proof Attempt of $\exists y. \forall x. x = y$

$$\begin{array}{c}
 (\wedge x. [F_2 x]) \\
 \Rightarrow [\forall x. F_2 x] \\
 \hline
 \begin{array}{c}
 [F_1 y_1] \Rightarrow [\exists y. \forall x. x = y] \Rightarrow \\
 [\exists x. F_1 x] \quad [\exists y. \forall x. x = y] \\
 \hline
 [\forall x. x = y_1] \Rightarrow [\exists y. \forall x. x = y] \quad \text{res} \\
 \hline
 (\wedge x. [x = y_1]) \Rightarrow [\exists y. \forall x. x = y] \quad \text{res}
 \end{array}
 \end{array}$$

Substitution? $[F_1 \leftarrow \quad, F_2 \leftarrow \quad]$.

Failed Proof Attempt of $\exists y. \forall x. x = y$

$$\begin{array}{c}
 (\wedge x. [F_2 x]) \\
 \Rightarrow [\forall x. F_2 x] \\
 \hline
 \begin{array}{c}
 [F_1 y_1] \Rightarrow [\exists y. \forall x. x = y] \Rightarrow \\
 [\exists x. F_1 x] \quad [\exists y. \forall x. x = y] \\
 \hline
 [\forall x. x = y_1] \Rightarrow [\exists y. \forall x. x = y] \quad \text{res} \\
 \hline
 (\wedge x. [x = y_1]) \Rightarrow [\exists y. \forall x. x = y] \quad \text{res}
 \end{array}
 \end{array}$$

Substitution? $[F_1 \leftarrow \lambda v. \forall x. x = v, F_2 \leftarrow \lambda v. v = y_1]$.

What to do next?

Failed Proof Attempt of $\exists y. \forall x. x = y$

$$\begin{array}{c}
 \frac{
 \begin{array}{c}
 (\bigwedge x. [F_2 x]) \\
 \Rightarrow [\forall x. F_2 x]
 \end{array}
 \quad
 \frac{
 \frac{
 [F_1 y_1] \Rightarrow [\exists y. \forall x. x = y] \Rightarrow \\
 [\exists x. F_1 x] \quad [\exists y. \forall x. x = y]
 }{
 [\forall x. x = y_1] \Rightarrow [\exists y. \forall x. x = y]
 } \text{res}
 }{
 (\bigwedge x. [x = y_1]) \Rightarrow [\exists y. \forall x. x = y]
 } \text{res}
 \end{array}$$

Substitution? $[F_1 \leftarrow \lambda v. \forall x. x = v, F_2 \leftarrow \lambda v. v = y_1]$.

What to do next? Resolution with *refl* lifted over parameter x : $\bigwedge x. [g_3 x = g_3 x]$. But $\bigwedge x. [x = y_1]$ and $\bigwedge x. [g_3 x = g_3 x]$ are not unifiable. Proof fails!

Free Variables in Goals

The resolution rule can be generalized to allow for instantiation of variables **in goals**:

$$\frac{[\phi_1, \dots, \phi_m] \Rightarrow \phi \quad [\psi_1, \dots, \psi_n] \Rightarrow \psi}{([\psi_1, \dots, \psi_{i-1}, \phi_1, \dots, \phi_m, \psi_{i+1}, \dots, \psi_n] \Rightarrow \psi)\theta} \text{res}$$

where $\phi\theta \equiv \psi_i\theta$.

But then we must distinguish the status of the free variables. Denote the **universal closure** of ψ by $\bigwedge _.\psi$. Then . . .

Instantiation of the Initial Goal

Previously, when we proved ψ we in fact proved $\bigwedge _.\psi$.

$$\frac{\begin{array}{c} \vdots \qquad \vdots \\ \hline \psi \end{array} \quad \frac{\psi \Rightarrow \psi}{\hline}}{\psi}$$

Instantiation of the Initial Goal

Previously, when we proved ψ we in fact proved $\bigwedge _.\psi$.

$$\begin{array}{c}
 \frac{\bigwedge _.\psi \Rightarrow \psi}{\psi \Rightarrow \psi} \quad \Lambda\text{-}E \\
 \vdots \qquad \vdots \\
 \hline
 \psi \\
 \hline
 \bigwedge _.\psi \quad \Lambda\text{-}I
 \end{array}$$

Instantiation of the Initial Goal

Previously, when we proved ψ we in fact proved $\bigwedge _.\psi$.

Now, allowing for instantiation of ψ , we in fact prove $\bigwedge _.\psi\theta$.

$$\begin{array}{c}
 \frac{\bigwedge _.\psi \Rightarrow \psi}{\psi \Rightarrow \psi} \quad \Lambda\text{-}E \\
 \hline
 \frac{\vdots \quad \vdots}{\psi\theta} \\
 \hline
 \frac{\psi\theta}{\bigwedge _.\psi\theta} \quad \Lambda\text{-}I
 \end{array}$$

Instantiation of the Initial Goal

Previously, when we proved ψ we in fact proved $\bigwedge _.\psi$.

Now, allowing for instantiation of ψ , we in fact prove $\bigwedge _.\psi\theta$.

$$\begin{array}{c}
 \frac{\bigwedge _.\psi \Rightarrow \psi}{\psi \Rightarrow \psi} \quad \Lambda\text{-}E \\
 \vdots \qquad \vdots \\
 \hline
 \psi\theta \\
 \hline
 \bigwedge _.\psi\theta \quad \Lambda\text{-}I
 \end{array}$$

This may not be **what we want**.

Problem: more unifiers, hence **bigger search space**.

Two Kinds of Free Variables

In Isabelle, control over instantiation is given by having two kinds of free variables:

- ordinary variables **must not** become instantiated;
- **metavariables** (**unknowns**, **schematic variables**) may become instantiated.

In **goals** we can have both kinds, in **rules** we have metavariables. **Try it out in Isabelle!**

Once a theorem is proven, any free variables will be **made metavariables**, and the reading is **as for rules**: The theorem is implicitly universally quantified over the free variables.

Conclusion on Isabelle's Metalogic

The logic \mathcal{M} and its proof system are **small**.

What makes \mathcal{M} powerful enough to encode a large variety of object logics?

- The **λ -calculus** is very powerful for expressing syntax and syntactic manipulations (\rightarrow substitution). \mathcal{M} must be extended by appropriate **signature** for an object logic.
- Rules of the object logic can be encoded and **added to \mathcal{M}** as axioms.

Conclusion (2)

General principles of proof building (e.g. resolution, proving by assumption, side condition checking) are **not** something that must be justified by complicated (and thus error-prone) explanations in natural language — they are formal derivations in the metalogic.

This has two big **advantages**: **shared support** and **high degree of confidence**. ►

More Detailed Explanations

Formalising Propositional Logic

We would formalise the language and the proof system as we did in the [first lecture](#). Any proofs of soundness and completeness or other meta-properties should be rigorous, but they still resort to natural language.

Σ Contains these Symbols

Σ contains \Rightarrow , \equiv and \wedge , but in addition, Σ may specify other symbols.

Alternative: Polymorphism

Alternatively, we could define that

- $\equiv_\alpha : \alpha \rightarrow \alpha \rightarrow bool \in \Sigma$, and
- $\bigwedge_\alpha : (\alpha \rightarrow bool) \rightarrow bool \in \Sigma$,

where α is a **type variable**.

The names of \Rightarrow , \equiv , and \bigwedge

\Rightarrow is called **meta-implication**, \equiv is called **meta-equality**, and \bigwedge is called **meta-universal-quantification**.

The Judgements

We define our proof system for \mathcal{M} using **natural deduction**.

The **judgements** are **formulae**, i.e., term of type *bool*. This means that a node ϕ in a derivation tree, as in

$$\frac{\dots}{\phi} \dots$$

must be a term of type *bool*. It cannot be a **derivability judgement** or **type judgement** or a term of type, say $bool \rightarrow bool$.

Extensionality

Extensionality is the rule

$$\frac{f\ x \equiv g\ x}{f \equiv g}$$

where the side condition is that x must not be free in f or g or any assumption on which the proof of $f\ x \equiv g\ x$ depend. It is equivalent to the η -axiom [HS86, pages 72-74].

Recall that we have used the notion of extensionality before, for sets. The idea is the same here.

Shorthand for Writing Signatures

We write

$$\langle \perp : o, \\ \wedge, \vee, \rightarrow : o \rightarrow o \rightarrow o, \\ \forall, \exists : (i \rightarrow o) \rightarrow o, \\ \text{true} : o \rightarrow \text{bool} \rangle$$

as shorthand for

$$\langle \perp : o, \\ \wedge : o \rightarrow o \rightarrow o, \\ \vee : o \rightarrow o \rightarrow o, \\ \rightarrow : o \rightarrow o \rightarrow o, \\ \forall : (i \rightarrow o) \rightarrow o, \\ \exists : (i \rightarrow o) \rightarrow o \\ \text{true} : o \rightarrow \text{bool} \rangle$$

Two Types for Truth Values

So we have truth values in the metalogic (type *bool*) and in the object logic (type *o*). To distinguish them clearly there are two different types for them.

The constant *true*

So we have truth values in the metalogic (type *bool*) and in the object logic (type *o*).

Paulson [Pau89] says: “the meta-formula $\llbracket A \rrbracket$ abbreviates *true* *A* and means that *A* is true”. More precisely, we can say that $\llbracket A \rrbracket$ is a meta-formula that may or may not be derivable in \mathcal{M}_L , and that this should reflect *derivability of *A* in *L**.

In the file `IFOL.thy` in your *Isabelle distribution*, you find

```
Trueprop      :: "o => prop"
```

`Trueprop` corresponds to *true*.

$$\psi \Rightarrow \psi$$

We have seen this **before** as a proof in propositional logic.

$$\frac{[\psi]^1}{\psi \rightarrow \psi} \Rightarrow\text{-I}^1$$

Resolution Using \vee

You may have seen the following formulation of the resolution rule:

$$\frac{A_1 \vee \dots \vee A_n \quad B_1 \vee \dots \vee B_m}{(A_1 \vee \dots \vee A_{i-1}, A_{i+1} \vee \dots \vee A_n \vee B_1 \vee \dots \vee B_{j-i}, B_{j+1} \vee \dots \vee B_m)\theta}$$

where either $A_i\theta = \neg B_j\theta$ or $\neg A_i\theta = B_j\theta$.

You can see the correspondence to the rule given here by recalling that in **first-order logic**, $\phi_1 \rightarrow \dots \rightarrow \phi_m \rightarrow \phi$ is equivalent to

$\phi_1 \wedge \dots \wedge \phi_m \rightarrow \phi$, which is in turn equivalent to $\neg\phi_1 \vee \dots \vee \neg\phi_m \vee \phi$.

You may still be wondering though why in the **rule res**, we only allow instantiation of $[\phi_1, \dots, \phi_m] \Rightarrow \phi$. This restriction will in fact be lifted **later**.

Why Is $\Rightarrow\text{-}E$ Applicable?

Recall that $\phi\theta \equiv \psi_i$.

Schematic Resolution Rule

The schematic form of the **resolution rule** is:

$$\frac{[\phi_1, \dots, \phi_m] \Rightarrow \phi \quad [\psi_1, \dots, \psi_n] \Rightarrow \psi}{[\psi_1, \dots, \psi_{i-1}, \phi_1\theta, \dots, \phi_m\theta, \psi_{i+1}, \dots, \psi_n] \Rightarrow \psi} \text{res}$$

where $\phi\theta \equiv \psi$.

We will work with this schematic form, but remember: if necessary, you could construct an actual derivation in \mathcal{M} .

In this schematic form, it is always assumed that the free variables in

$[\phi_1, \dots, \phi_m] \Rightarrow \phi$ are **fresh**,

i.e. $FV([\phi_1, \dots, \phi_m] \Rightarrow \phi) \cap FV([\psi_1, \dots, \psi_n] \Rightarrow \psi) = \emptyset$.

This assumption may be justified considering the **formal derivation of the resolution rule**. Suppose that the free variables in $[\phi_1, \dots, \phi_m] \Rightarrow \phi$ are not all fresh, and consider $\bigwedge x'_1 \dots x'_k. [\phi'_1, \dots, \phi'_m] \Rightarrow \phi'$, obtained from

$\bigwedge x_1 \dots x_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi$ by replacing each x_i with x'_i , where the x'_i are fresh.

It is easy to see that in the formal derivation of the resolution rule, one can replace

$$\frac{\bigwedge x_1 \dots x_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi}{[\phi_1\theta, \dots, \phi_m\theta] \Rightarrow \phi\theta} \quad \wedge\text{-}E$$

with

$$\frac{\bigwedge x'_1 \dots x'_k. [\phi'_1, \dots, \phi'_m] \Rightarrow \phi'}{[\phi_1\theta, \dots, \phi_m\theta] \Rightarrow \phi\theta} \quad \wedge\text{-}E$$

Therefore we can assume without loss of generality that the free variables in $[\phi_1, \dots, \phi_m] \Rightarrow \phi$ are fresh.

The next question is: why do we want fresh variables? Maybe this is clear intuitively: A rule is always meant to be schematic and the choice of variables names in a rule should be irrelevant. More concretely, one

may say that if one does not rename the variables in a rule and hence there is some variable, say A , that occurs in the current subgoal, then resolution may lead to a subgoal containing occurrences of A originating from the goal and others originating from the rule, and these are inadvertently identified, leading to a proof state that is more instantiated than it should be.

A Hint

On the one hand, we want to resolve

$$([A \wedge B] \Rightarrow [C \rightarrow A \wedge C]) \Rightarrow [A \wedge B \rightarrow (C \rightarrow A \wedge C)],$$

i.e., we have to match $([A \wedge B] \Rightarrow [C \rightarrow A \wedge C])$ against the conclusion of some meta-axiom.

On the other hand, think what Isabelle would display in this situation. The (only) subgoal would be

$$1. A \wedge B \Rightarrow C \rightarrow A \wedge C,$$

so we have to show $C \rightarrow A \wedge C$ (using assumption $A \wedge B$). So you should look at $C \rightarrow A \wedge C$ to guess which meta-axiom should be used now.

Resolution not Applicable

In our current situation, Isabelle would display:

Level 1(1 subgoal)
 $A \wedge B \rightarrow (C \rightarrow A \wedge C)$
1. $A \wedge B \Longrightarrow C \rightarrow A \wedge C$

From your experience with Isabelle, it is clear that since the top-level symbol in $C \rightarrow A \wedge C$ is \rightarrow , you would use \rightarrow -I.

But look at the **resolution rule** again. We would take a fresh instance of \rightarrow -I, say $([A_2] \Rightarrow [B_2]) \Rightarrow [A_2 \rightarrow B_2]$. The problem is that $[A_2 \rightarrow B_2]$ is not unifiable with $[A \wedge B] \Rightarrow [C \rightarrow A \wedge C]$, and so *res* is not applicable.

The Lifted \rightarrow -/

$$(\llbracket A \wedge B \rrbracket \Rightarrow (\llbracket A_2 \rrbracket \Rightarrow \llbracket B_2 \rrbracket)) \Rightarrow (\llbracket A \wedge B \rrbracket \Rightarrow \llbracket A_2 \rightarrow B_2 \rrbracket)$$

is the \rightarrow -/rule (meta-axiom) lifted over the assumption $A \wedge B$.

The Lifted \wedge -I

$$(\Omega \Rightarrow \llbracket A_3 \rrbracket) \Rightarrow (\Omega \Rightarrow \llbracket B_3 \rrbracket) \Rightarrow (\Omega \Rightarrow \llbracket A_3 \wedge B_3 \rrbracket)$$

is the \wedge -I-rule (meta-axiom) lifted over the assumption list Ω . Recall that Ω was an abbreviation for $\llbracket A \wedge B \rrbracket, \llbracket C \rrbracket$, but this is obviously irrelevant for the process of lifting.

Why Is the Assumption Axiom Schematic?

The assumption axiom

$$\frac{}{[\phi_1, \dots, \phi_m] \Rightarrow \phi_i} \text{assum}$$

is schematic in two senses:

- the Greek letters could stand for arbitrary formulae;
- just like for resolution rule, we don't even know how many formulae are involved (m, i could be any natural numbers).

However, one could also write the axiom as

$$\frac{}{[A_1, \dots, A_m] \Rightarrow A_i} \text{assum}$$

where the A 's are variables (of type *bool*) and instantiate it later when it is used in some resolution step.

Discharging Is Optional

Recall here that the rule $\Rightarrow\text{-I}$, just like $\rightarrow\text{-I}$, allows you to discharge **zero or more** assumptions. In the present derivation, we discharge the assumption ϕ_i at some point but we do not discharge any other assumptions.

Using the more Generic Assumption Axiom

As explained [previously](#), we could use a more generic variant of the assumption axiom, in that we have variables in it that may become instantiated upon resolution. Note however that the variant is still **specific** in the sense that $m = 2$. The proof fragment would then look as follows:

$$\frac{[A_1, A_2] \Rightarrow A_2 \quad \frac{\quad \quad \quad \vdots}{(\Omega \Rightarrow [A]) \Rightarrow (\Omega \Rightarrow [C]) \Rightarrow \psi}}{(\Omega \Rightarrow [A]) \Rightarrow \psi} \text{res}$$

where $\theta = \{A_1 \leftarrow [A \wedge B], A_2 \leftarrow [C]\}$.

Turning Trees Upside-Down?

Intuitively, as far as the order in which the **object rules**, resp. **meta-axioms**, are applied, the proof in \mathcal{M}_{Prop} is the proof in $Prop$ turned upside-down.

However, this may seem suspicious for two reasons:

- In derivation trees, the direction of implication (forgetting about whether it is meta- or object implication) is “downwards”: whatever is above implies whatever is below. So it seems strange that this order should be reversed just because we go from the object to the meta-level.
- In general, a derivation tree in the object level is a **proper** tree, i.e., there are nodes where it branches. So what sense does it make to “turn it upside-down”? The result would not be any tree at all.

These points will now be **addressed**.

Interleaving Proofs

If one pictures the object level proof and how it is modeled in \mathcal{M}_{Prop} , one intuitive way of thinking of it is as follows: Each rule application in the object level proof must also be performed at the meta-level. Now, starting at the root of the object level proof, we may do any rule application that is the **child** of a rule application we have done previously. Take for example the following object level proof:

$$\begin{array}{c}
 \frac{[A \wedge (B \wedge C)]^1}{A} \wedge\text{-}EL^3 \qquad \frac{[A \wedge (B \wedge C)]^1}{B \wedge C} \wedge\text{-}ER^5 \\
 \qquad \qquad \qquad \frac{B \wedge C}{C} \wedge\text{-}ER^4 \\
 \frac{A}{A \wedge C} \wedge\text{-}I^2 \\
 \frac{A \wedge C}{A \wedge (B \wedge C) \rightarrow A \wedge C} \rightarrow\text{-}I^1
 \end{array}$$

Then in the meta-proof, the meta-axioms might be applied in the

$\rightarrow-I^1, \wedge-I^2, \wedge-ER^4, \wedge-ER^5, \wedge-EL^3$, or

following orders: $\rightarrow-I^1, \wedge-I^2, \wedge-EL^3, \wedge-ER^4, \wedge-ER^5$, or

$\rightarrow-I^1, \wedge-I^2, \wedge-ER^4, \wedge-EL^3, \wedge-ER^5$.

But this is not new to you: In Isabelle, you are always free to choose the subgoal that you want to work on next, and so you can interleave the proofs of the different subgoals.

A Different Way of Lifting \forall -/

In our proof, we lifted \forall -/ over assumption $\llbracket \forall z. G \ z \rrbracket$ as follows:

$$(\llbracket \forall z. G \ z \rrbracket \Rightarrow (\bigwedge x. \llbracket F_1 \ x \rrbracket)) \Rightarrow (\llbracket \forall z. G \ z \rrbracket \Rightarrow \llbracket \forall x. F_1 \ x \rrbracket)$$

It would have been possible to derive (formally, in \mathcal{M}) the following rule instead:

$$(\bigwedge x. \llbracket \forall z. G \ z \rrbracket \Rightarrow \llbracket F_1 \ x \rrbracket) \Rightarrow (\llbracket \forall z. G \ z \rrbracket \Rightarrow \llbracket \forall x. F_1 \ x \rrbracket)$$

This is essentially so since $z \notin FV \llbracket \forall z. G \ z \rrbracket$. If we had done it like that,

step 2 would have looked as follows

$$\begin{array}{c}
 \vdots \\
 \hline
 \begin{array}{cc}
 (\bigwedge x. [\forall z. G z] \Rightarrow [F_1 x]) & ([\forall z. G z] \Rightarrow [\forall z. G z \vee H z]) \\
 \Rightarrow ([\forall z. G z] \Rightarrow [\forall x. F_1 x]) & \Rightarrow [(\forall z. G z) \rightarrow (\forall z. G z \vee H z)]
 \end{array} \\
 \hline
 \begin{array}{c}
 (\bigwedge z. [\forall z. G z] \Rightarrow [G z \vee H z]) \\
 \Rightarrow [(\forall z. G z) \rightarrow (\forall z. G z \vee H z)]
 \end{array}
 \end{array}
 \text{res}$$

The rest of the proof would then have looked slightly differently due to the different scope of the \bigwedge . For example, it would have been necessary to lift \forall - IL over assumptions **before** lifting it over parameters.

In fact, if we denote a vector of variables by overlining, then we can

derive the following rule for lifting over assumptions:

$$\frac{[(\bigwedge \bar{x}_1.\phi_1), \dots, (\bigwedge \bar{x}_m.\phi_m)] \Rightarrow \phi}{[(\bigwedge \bar{x}_1.\Psi \Rightarrow \phi_1), \dots, (\bigwedge \bar{x}_1.\Psi \Rightarrow \phi_m)] \Rightarrow (\Psi \Rightarrow \phi)}$$

where $\bar{x}_1, \dots, \bar{x}_m \notin FV(\Psi)$. Compare this to rule *a-lift*. Using the more complicated rule, where the assumption list Ψ is pulled into the scope of \bigwedge 's surrounding each rule premise ϕ_i , would probably have made the presentation here somewhat more complicated. On the other hand, this is indeed what happens in Isabelle (try to do the proof of $(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)$ in Isabelle).

Lifting *refl*

Note that *lifting refl*

$$\bigwedge z. \llbracket z = z \rrbracket$$

over x gives

$$\bigwedge g_3. \bigwedge x. \llbracket g_3 x = g_3 x \rrbracket.$$

Here the variable z in *refl* was replaced by the variable g_3 that depends on x . However, we *drop* the outer quantification $\bigwedge g_3$. In this particular case, $\bigwedge x$ is also an outer quantification, but we keep it, since obtaining this quantification was the very purpose of lifting (recall that lifting is done to achieve *unifiability*).

Unifying $\bigwedge x.[x = y_1]$ and $\bigwedge x.[g_3 x = g_3 x]$

Recall that $\bigwedge x.\phi$ is syntactic sugar for $\bigwedge x.(\lambda x.\phi)$.

So we have to unify $\lambda x.[x = y_1]$ and $\lambda x.[g_3 x = g_3 x]$.

It turns out that this task can be decomposed into having to unify $\lambda x.x$ and $\lambda x.g_3 x$ on the one hand, and $\lambda x.y_1$ and $\lambda x.g_3 x$ on the other hand. Unification of $\lambda x.x$ and $\lambda x.g_3 x$ forces g_3 to be $\lambda x.x$, so we are left with having to unify $\lambda x.y_1$ and $\lambda x.x$. But these terms are not unifiable!

This was just a semi-formal argument that $\bigwedge x.[x = y_1]$ and $\bigwedge x.[g_3 x = g_3 x]$ are not unifiable, but it gives you the idea.

The Universal Closure

The **universal closure** of a meta-formula ψ is the formula $\bigwedge x_1 \dots x_n. \psi$ where $FV(\psi) = \{x_1 \dots x_n\}$.

As might be expected, the same concept is also used for **FOL** formulae where it is defined in analogy using \forall instead of \bigwedge .

Proving what We Want

Suppose we want to prove $((A \rightarrow B) \rightarrow A) \rightarrow A$. If we allow for instantiation of the free variables A and B , we could easily end up proving $((A \rightarrow A) \rightarrow A) \rightarrow A$. This is probably not what we want. In fact the proof has little to do with the proof of $((A \rightarrow B) \rightarrow A) \rightarrow A$ that is **schematic in A and B** .

In terms of \mathcal{M}_{Prop} , we want to prove $\bigwedge AB. \llbracket ((A \rightarrow B) \rightarrow A) \rightarrow A \rrbracket$. Recall that $((A \rightarrow B) \rightarrow A) \rightarrow A$ is **Peirce's law**.

Too many Unifiers

The more free variables in the goal we allow Isabelle to instantiate, the more unifiers there are. This may increase the search space to the extent of making it impossible to find a proof.

Test the Difference

To understand the difference, try proving $A \wedge B \rightarrow P$ and $A \wedge B \rightarrow ?P$ in Isabelle. The first won't succeed while the second may succeed in various ways.

See how Variables Are Turned into Metavariables

Prove $A \wedge B \rightarrow ?P$ in Isabelle and save (qed) it as a theorem and then have a look at the theorem.

The Inference Rules of \mathcal{M} Are not Enough

In some course on propositional logic, you may have learned that the connective \rightarrow is not really necessary since $A \rightarrow B$ is equivalent to $\neg A \vee B$. Likewise, we considered $\neg A$ as **syntactic sugar** for $A \rightarrow \perp$. Therefore, when we introduce a logic \mathcal{M} that is so extremely simple as far as the number of **logical symbols** is concerned (just $\Rightarrow, \equiv, \wedge$), one might think that the idea is that all the other logical symbols one usually needs are just syntactic sugar. **This is not the case!**

To encode **propositional logic** or **FOL** in \mathcal{M} , we must add their rules as axioms.

Later, we will be working with a logic just slightly richer than \mathcal{M} but still quite simple, and there the idea is indeed that all the other logical symbols one usually needs are just syntactic sugar.

HOL: Foundations

Overview

HOL is expressive foundation for

- **Mathematics**: analysis, algebra, ...
- **Computer science**: program correctness, hardware verification, ...

Overview

HOL is expressive foundation for

- **Mathematics**: analysis, algebra, ...
- **Computer science**: program correctness, hardware verification, ...

HOL is very similar to \mathcal{M} , but it “is” an object logic!

- Reasoning is classical.
- Still important: **modeling** of problems/domains (now within HOL).
- Still important: **deriving** relevant reasoning principles.

Isabelle/HOL vs. Alternatives

We will use Isabelle/HOL.

- Could forgo the use of a metalogic and employ alternatives, e.g., HOL system or PVS, or constructive provers such as Coq, Nuprl.
- Choice depends on culture and application.

Safety through Strength

Safety via **conservative** (definitional) **extensions**:

- Small kernel of constants and rules;
- extend theory with new constants and types defined using existing ones;
- **derive** properties/theorems.

Contrast with:

- Weak logics (e.g., propositional logic): can't define much;
- **axiomatic extensions**: can lead to inconsistency.

Bertrand Russel once likened the advantages of postulation over definition to the advantages of theft over honest toil!

Set Theory as Alternative?

Set theory is the logician's choice as basis for modern mathematics.

- ZFC (in Isabelle): impressive applications!
- Bernays-Gödel: used for resolution since finitely axiomatizable.

Set theories (both) distinguish between sets and classes.

- Consistency maintained as some collections are “too big” to be sets, e.g., class of all sets V is not a set.
- A class cannot belong to another class (let alone a set)!

Finally: We Choose HOL!

HOL developed by [Chu40, Hen50] and rediscovered by [And02, GM93].

- **Rationale:** one usually works with typed entities.
- Reasoning is then easier with support for types.
HOL is **classical** logic based on λ^{\rightarrow} .
- Isabelle/HOL also supports “mod cons” like **polymorphism** and **type classes**!

HOL is weaker than ZF set theory, but for most applications this does not matter. If you prefer ML to Lisp, you will probably prefer HOL to ZF. (Paulson)

What Does Higher-Order Mean?

“Type” order	Logic order	
		Example
0		$A \wedge B \rightarrow B \wedge A$
+ quantification		
1	1	$\forall x, y. R(x, y) \rightarrow R(y, x)$
+ quantification	2	$False \equiv \forall P. P$ $P \wedge Q \equiv \forall R. (P \rightarrow Q \rightarrow R)$
2	3	
+ quantification	4	$\forall X. (X(R, S) \leftrightarrow (\forall x. R(x) \rightarrow S(x)))$ $\rightarrow \dots X(R', S') \dots (\equiv \textit{subrel}(R, S))$

HOL = Union of All Finite Orders

ω -order logic, also called finite-type theory or higher-order logic (HOL), includes logics of all finite orders.

Syntax

Syntactically, HOL is a polymorphic (although not necessarily) variant of λ^{\rightarrow} with certain default **types** and **constants**.

Default constants can be called **logical symbols**.

Types (Review)

Given a set of type constructors, say

$\mathcal{B} = \{bool, _ \rightarrow _, ind, _ \times _, _ list, _ set, \dots\}$, polymorphic types are defined by $\tau ::= \alpha \mid (\tau, \dots, \tau) T$, where α is type variable.

- *bool* is also called *o* in literature [Chu40, And02], but avoid confusion.
- *bool* and \rightarrow always present in HOL; *ind* will also play a special role; other type constructors may be defined.
- Note polymorphism!

Terms

Reminder: $e ::= x \mid c \mid (ee) \mid (\lambda x^\tau. e)$

Typing rules as in polymorphic λ -calculus, with Σ defining and typing constants.

Terms of type *bool* are called (well-formed) formulae.

In HOL, Σ always includes:

$True, False : bool$

$= : \alpha \rightarrow \alpha \rightarrow bool$ (polymorphic, or set)

$\rightarrow : bool \rightarrow bool \rightarrow bool$

$\epsilon : (\alpha \rightarrow bool) \rightarrow \alpha$ (in Isabelle: Eps or SOME)

True and *False* are intuitive but not strictly necessary.

Semantics

Intuitively: many-sorted semantics + functions

- FOL: structure is domain and functions/relations.

$$\mathcal{I} = \langle \mathcal{D}, f_1^{\mathcal{I}}, \dots, f_k^{\mathcal{I}}, r_1^{\mathcal{I}}, \dots, r_j^{\mathcal{I}} \rangle$$

Semantics

Intuitively: many-sorted semantics + functions

- FOL: structure is domain and functions/relations.

Many-sorted FOL: domains are sort-indexed

$$\mathcal{I} = \langle \mathcal{D}_1, \dots, \mathcal{D}_n, f_1^{\mathcal{I}}, \dots, f_k^{\mathcal{I}}, r_1^{\mathcal{I}}, \dots, r_j^{\mathcal{I}} \rangle$$

Semantics

Intuitively: many-sorted semantics + functions

- FOL: structure is domain and functions/relations.
Many-sorted FOL: domains are sort-indexed

$$\mathcal{I} = \langle \mathcal{D}_1, \dots, \mathcal{D}_n, f_1^{\mathcal{I}}, \dots, f_k^{\mathcal{I}}, r_1^{\mathcal{I}}, \dots, r_j^{\mathcal{I}} \rangle$$

(if no relations then we have a heterogeneous algebra).

- HOL extends idea: \mathcal{D} indexed by (infinitely many) types.
- Complications due to polymorphism [GM93].
- We only give a monomorphic variant of semantics here!

Model Based on Universe of Sets \mathcal{U}

\mathcal{U} is a collection of sets (**domains**), fulfilling **closure conditions**:

Inhab: Each $X \in \mathcal{U}$ is a nonempty set

Sub: If $X \in \mathcal{U}$ und $Y \neq \emptyset \subseteq X$, then $Y \in \mathcal{U}$

Prod: If $X, Y \in \mathcal{U}$ then $X \times Y \in \mathcal{U}$.

Pow: If $X \in \mathcal{U}$ then $\wp(X) = \{Y \mid Y \subseteq X\} \in \mathcal{U}$

Infty: \mathcal{U} contains a distinguished **infinite set** I

Choice: There is a function $ch \in \prod_{X \in \mathcal{U}} X$.

Prod: Encoding $X \times Y$

$X \times Y$ is the Cartesian product.

One can actually encode a tuple (x, y) without explicitly postulating the “existence of tuples”. E.g.:

$$(x, y) \equiv \{\{x\}, \{x, y\}\}$$

Choice: Picking a Member

The function ch takes a set $X \in \mathcal{U}$ as argument and returns a **member** of X .

We hence write $ch \in \Pi_{X \in \mathcal{U}}.X$, i.e., ch is of dependent type. Essentially, the constant ϵ will be interpreted as ch , but you will see the technical details **later**.

Function Space in \mathcal{U}

Define set $X \rightarrow Y$ as (graph of) functions from X to Y .

- For nonempty Y , this set is nonempty and is a subset of $\wp(X \times Y)$.
- From closure conditions: $X, Y \in \mathcal{U}$ then $X \rightarrow Y \in \mathcal{U}$.

Distinguished Sets

From

Infty: \mathcal{U} contains a distinguished infinite set I

Sub: If $X \in \mathcal{U}$ und $Y \neq \emptyset \subseteq X$, then $Y \in \mathcal{U}$

it follows that there is (at least one) set

Unit: A distinguished 1-element set $\{1\}$

Bool: A distinguished 2-element set $\{T, F\}$.

Frames

For semantics, we neglect polymorphism. τ and σ range over types.

A **frame** is a collection $\{\mathcal{D}_\tau\}_\tau$ of non-empty sets (**domains**) $\mathcal{D}_\tau \in \mathcal{U}$, one for each type τ , where:

- $\mathcal{D}_{bool} = \{T, F\}$;
- $\mathcal{D}_{\tau \rightarrow \sigma} \subseteq \mathcal{D}_\tau \rightarrow \mathcal{D}_\sigma$, i.e., **some** collection of functions from \mathcal{D}_τ to \mathcal{D}_σ .
- $\mathcal{D}_{ind} = I$.

Note: for fundamental reasons discussed **later**, one cannot simply define $\mathcal{D}_{\tau \rightarrow \sigma} = \mathcal{D}_\tau \rightarrow \mathcal{D}_\sigma$ at this stage.

Interpretations

An **interpretation** $\langle \{\mathcal{D}_\tau\}_\tau, \mathcal{J} \rangle$ is a frame $\{\mathcal{D}_\tau\}_\tau$ and a denotation function \mathcal{J} mapping each constant of type τ to an element of \mathcal{D}_τ where:

- $\mathcal{J}(True) = T$ and $\mathcal{J}(False) = F$;
- $\mathcal{J}(=_{\tau \rightarrow \tau \rightarrow bool})$ is **identity** on \mathcal{D}_τ ;
- $\mathcal{J}(\rightarrow)$ is **implication** function over \mathcal{D}_{bool} . For $b, b' \in \{T, F\}$,

$$\mathcal{J}(\rightarrow)(b, b') = \begin{cases} F & \text{if } b = T \text{ and } b' = F \\ T & \text{otherwise} \end{cases}$$

Interpretations (Cont.)

- $\mathcal{J}(\epsilon_{(\tau \rightarrow \text{bool}) \rightarrow \tau})$ is defined by (for $f \in (\mathcal{D}_\tau \rightarrow \mathcal{D}_{\text{bool}})$):

$$\mathcal{J}(\epsilon_{(\tau \rightarrow \text{bool}) \rightarrow \tau})(f) = \begin{cases} ch(f^{-1}(\{T\})) & \text{if } f^{-1}(\{T\}) \neq \emptyset \\ ch(\mathcal{D}_\tau) & \text{otherwise} \end{cases}$$

Note: If a frame $\{\mathcal{D}_\tau\}_\tau$ does not contain all of the functions used above, then that frame cannot belong to any interpretation.

Alternative: define interpretation as a **structure**.

The Value of Terms (Naive)

In analogy to FOL, given an interpretation $\mathfrak{M} = \langle \{\mathcal{D}_\tau\}_\tau, \mathcal{J} \rangle$ and a type-indexed collection of assignments $A = \{A_\tau\}_\tau$, define $\mathcal{V}_A^\mathfrak{M}$ such that $\mathcal{V}_A^\mathfrak{M}(t_\rho) \in \mathcal{D}_\rho$ for all t , as follows:

1. $\mathcal{V}_A^\mathfrak{M}(x_\tau) = A(x_\tau)$;
2. $\mathcal{V}_A^\mathfrak{M}(c) = \mathcal{J}(c)$ for c a constant;
3. $\mathcal{V}_A^\mathfrak{M}(s_{\tau \rightarrow \sigma} t_\tau) = (\mathcal{V}_A^\mathfrak{M}(s))(\mathcal{V}_A^\mathfrak{M}(t))$, i.e., the value of the function $\mathcal{V}_A^\mathfrak{M}(s)$ at the argument $\mathcal{V}_A^\mathfrak{M}(t)$;
4. $\mathcal{V}_A^\mathfrak{M}(\lambda x^\tau. t_\sigma) =$ the function from \mathcal{D}_τ into \mathcal{D}_σ whose value for each $e \in \mathcal{D}_\tau$ is $\mathcal{V}_{A[x \leftarrow e]}^\mathfrak{M}(t)$.

What is the problem?

The Value of Terms (Naive)

In analogy to FOL, given an interpretation $\mathfrak{M} = \langle \{\mathcal{D}_\tau\}_\tau, \mathcal{J} \rangle$ and a type-indexed collection of assignments $A = \{A_\tau\}_\tau$, define $\mathcal{V}_A^\mathfrak{M}$ such that $\mathcal{V}_A^\mathfrak{M}(t_\rho) \in \mathcal{D}_\rho$ for all t , as follows:

1. $\mathcal{V}_A^\mathfrak{M}(x_\tau) = A(x_\tau)$;
2. $\mathcal{V}_A^\mathfrak{M}(c) = \mathcal{J}(c)$ for c a constant;
3. $\mathcal{V}_A^\mathfrak{M}(s_{\tau \rightarrow \sigma} t_\tau) = (\mathcal{V}_A^\mathfrak{M}(s))(\mathcal{V}_A^\mathfrak{M}(t))$, i.e., the value of the function $\mathcal{V}_A^\mathfrak{M}(s)$ at the argument $\mathcal{V}_A^\mathfrak{M}(t)$;
4. $\mathcal{V}_A^\mathfrak{M}(\lambda x^\tau. t_\sigma) =$ the function from \mathcal{D}_τ into \mathcal{D}_σ whose value for each $e \in \mathcal{D}_\tau$ is $\mathcal{V}_{A[x \leftarrow e]}^\mathfrak{M}(t)$.

What is the problem? Condition 4!

Condition 4 Is Critical

For $\mathcal{V}_A^{\mathfrak{M}}$ to be well-defined, the function from \mathcal{D}_τ into \mathcal{D}_σ in condition 4 must live

- in **some domain** of \mathcal{U} (since it is required that $\mathcal{V}_A^{\mathfrak{M}}(t_\rho) \in \mathcal{D}_\rho$ for all t , and $\mathcal{D}_\rho \in \mathcal{U}$): this is guaranteed by closure conditions on \mathcal{U} ;

Condition 4 Is Critical

For $\mathcal{V}_A^{\mathfrak{M}}$ to be well-defined, the function from \mathcal{D}_τ into \mathcal{D}_σ in condition 4 must live

- in **some domain** of \mathcal{U} (since it is required that $\mathcal{V}_A^{\mathfrak{M}}(t_\rho) \in \mathcal{D}_\rho$ for all t , and $\mathcal{D}_\rho \in \mathcal{U}$): this is guaranteed by **closure conditions on \mathcal{U}** ;
- in **a certain domain** of \mathcal{U} , namely $\mathcal{D}_{\tau \rightarrow \sigma}$; for this, $\mathcal{D}_{\tau \rightarrow \sigma}$ must be **big enough**.

If $\mathcal{V}_A^{\mathfrak{M}}$ **is** well-defined, we call $\mathfrak{M} = \langle \mathcal{D}_\tau, \mathcal{J} \rangle$ a **(general) model**.

Models

Hence: Not all interpretations are general models, but we restrict our attention to the general models.

If $\mathcal{D}_{\tau \rightarrow \sigma}$ is the set of **all** functions from \mathcal{D}_{τ} to \mathcal{D}_{σ} , then it is certainly “big enough”. In this case, we speak of a **standard model**. Important for **completeness**.

If $\mathcal{V}_A^{\mathfrak{M}}$ is a general model and A an assignment, **then** $\mathcal{V}_A^{\mathfrak{M}}$ is uniquely determined.

$\mathcal{V}_A^{\mathfrak{M}}(t)$ is **value** of t in $\mathcal{V}_A^{\mathfrak{M}}$ wrt. A .

Note that in **contrast to first-order logic**, “model” does **not** mean “an interpretation that makes a formula true”.

Satisfiability and Validity

A formula (term of type *bool*) ϕ is **satisfiable wrt. a model** \mathfrak{M} if there exists an assignment A such that $\mathcal{V}_A^{\mathfrak{M}}(\phi) = T$.

A formula ϕ is **valid wrt. a model** \mathfrak{M} if for all assignments A , we have $\mathcal{V}_A^{\mathfrak{M}}(\phi) = T$.

A formula ϕ is **valid in the general sense** if it is valid in every **general model**.

A formula ϕ is **valid in the standard sense** if it is valid in every **standard model**.

Existence of Values

Closure conditions for **general models** guarantee every well-formed term has a value under every assignment, and this means that certain values **must exist**, e.g.,

- Closure under functions: since $\mathcal{V}_A^{\mathfrak{M}}(\lambda x^\tau. x)$ is defined, the identity function from \mathcal{D}_τ to \mathcal{D}_τ must always belong to $\mathcal{D}_{\tau \rightarrow \tau}$.
- Closure under application: if $\mathcal{D}_{\mathbb{N}}$ is natural numbers, and $\mathcal{D}_{\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}}$ contains addition function p where $p x y = x + y$, then $\mathcal{D}_{\mathbb{N} \rightarrow \mathbb{N}}$ must contain k where $k x = 2x + 5$, since $k = \mathcal{V}_A^{\mathfrak{M}}(\lambda x_{\mathbb{N}}. f(f x x) y)$ where $A(f) = p$ and $A(y) = 5$.

Basic Rules

We now give the core calculus of HOL.

The constants *True* and *False* are not strictly necessary here: we will later **define** them (i.e., regard them as **syntactic sugar**).

So the rules can be stated using only the constants $=$, \rightarrow , and ϵ , but we will “**cheat**” and also use *True*, *False*, \vee to write rule *tof* (“true or false”).

Basic Rules in Sequent Notation

$$\frac{}{\Gamma \vdash \phi = \phi} \text{refl}$$

$$\frac{\Gamma \vdash \phi = \eta \quad \Gamma \vdash P(\phi)}{\Gamma \vdash P(\eta)} \text{subst}$$

$$\frac{\Gamma \vdash \phi x = \eta x}{\Gamma \vdash \phi = \eta} \text{ext}^*$$

$$\frac{\{\phi\} \cup \Gamma \vdash \eta}{\Gamma \vdash \phi \rightarrow \eta} \text{impl}$$

$$\frac{\Gamma \vdash \phi \rightarrow \eta \quad \Gamma \vdash \phi}{\Gamma \vdash \eta} \text{mp}$$

$$\frac{}{\Gamma \vdash (\phi \rightarrow \eta) \rightarrow (\eta \rightarrow \phi) \rightarrow \phi = \eta} \text{iff}$$

$$\frac{}{\phi = \text{True} \vee \phi = \text{False}} \text{tof}$$

$$\frac{\Gamma \vdash \phi x}{\Gamma \vdash \phi(\epsilon x. \phi x)} \text{selectl}$$

Axiom of Infinity

There is one additional rule (axiom) that will give us the existence of **infinite sets**:

$$\frac{}{\exists f^{(ind \rightarrow ind)}. \textit{injective } f \wedge \neg \textit{surjective } f} \textit{infty}$$

Has special role. Interesting to look at HOL **with or without infinity**. **Won't** consider infinity today.

Note “**cheating**” (use of \exists).

These eight (nine) rules are the entire basis!

Soundness and Completeness

Soundness is straightforward [And02, p. 240].

Completeness only follows w.r.t. **general models**, as opposed to **standard models**. Recall that a **standard model** is one where $\mathcal{D}_{\tau \rightarrow \sigma}$ is always the set of **all** functions from \mathcal{D}_{τ} to \mathcal{D}_{σ} . There are formulas that are **valid** in certain general models, but not in standard models, and which cannot be proven in our **calculus** (this reconciles HOL with Gödel's incompleteness theorem).

If we consider a version of HOL without **infinity**, then **every model is a standard model** and so completeness holds.

Isabelle/HOL

We now look at a particular instance of HOL (given by defining certain types and constants) which essentially corresponds to the **HOL theory of Isabelle**.

We present language and **rules** using “mathematical” syntax, but also comparing with Isabelle (concrete/**HOAS**) syntax.

We take **polymorphism** back on board.

(Central Parts of the) Language

 $\Sigma_0 =$

$$\{ \begin{array}{ll} \textit{True}, \textit{False} & : \textit{bool}, \\ \neg_ & : \textit{bool} \rightarrow \textit{bool}, \\ _ \wedge _, _ \vee _, _ \rightarrow _ & : \textit{bool} \rightarrow \textit{bool} \rightarrow \textit{bool}, \\ \forall _, \exists _ & : (\alpha \rightarrow \textit{bool}) \rightarrow \textit{bool}, \\ \epsilon_ & : (\alpha \rightarrow \textit{bool}) \rightarrow \alpha, \\ \textit{if_then_else_} & : \textit{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha, \\ _ = _ & : \alpha \rightarrow \alpha \rightarrow \textit{bool} \end{array} \}$$

Basic Rules in Isabelle Notation

```
refl:           "t = t"
subst:          "[| s = t; P(s) |] ==> P(t)"
ext:            "(!!x. (f x) = g x) ==>
                (%x. f x) = (%x. g x)"
impI:           "(P ==> Q) ==> P-->Q"
mp:             "[| P-->Q;  P |] ==> Q"
iff:            "(P-->Q) --> (Q-->P) --> (P=Q)"
True_or_False: "(P=True) | (P=False)"
selectI:        "P (x) ==> P (@x. P x)"
```

See [HOL.thy](#).

Basic Rules in Mixed Notation

$$\begin{array}{c}
 \frac{}{\phi = \phi} \textit{refl} \qquad \frac{\phi = \eta \quad P(\phi)}{P(\eta)} \textit{subst} \\
 \frac{\phi x = \eta x}{\phi = \eta} \textit{ext}^* \qquad \frac{\phi \Longrightarrow \eta}{\phi \rightarrow \eta} \textit{impl} \\
 \frac{\phi \rightarrow \eta \quad \phi}{\eta} \textit{mp} \\
 \frac{}{(\phi \rightarrow \eta) \rightarrow (\eta \rightarrow \phi) \rightarrow \phi = \eta} \textit{iff} \\
 \frac{}{\phi = \textit{True} \vee \phi = \textit{False}} \textit{tof} \qquad \frac{\phi x}{\phi(\epsilon x. \phi x)} \textit{selectI}
 \end{array}$$

No more “Cheating”: The Definitions

$$True = (\lambda x^{bool}.x = \lambda x.x)$$

$$\forall = \lambda\phi^{\alpha \rightarrow bool}.(\phi = \lambda x.True)$$

$$False = \forall\phi^{bool}.\phi$$

$$\vee = \lambda\phi\eta.\forall\psi.(\phi \rightarrow \psi) \rightarrow (\eta \rightarrow \psi) \rightarrow \psi$$

$$\wedge = \lambda\phi\eta.\forall\psi.(\phi \rightarrow \eta \rightarrow \psi) \rightarrow \psi$$

$$\neg = \lambda\phi.(\phi \rightarrow False)$$

$$\exists = (\lambda\phi.\phi(\epsilon x.\phi x))$$

$$If = \lambda\phi^{bool}xy.\epsilon z.(\phi = True \rightarrow z = x) \wedge \\ (\phi = False \rightarrow z = y)$$

Note: Different Syntaxes

Mathematical

vs. Isabelle, e.g.

$\neg \phi$

Not Phi

$\lambda x^{Bool}. P$

`%x :: bool. P`

HOAS

vs. concrete, e.g.

$\forall (\lambda x^{\tau}. (\wedge p(x) q(x)))$

$\forall x^{\tau}. p(x) \wedge q(x)$

$\in (P)$

$\in x. P(x)$

We use all those forms as convenient. For displaying Isabelle files, we will sometimes use a style where some ASCII words (e.g. %) are replaced with mathematical symbols (e.g. λ).

Conclusions on HOL

- HOL generalizes semantics of FOL:
 - *bool* serves as type of propositions;
 - Syntax/semantics allows for higher-order functions.
- Logic is rather minimal: 8 or 9 rules, based on 3 constants, **soundness** straightforward.
- Logic **complete** (w.r.t. general models, but not **standard** models).
- Next lecture we will see how all well-known inference rules can be derived. ►

More Detailed Explanations

HOL Applications

Theorem proving in higher-order logic is an active research area with some impressive applications.

HOL “Is” an Object Logic

The differences between \mathcal{M} and HOL are subtle and the matter is further complicated by the fact that there are little variations in the way in which the Isabelle metalogic \mathcal{M} on the one hand and the object logic HOL are presented.

But what matters for us here is that HOL is an **object logic**, i.e., it is one of the object logic that can be represented by \mathcal{M} , just like **propositional logic** or **first-order logic**. That is to say, we **use** HOL as object logic.

Modeling of Problems/Domains in HOL

We have previously looked at **metatheory**, i.e., how can one logic be **represented/modeled** in a metalogic.

In particular, we have seen how **general reasoning principles** can be derived in the metalogic.

We now set aside the issue of metalogics, but there is still an issue of modeling one system within another: how do we model problems/domains within HOL? How do we derive reasoning principles?

Classical Reasoning

Recall the distinction between classical and intuitionistic logics. There is a particular rule in HOL from which the rule of the excluded middle can be derived. This is in contrast to constructive (intuitionistic) logics.

Not just HOL, but Isabelle/HOL

We use Isabelle/HOL, and this means that HOL is an object logic represented by the metalogic \mathcal{M} .

Forgoing Metalogic

There are theorem proving systems that have no metalogic, but rather have a particular logic hard-wired into them, e.g. a HOL system or PVS.

Constructive Provers

Constructive provers are based on **intuitionistic** logic. The rationale is that one has to give **evidence** for any statement. **Coq** and **Nuprl** are examples of such systems.

Safety

The principle is simple: the smaller a system is, the easier it is to check that it is correct, and the more confident one can be about it.

We have seen this before when we argued for the use of **metalogics**.

However, in that context, we still had to **add further axioms** to \mathcal{M} . Here this is **not the case**.

No Axiomatic Extensions

What we attempt to do here has similarities to the process of **representing** an object logic in a metalogic. But an important difference must be noted.

We will see many extensions of the HOL kernel by **constants** (and **types**). The definitions of those constants and types involve axioms that must be added **according to a strict discipline**. Other than that, we will **not** add any axioms!

ZFC

ZFC stands for **Z**ermelo-**F**ränkel set theory with **c**hoice [Dev93, Ebb94].

Mod Cons

“Mod cons” stands for “modern conveniences”.

Type Order

Recall the definition of an [order on types](#).

Assume here, as we did in the [lecture on representing syntax](#), that there is a type i of individuals and a type o for truth values. These objects are of 0th-order type.

A predicate symbol is a function taking individuals to a truth value, so it is an object of first-order type. A function symbol is a function taking individuals to an individual, so it is also an object of first-order type.

In the [table](#), we classify logics by the [order](#) of the [logical symbols](#) (e.g., for first-order logic: variables, function symbols, predicate symbols).

A hierarchy of logics is obtained by the following alternation:

- admit an additional order for the non-logical symbols in the logic;
- admit [quantification](#) over symbols of that order.

If one only admits symbols of type o (propositional variables), one obtains **propositional logic**. At this stage, one also admits symbols of type i , although this is still degenerate since one cannot form any formulas with them (without any function or predicate symbols).

One can now admit quantification over individuals, although one can still not do much with that since one cannot form any formulas.

Now, if one admits symbols of first-order type (function symbols, predicate symbols), one obtains **first-order logic**. Quantification is allowed only over individuals.

Now, if one admits quantification over symbols of first-order type, one obtains **second-order logic**.

Now, if one admits symbols of second-order type (symbols taking predicate/function symbols as arguments), one obtains **third-order logic**.

Now, if one admits quantification over symbols of second-order type, one

obtains **fourth-order** logic.

Hence quantification over n th-order variables corresponds to $(2n)$ th-order logic.

This particular terminology is due to [And02, §50] and in the end, one will never bother to discuss, say, 7 th-order logic, since higher-order logic is the **union** of all logics of finite order, and this is what we will be working with.

You may find that especially below first-order logic, there are some oddities in this hierarchy (e.g., should propositional logic be called “ -1 th”-order logic?). However, once you take higher-order logic as your point of reference and not propositional or first-order logic, which can just be viewed as special cases, you will probably not find this bothering anymore.

subrel

Consider the binary predicate *subrel* which takes two unary relations as arguments. $\text{subrel}(R, S)$ is defined as true whenever R is a subrelation of S , i.e. when $\forall x. R(x) \rightarrow S(x)$.

Now instead of defining such a predicate and writing, say, a formula $\text{subrel}(R', S')$, one could abstract from that name and write

$$\forall X. (X(R, S) \leftrightarrow (\forall x. R(x) \rightarrow S(x))) \rightarrow X(R', S')$$

The subformula $X(R, S) \leftrightarrow (\forall x. R(x) \rightarrow S(x))$ is true if and only if X is indeed the predicate *subrel* and so the entire formula is true if R' is indeed a subrelation of S' .

The Type signature of HOL

As before, we use the letter \mathcal{B} to denote a particular set of type constructors.

Note that this set is not hard-wired into HOL, but can be specified as part of a particular HOL language. One can therefore speak of \mathcal{B} as a *type signature*.

\mathcal{B} is some fixed set “defined by the user”. In Isabelle, there is a syntax provided for this purpose.

However, some type constructors are *always present*.

ind

ind (“indefinite”) is a type constructor which stands for a type with infinitely many members, a concept which is central in HOL, as we will see later.

Pair Type

For any two types τ and σ , we write $\tau \times \sigma$ for the type of pairs where the first component is of type τ and the second component is of type σ .

The infix syntax is in analogy to \rightarrow .

The pair type is not in the core of HOL, but it can be **defined** in it.

Variable Augmented with Type?

Strictly speaking, a variable should be augmented with a type in the term $\lambda x. e$. The type τ in $\lambda x^\tau. e$ is often omitted if it is clear from the context. Here, “clear from the context” does not only refer to a human reader, but also to the formal Isabelle syntax. It is possible to give an explicit type to a variable and also specify the class of that type (one can impose type and class [constraints](#)), but often, Isabelle can infer the type of a variable.

Generalizing λ^{\rightarrow} to Polymorphism

We have seen the **generalization** of λ^{\rightarrow} to polymorphism.

Note that in order to simplify the presentation, we neglect polymorphism in the **section on semantics**. In that section, τ and σ will be metavariables (used in the description of the formalism) ranging over types, rather than **type variables** of a polymorphic type system.

The Type of $=$

The type of $=$ can either be declared to be **polymorphic** (then α would be a type variable), or one can, conceptually, assume that there is an infinite set containing a constant $=_{\tau}$ for each type τ .

Intuition for the Constants

We will give the formal semantics *later*, but now some intuition!

True, *False*, $=$ and \rightarrow have the meanings you would expect although $=$ is more general than in first-order logic since for any type τ , there is the $=$ symbols for terms of type τ . This is just like in \mathcal{M} .

The constant ϵ (also called **Hilbert operator**) denotes a function which takes a set and picks a member from that set. We give a more detailed intuitive explanation *later*.

Note that the derivation rules *can be formulated* without using *True* and *False*.

Isabelle Syntax for ϵ

The Hilbert operator is written `Eps` in Isabelle.

Now similarly as for quantifiers \forall, \exists , Isabelle provides a syntax `SOME x . P` that stands for `Eps. λx . P` .

In older versions of Isabelle, `SOME` was written `'@'`.

Structures in FOL

Recall the [definition of a structure](#) in FOL. Here we use the notation

$$\langle \mathcal{D}, f_1^{\mathcal{I}}, \dots, f_k^{\mathcal{I}}, r_1^{\mathcal{I}}, \dots, r_j^{\mathcal{I}} \rangle$$

which corresponds to the notation used previously in the obvious way: each $f_i^{\mathcal{I}}$ is the (semantic) function $I_{\mathcal{A}}(f_i)$, and $p_i^{\mathcal{I}}$ is the (semantic) predicate $I_{\mathcal{A}}(p_i)$.

Heterogeneous Algebra

This is really just an aside.

Recall the definition of a **first-order algebra**. An algebra assigns a meaning to function symbols. If such an algebra is sorted we speak of a **heterogeneous algebra**.

Prod: Encoding $X \times Y$

According to usual mathematical practice, one would argue that if two sets A and B are well-defined, then the set $A \times B$ of pairs (tuples) (a, b) where $a \in A$ and $b \in B$ is also well-defined.

Inhabitation

It is crucial in the semantics that any type is **inhabited**, i.e., has an element. The **function space** $X \rightarrow Y$ is empty if X is empty.

This means that $\mathcal{D}_{\tau \rightarrow \sigma}$ would necessarily be empty if τ is empty. This in turn would lead to a very nasty behavior of β -reduction: for any types τ, σ and a constant c of type σ , we expect $(\lambda x^\tau. c)x = c$. But this wouldn't hold if we cannot give a semantics to $(\lambda x^\tau. c)$ since $\mathcal{D}_{\tau \rightarrow \sigma}$ is empty.

Therefore: inhabitation.

We will also see two more specific points where inhabitation is crucial, one concerning the **ϵ -operator** and one concerning **sets**.

Dependent Type

When we write $ch \in \Pi_{X \in \mathcal{U}}.X$, i.e., ch is of dependent type, then this is a statement on the semantic level. The expression $\Pi_{X \in \mathcal{U}}.X$ is not part of the formal syntax of HOL (unlike in LF, a system we have not treated here), and its meaning is only described in plain English, by saying that ch takes a set $X \in \mathcal{U}$ as argument and returns a **member** of X .

What Are Functions?

In any basic math course on algebra, we learn that a binary relation between X and Y is set of a pairs of tuples of the form (x, y) where $x \in X$ and $y \in Y$. One also calls such a set a **graph** since one can view pairs (x, y) as edges.

We also learn that a relation R is called a **function** from X to Y if for each $x \in X$, there exists exactly one $y \in Y$ such that $(x, y) \in R$.

Provided that Y is nonempty, a function from X to Y always exists.

Thus the **set of functions** from X to Y , denoted $X \rightarrow Y$, is a nonempty subset of the **set of relations** on X and Y , i.e., $\wp(X \times Y)$. Since $X \rightarrow Y$ is nonempty, by **Prod** we have that $X \rightarrow Y \in \mathcal{U}$.

Why $\{1\}$ and $\{T, F\}$?

Of course, the conditions on \mathcal{U} do not per se enforce the existence of sets containing the elements 1 or T or F . Just as well, one could say that they enforce the existence of sets containing elements ☕ or 🚲 or ⚽. It is only because the **name** of a semantic element is ultimately irrelevant that we claim, without loss of generality, that there is a 1-element set $\{1\}$ and a 2-element set $\{T, F\}$. We say that these sets are **distinguished** because they play a special role in the setup of the semantics.

Interpretation as Structure

Instead of having a tuple $\langle \{\mathcal{D}_\tau\}_\tau, \mathcal{J} \rangle$, where the function \mathcal{J} assigns a meaning to each constant, we could have a structure with one component $c^{\mathcal{J}}$ for each constant c . This is only a small difference of notation.

Standard and General Models

General models must be distinguished from **standard** models, as well see later.

We sometimes omit the word “general” in **general model**.

Type Subscript

For $=$ and ϵ , we give type subscripts in the presentation of the semantics since we assume, conceptually, that there are infinitely many **copies** of those constants, one for each type. We do this to avoid explicit polymorphism in this presentation.

Intuition for ϵ and ch

We have

$$\mathcal{J}(\epsilon_{(\tau \rightarrow bool) \rightarrow \tau})(f) = \begin{cases} ch(f^{-1}(\{T\})) & \text{if } f^{-1}(\{T\}) \neq \emptyset \\ ch(\mathcal{D}_\tau) & \text{otherwise} \end{cases}$$

ch is a (semantic) function which takes a nonempty set and returns an element from that set. f is a semantic function from \mathcal{D}_τ to \mathcal{D}_{bool} . However, f can be interpreted as set. This is done in all formality here: we write $f^{-1}(\{T\})$. One says that f is the characteristic function of the set $f^{-1}(\{T\})$.

Now the type of ϵ is $(\tau \rightarrow bool) \rightarrow \tau$ (for any τ), so ϵ expects a function as argument, which can be interpreted as a **set** as just stated. This set can be empty or nonempty. In case it is nonempty, an element is picked from the set non-deterministically. If the set is empty, an element from

the **type** τ (which must be nonempty since each type is interpreted as nonempty set). Note the importance of inhabitation.

Type-Indexed Assignments

An **assignment** (previously called **valuation**) maps variables to elements of a **domain**.

A type-indexed collection of assignments is an assignment that respects the types: a variable of type τ will be assigned to a member of \mathcal{D}_τ [GM93]. Note that a variable has a type by virtue of a context Γ , which is suppressed in our presentation of models.

Type Subscript

In the presentation of models, we give type subscripts for the cases $\mathcal{V}_A^{\mathfrak{M}}(s_{\tau \rightarrow \sigma} t_{\sigma})$ and $\mathcal{V}_A^{\mathfrak{M}}(\lambda x^{\tau}. t_{\sigma})$ to indicate the types of s and t in those definitions. Note that a term has a type in a certain context Γ , which is suppressed in our presentation of models. The semantics is only defined for well-formed terms, in particular, applications and abstractions having types of the indicated forms.

$$A[x \leftarrow e]$$

$A[x \leftarrow e]$ denotes the assignment that is identical to A except that $A(x) = e$.

Condition 4 Violated

In condition 4, the semantics of $\lambda x^\tau. t_\sigma$ is defined unambiguously as a certain function. But in general, there is no guarantee that this function is actually in $\mathcal{D}_{\tau \rightarrow \sigma}$, and in this case, $\mathfrak{M} = \langle \{\mathcal{D}_\tau\}_\tau, \mathcal{J} \rangle$ would not be a model.

Completeness

This is a standard trick when faced with the problem that a deductive system is not complete. One can either enlarge the set of axioms, or one can weaken the models by permitting more models. If we allow more models, then fewer theorems will be valid (i.e., hold in all models), and so fewer theorems will have to be provable in the derivation system.

Here, completeness is based on general models, and **not standard** models. This resolves the apparent contradiction with Gödel's incompleteness theorem: HOL with infinity contains *I*, hence the **natural numbers**, hence arithmetic . . .). By Gödel's incompleteness theorem, there cannot be a consistent derivation system that can prove all valid theorems in the natural numbers.

A readable account on this problem can be found in [And02, ch. 7].

Finite Domains

We might consider a version of HOL without infinity, i.e., one where each **domain** is finite (note that \mathcal{U} is still infinite, since there are infinitely many types, e.g., $bool$, $bool \rightarrow bool$, $bool \rightarrow bool \rightarrow bool$, \dots)).

One can see that **every** function in such a finite domain is representable as a λ -term, and so for any σ and τ , we **must have** $\mathcal{D}_{\tau \rightarrow \sigma} = \mathcal{D}_{\tau} \rightarrow \mathcal{D}_{\sigma}$.

For details consult [And02, §54].

Syntaxes for Rules

We will mix **natural deduction** (with discharging assumptions), natural deduction written in **sequent style**, and Isabelle syntax.

For a thorough account of this, consult [SH84].

Some **general** remarks about the correspondence: A rule

$$\frac{\psi}{\phi}$$

in ND notation corresponds to an Isabelle rule $\psi \Longrightarrow \phi$.

A rule

$$\frac{[\rho] \quad \vdots \quad \psi}{\phi}$$

is written as

$$\frac{\rho, \Gamma \vdash \psi}{\Gamma \vdash \phi}$$

in sequent style or

$$\frac{\rho \Longrightarrow \psi}{\phi}$$

using the Isabelle meta-implication \Longrightarrow .

A rule

$$\frac{\psi}{\phi(x)}$$

with side condition that x must not occur free in any undischarged assumption on which ψ depends is written as

$$\frac{\Gamma \vdash \psi}{\Gamma \vdash \phi(x)}$$

in sequent style, where the side condition reads: x must not occur free in Γ . Using the Isabelle meta-universal quantification, the rule is written

$$\frac{\bigwedge x. \psi}{\phi(x)}$$

We will switch between the various ways of writing the rules! This means in particular that we will use two symbols from Isabelle's metalogic, \implies and \bigwedge just like logical symbols.

Readability of Rule *tof*

Rule *tof* can be written as follows:

$$\frac{\forall\psi. \ (\phi = (\lambda x.x = \lambda x.x) \rightarrow \psi) \rightarrow (\phi = ((\lambda\eta.\eta) = \lambda x.(\lambda x.x = \lambda x.x)) \rightarrow \psi) \rightarrow \psi}{\text{tof}}$$

Our notation for rule *tof* is thus based on the following definitions:

$$\begin{aligned} \text{True} &= (\lambda x^{\text{bool}}.x = \lambda x.x) \\ \forall &= \lambda\phi^{\alpha \rightarrow \text{bool}}.(\phi = \lambda x.\text{True}) \\ \text{False} &= \forall\phi^{\text{bool}}.\phi \\ \vee &= \lambda\phi\eta.\forall\psi.(\phi \rightarrow \psi) \rightarrow (\eta \rightarrow \psi) \rightarrow \psi \end{aligned}$$

Intuition for . . .

True, False, \forall , \exists , \vee , \wedge , \neg , If

Intuition for *True*

$$True = (\lambda x^{Bool}.x = \lambda x.x)$$

The term $\lambda x^{Bool}.x = \lambda x.x$ evaluates to T , and so it is a suitable definition for the constant *True*.

Note that we give the type for x once. The right-hand side $\lambda x.x$ will thereby also be forced to be of type $bool \rightarrow bool$.

This is necessary for reasons that will become clear later.

Intuition for \forall

$$\forall = \lambda\phi.(\phi = \lambda x. True)$$

Note the use of **HOAS** here. \forall should be a function that expects an argument ϕ of type $\alpha \rightarrow bool$ (generalizing the technique we used for encoding **first-order** \forall). So ϕ is such that when you pass it an argument x of type α , it will return a proposition (something of type $bool$).

The **expected semantics** of $\forall\phi$ wrt. a **model** \mathfrak{M} and an assignment A is: $\mathcal{V}_A^{\mathfrak{M}}(\forall\phi) = T$ iff $\mathcal{V}_{A[x \leftarrow e]}^{\mathcal{M}}(\phi x) = T$ for any e (from the **domain** of x 's type).

Now when does ϕx hold **for all** x ? This is the case exactly when ϕx evaluates to T for all x , which is the same (applying some HOL rules) as saying that ϕ is the function $\lambda x. True$.

Here α could be arbitrarily instantiated to some type.

Intuition for \forall

$$\forall = \lambda\phi\eta.\forall\psi.(\phi \rightarrow \psi) \rightarrow (\eta \rightarrow \psi) \rightarrow \psi$$

First, observe the similarity of this definition with the \forall - E rule of propositional logic.

Secondly, just go through the cases:

- If ϕ is true, then:
 - If ψ is false, then $\phi \rightarrow \psi$ is false and so $(\phi \rightarrow \psi) \rightarrow (\eta \rightarrow \psi) \rightarrow \psi$ is true;
 - If ψ is true, then $(\eta \rightarrow \psi) \rightarrow \psi$ is true and hence $(\phi \rightarrow \psi) \rightarrow (\eta \rightarrow \psi) \rightarrow \psi$ is true.

Thus for all ψ , we have that $(\phi \rightarrow \psi) \rightarrow (\eta \rightarrow \psi) \rightarrow \psi$ is true.

- Otherwise, if η is true, then:

- If ψ is false, then $\eta \rightarrow \psi$ is false and so $(\eta \rightarrow \psi) \rightarrow \psi$ is true and so $(\phi \rightarrow \psi) \rightarrow (\eta \rightarrow \psi) \rightarrow \psi$ is true.
- If ψ is true, then $(\eta \rightarrow \psi) \rightarrow \psi$ is true and hence $(\phi \rightarrow \psi) \rightarrow (\eta \rightarrow \psi) \rightarrow \psi$ is true.

Thus for all ψ , we have that $(\phi \rightarrow \psi) \rightarrow (\eta \rightarrow \psi) \rightarrow \psi$ is true.

- Otherwise (if both ϕ and η are false), then for all ψ , both $\phi \rightarrow \psi$ and $\eta \rightarrow \psi$ are true, and so there exists a ψ , say $\psi \equiv \text{False}$, such that $(\phi \rightarrow \psi) \rightarrow (\eta \rightarrow \psi) \rightarrow \psi$ is false.

Thus it is **not** the case that for all ψ , $(\phi \rightarrow \psi) \rightarrow (\eta \rightarrow \psi) \rightarrow \psi$ is true.

So the definition of \forall behaves exactly as it should.

Intuition for \wedge

$$\wedge = \lambda\phi\eta.\forall\psi.(\phi \rightarrow \eta \rightarrow \psi) \rightarrow \psi$$

Similarly as for \vee , we can go through the cases:

- If η is false, then there exists a ψ , namely $\psi \equiv \text{False}$, such that $\eta \rightarrow \psi$ is true, hence $\phi \rightarrow \eta \rightarrow \psi$ is true, hence $\phi \rightarrow \eta \rightarrow \psi) \rightarrow \psi$ is false.
Thus it is **not** the case that for all ψ , $\phi \rightarrow \eta \rightarrow \psi) \rightarrow \psi$ is true.
- Otherwise, if ϕ is false, then $\phi \rightarrow \eta \rightarrow \psi$ is true, and there exists a ψ , namely $\psi \equiv \text{False}$, such that $\phi \rightarrow \eta \rightarrow \psi) \rightarrow \psi$ is false.
Thus it is **not** the case that for all ψ , $\phi \rightarrow \eta \rightarrow \psi) \rightarrow \psi$ is true.
- Otherwise (if ϕ and η are true), then:
 - If ψ is false, then $\eta \rightarrow \psi$ is false, hence $\phi \rightarrow \eta \rightarrow \psi$ is false, hence $\phi \rightarrow \eta \rightarrow \psi) \rightarrow \psi$ is true.

- If ψ is true, then $\phi \rightarrow (\eta \rightarrow \psi) \rightarrow \psi$ is true.

Thus for all ψ , we have that $\phi \rightarrow (\eta \rightarrow \psi) \rightarrow \psi$ is true.

So the definition of \wedge behaves exactly as it should.

Intuition for \neg

$$\neg = \lambda\phi.(\phi \rightarrow \textit{False})$$

We know that one already from [propositional logic](#).

Intuition for \exists

$$\exists = (\lambda\phi.\phi(\epsilon x.\phi x))$$

Using the **abstract syntax** for ϵ , one could also write

$$\exists = (\lambda\phi.\phi(\epsilon\phi))$$

Recall first the **definition of \forall** to understand the type of \exists .

The **expected semantics** of $\exists\phi$ wrt. a **model \mathfrak{M}** and an assignment A is:

$\mathcal{V}_A^{\mathfrak{M}}(\exists\phi) = T$ iff $\mathcal{V}_{A[x \leftarrow e]}^{\mathfrak{M}}(\phi x) = T$ for some e (from the **domain** of x 's type).

The **semantics of ϵ** is such that $\phi(\epsilon\phi)$ is true, if and only if a term t exists for which $\phi(t)$ is true.

So this is exactly the expected semantics of $\exists\phi$.

Intuition for *If*

$$If = \lambda\phi xy. \epsilon z. (\phi = True \rightarrow z = x) \wedge (\phi = False \rightarrow z = y)$$

The constant *If* stands for the if-then-else construct. Note first that $\epsilon z. (\phi = True \rightarrow z = x) \wedge (\phi = False \rightarrow z = y)$ is η -equivalent to $\epsilon z. (\lambda z. (\phi = True \rightarrow z = x) \wedge (\phi = False \rightarrow z = y)) z$, which is written $\epsilon(\lambda z. (\phi = True \rightarrow z = x) \wedge (\phi = False \rightarrow z = y))$ in the “real” HOL syntax, which uses the concept of **HOAS**.

The expression $\epsilon(\lambda z. (\phi = True \rightarrow z = x) \wedge (\phi = False \rightarrow z = y))$ picks a term from the set of terms z such that $(\phi = True \rightarrow z = x) \wedge (\phi = False \rightarrow z = y)$ holds. But this means that $z = x$ if $\phi = True$, or $z = y$ if $\phi = False$.

Since *If* should be a function which takes ϕ , x and y as arguments, we

must abstract over those variables, giving

$$\lambda\phi xy.\epsilon z.(\phi = \textit{True} \rightarrow z = x) \wedge (\phi = \textit{False} \rightarrow z = y).$$

Definitions by $=$ or \equiv ?

It is a design choice if we want to add these definitions at the level of the **object logic (HOL)** or at the level of the \mathcal{M} . In the first case, we would use $=$ and have axioms such as

$$True = (\lambda x^{bool}.x = \lambda x.x)$$

In the second case, we would have meta-axioms

$$True \equiv (\lambda x^{bool}.x = \lambda x.x)$$

This would mean that we would regard $True$ merely as **syntactic sugar**. The second way corresponds to what is done in Isabelle, see **HOL.thy**. It is technically more convenient since **rewriting** is based on meta-level equalities.

Logically, it is not a big difference which way one chooses. We will have an exercise on this.

Intuition for *False*

$$\textit{False} = \forall\phi.\phi$$

The essence of *False* is that anything can be derived from it. But this is exactly what $\forall\phi.\phi$ says.

Concrete Syntax for \forall

The HOL constant \forall is defined first in the style of **HOAS**. But we also use concrete syntax, so we write $\forall x.\psi$ instead of $\forall(\lambda x.\psi)$. In the concrete syntax, one may also **annotate** the variable with a type.

Type Annotation for Variables

In HOL, the quantifiers, which one expects to be variable binders, are realized using λ in the style of HOAS.

We have said binding occurrences of variables in a λ -term should, strictly speaking, be annotated with a type, but that this type can often be omitted.

Now whenever we use **concrete** quantifier syntax for convenience, so we write $\forall x.\psi$ instead of $\forall(\lambda x.\psi)$ (and likewise for \exists), we may annotate the variable in the obvious way: $\forall x^\tau.\psi$ is concrete syntax for $\forall(\lambda x^\tau.\psi)$.

Sometimes we will annotate variables for clarity, sometimes we trust that the type is clear from the context.

Side Condition of ext

The rule

$$\frac{\Gamma \vdash \phi x = \eta x}{\Gamma \vdash \phi = \eta} ext$$

has the side condition that $x \notin FV(\Gamma)$.

Phrased like

$$\frac{\phi x = \eta x}{\phi = \eta} ext$$

the rule has the side condition that x must not occur **freely** in the derivation of $\phi x = \eta x$.

Concrete Syntax for ϵ , \forall , \exists

For readability, we will frequently use a syntax that one is more used to than [higher-order abstract syntax](#):

$\epsilon x. \phi x$ stands for $\epsilon(\phi)$.

$\forall x. \phi(x)$ stands for $\forall(\phi)$, and likewise for \exists .

We have done the same [previously](#) for \mathcal{M} .

Infinity in HOL

The infinity axiom

$$\frac{}{\exists f^{(ind \rightarrow ind)}.injective\ f \wedge \neg surjective\ f} \text{infty}$$

says that there is a function from I to I (the postulated infinite set in \mathcal{U}) which is injective (any two different elements e, e' of I have different images under f) but not surjective (there exists an element of I which is not the image of any element).

Such a function can only exist if I is infinite, and in fact the axiom expresses the very essence of infinity, as we will see [later](#).

Think of the natural numbers and the successor function as an example: for any two different natural numbers, the successors are different, and the number 0 is not the successor of any number.

Injective – Surjective

A function is **injective** (also called **one-to-one**) if any two different elements e, e' have different images under f .

A function is **surjective** (also called **onto**) if for any e (in the function's range type, i.e. the type τ where $\sigma \rightarrow \tau$ is the type of the function), there exists an e' whose image under the function is e .

Formally, the definitions are

$$\begin{aligned} \text{injective} &= \lambda f. \forall x y. f\ x = f\ y \rightarrow x = y \\ \text{surjective} &= \lambda f. \forall y. \exists x. f\ x = y \end{aligned}$$

Just One Lambda!

Note that the λ -binder of the object logic HOL is not distinguished from the λ -binder of Isabelle's metalogic \mathcal{M} . One could introduce an object level constant *lambda*, but one quickly sees that it would be an unnecessary overhead.

Optional Type Superscript

As we have learned [previously](#), λ -abstracted variables should have a type superscript, although this superscript is often omitted since the type can be [inferred](#).

Since $\forall x. p(x) \wedge q(x)$ is the “concrete syntax” version of $\forall (\lambda x. (\wedge p(x) q(x)))$, it makes sense that we allow an optional superscript also for \forall -bound (and likewise for \exists -bound) variables.

In Isabelle the optional type annotation is written using $::$ instead of a superscript.

Gather Constants with Same Type

For convenience (and to save space, we write $\dots a : \tau, b : \tau \dots$ as $\dots a, b : \tau \dots$ in a signature. This is of course **syntactic sugar**.

Notation Using $_$

We use a notation with $_$ to indicate the arity and fixity of constants, as this has been done for **type constructors** before.

The whole matter of arity of fixity is one of notational convenience. For example, as the type of \wedge indicates, we should write $(\wedge\phi)\psi$ (**Curried notation**), but we write $\phi \wedge \psi$ since it is more what we are used to.

HOL.thy

This file should be contained in your Isabelle distribution. In Freiburg it is

```
/usr/local/isabelle/Isabelle/src/HOL/HOL.thy
```

Moreover, in

```
/usr/local/isabelle/Isabelle/src/HOL/HOL_lemmas.ML
```

you find all the derivations of the rules presented in this lecture.

However, the presentation of this lecture is still partly based on Isabelle 98, which you will find in

```
/usr/local/isabelle/Isabelle98/src/HOL/HOL.thy
```

Note also that here in the slides, we use a style of displaying Isabelle files which uses some symbols beyond the [usual ASCII set](#).

HOL: Deriving Rules

Outline

Last lecture: Introduction to HOL

- Basic **syntax** and **semantics**
- Basic eight (or nine) **rules**
- **Definitions** of *True*, *False*, \wedge , \vee , \forall . . .

Today:

- **Deriving rules** for the defined constants
- **Outlook** on the rest of this course

Reminder: Different Syntaxes

Mathematical vs. Isabelle, e.g.

 $\neg \phi$

Not Phi

 $\lambda x^{bool}.P$

$\%x :: \text{bool}. P$

HOAS

vs. concrete, e.g.

 $\forall (\lambda x^\tau. (\wedge p(x) q(x)))$
 $\forall x^\tau. p(x) \wedge q(x)$
 $\in (P)$
 $\in x. P(x)$

We will be using **all those forms** as convenient.

Reminder: Definitions

$$True = (\lambda x^{bool}.x = \lambda x.x)$$

$$\forall = \lambda\phi^{\alpha \rightarrow bool}.(\phi = \lambda x.True)$$

$$False = \forall\phi^{bool}.\phi$$

$$\vee = \lambda\phi\eta.\forall\psi.(\phi \rightarrow \psi) \rightarrow (\eta \rightarrow \psi) \rightarrow \psi$$

$$\wedge = \lambda\phi\eta.\forall\psi.(\phi \rightarrow \eta \rightarrow \psi) \rightarrow \psi$$

$$\neg = \lambda\phi.(\phi \rightarrow False)$$

$$\exists = (\lambda\phi.\phi(\epsilon x.\phi x))$$

$$If = \lambda\phi xy.\epsilon z.(\phi = True \rightarrow z = x) \wedge \\ (\phi = False \rightarrow z = y)$$

Derived Rules

The **definitions** can be understood either semantically (checking if each definition captures the usual meaning of that constant) or by their properties (= derived rules).

We now look at the constants in turn and derive rules for them. We will present derivations in natural deduction style.

We usually proceed as follows: first show a rule involving a constant, then replace the constant with its definition (if applicable), then show the derivation.

Equality

- Rule *sym*

$$\frac{s = t}{t = s} \text{ } \textit{sym}$$

Equality

- Rule *sym* and ND derivation

$$\frac{s = t \quad \frac{}{s = s} \text{refl}}{t = s} \text{subst}$$

- Isabelle rule $s=t \implies t=s$. Proof script:

```
Goal "s=t ==> t=s";  
by (etac subst 1);           (* P is %x.x=s *)  
by (rtac refl 1);           (* s=s *)  
qed "sym";
```

Equality: Transitivity and Congruences

- Rule *trans*

$$\frac{s = t \quad r = s}{r = t} \text{trans}$$

Equality: Transitivity and Congruences

- Rule *trans* and ND derivation

$$\frac{s = t \quad r = s}{r = t} \textit{subst}$$

Isabelle rule `[| r=s; s=t |] ==> r=t`

Equality: Transitivity and Congruences

- Rule *trans* and ND derivation

$$\frac{s = t \quad r = s}{r = t} \text{subst}$$

Isabelle rule $[| \ r=s; \ s=t \ |] \implies r=t$

- Congruences (only Isabelle forms):
 $(f :: 'a \Rightarrow 'b) = g \implies f(x) = g(x) \quad (\text{fun_cong})$
 $x=y \implies f(x) = f(y) \quad (\text{arg_cong})$
 Isabelle proofs using *subst* and *refl*.

Equality of Booleans (*iffI*)

Rule *iffI*

$$\frac{\begin{array}{c} [P] \\ \vdots \\ Q \end{array} \quad \begin{array}{c} [Q] \\ \vdots \\ P \end{array}}{P = Q} \text{iffI}$$

Equality of Booleans (*iffI*)

Rule *iffI* and ND derivation

$$\frac{
 \frac{
 \frac{
 (P \rightarrow Q) \rightarrow (Q \rightarrow P) \rightarrow (P = Q)
 }{}
 }{}
 }{}
 }{}
 }{}
 \frac{
 \frac{
 \frac{
 [P]
 }{}
 }{}
 }{}
 }{}
 }{}
 \frac{
 \frac{
 \frac{
 [Q]
 }{}
 }{}
 }{}
 }{}
 }{}
 }{}
 }{}
 }{}
 P = Q$$

Isabelle rule `[| P ==> Q; Q ==> P |] ==> P=Q.`

Equality of Booleans (*iffD2*)

Rule *iffD2*

$$\frac{P = Q}{P} \quad \frac{Q}{P} \text{ iffD2}$$

Equality of Booleans (*iffD2*)

Rule *iffD2* and ND derivation

$$\frac{\frac{P = Q}{Q = P} \text{sym}}{P} \frac{Q}{\text{subst}}$$

Isabelle rule $[| P=Q; Q |] ==> P.$

True

$$True = ((\lambda x^{bool}.x) = (\lambda x.x))$$

• Rule *TrueI*

$$\frac{}{True} \quad TrueI$$

True

$$\text{True} = ((\lambda x^{\text{bool}}.x) = (\lambda x.x))$$

• Rule *TrueI*

$$\frac{}{(\lambda x.x) = (\lambda x.x)} \text{TrueI}$$

True

$$True = ((\lambda x^{bool}.x) = (\lambda x.x))$$

- Rule *TrueI* and ND derivation

$$\frac{}{(\lambda x.x) = (\lambda x.x)} refl$$

True

$$True = ((\lambda x^{bool}.x) = (\lambda x.x))$$

- Rule *TrueI* and ND derivation

$$\frac{}{(\lambda x.x) = (\lambda x.x)} \text{ refl}$$

- Rule *eqTrueE*

$$\frac{P = True}{P} \text{ eqTrueE}$$

True

$$True = ((\lambda x^{bool}.x) = (\lambda x.x))$$

- Rule *TrueI* and ND derivation

$$\frac{}{(\lambda x.x) = (\lambda x.x)} \text{ refl}$$

- Rule *eqTrueE* and ND derivation

$$\frac{P = True \quad \frac{}{True} \text{ TrueI}}{P} \text{ iffD2}$$

Isabelle rule $P=True \implies P$.

True (Cont.)

- Rule $eqTrueI$

$$\frac{P}{P = True} eqTrueI$$

True (Cont.)

- Rule *eqTrue1* and ND derivation

$$\frac{\frac{}{True} \quad True \quad P}{P = True} \text{iff1}$$

Note that 0 assumptions were discharged.
Isabelle rule $P \implies P = \text{True}$.

Universal Quantification

$$\forall P = (P = (\lambda x. True))$$

- Rule *all*

$$P(x)$$

$$\frac{}{\forall P} \text{ all}$$

Universal Quantification

$$\forall P = (P = (\lambda x. True))$$

- Rule *all*

$$P(x)$$

$$\frac{}{P = \lambda x. True} all$$

Universal Quantification

$$\forall P = (P = (\lambda x. True))$$

- Rule *all* and **ND** derivation

$$\frac{\frac{P(x)}{P(x) = True} \text{eqTrueI}}{P = \lambda x. True} \text{ext}$$

Inherits the side condition of *ext*: x must not occur freely in the derivation of $P(x)$.

Isabelle rule $(!!x. P(x)) \implies !x. P(x)$.

Example Illustrating Side Condition

$$\frac{\frac{[r(x)]^1}{r(x) \rightarrow r(x)} \rightarrow -/{}^1}{\forall x. r(x) \rightarrow r(x)} \text{all}$$

Why is this correct?

Example Illustrating Side Condition

$$\frac{\frac{[r(x)]^1}{r(x) \rightarrow r(x)} \rightarrow -I^1}{\forall x. r(x) \rightarrow r(x)} a//$$

Why is this correct? Let's do it without using *a//* explicitly:

$$\frac{\frac{\frac{[r(x)]^2}{r(x) \rightarrow r(x)} \rightarrow -I^2}{(r(x) \rightarrow r(x)) = True} eq TrueI}{\lambda x. (r(x) \rightarrow r(x)) = \lambda x. True} ext$$

The side condition is respected.

Universal Quantification (Cont.)

- Rule *spec* (recall $\forall P$ means $\forall x.Px$)

$$\forall P$$

$$\frac{\quad}{P(t)} \text{ } spec$$

Universal Quantification (Cont.)

- Rule *spec* (recall $\forall P$ means $\forall x.Px$)

$$P = \lambda x. True$$

$$\frac{}{P(t)} \text{ spec}$$

Universal Quantification (Cont.)

- Rule *spec* (recall $\forall P$ means $\forall x.Px$) and ND derivation

$$\frac{\frac{P = \lambda x.True}{P(t) = True} \text{fun_cong}}{P(t)} \text{eqTrueE}$$

Isabelle rule !x::'a. P(x) ==> P(x).

Note: Need universal quantification to reason about *False* (since $False = (\forall P.P)$).

False

False = $(\forall P.P)$ $(= \forall(\lambda P.P))$

- Falsel:

False

$False = (\forall P.P) \quad (= \forall(\lambda P.P))$

- Falsel: No rule!
- Rule *FalseE*

$$\frac{False}{P} \text{ FalseE}$$

False

$$\textit{False} = (\forall P.P) \quad (= \forall(\lambda P.P))$$

- Falsel: No rule!
- Rule *FalseE*

$$\frac{\forall P.P}{P} \textit{FalseE}$$

False

$False = (\forall P.P) \quad (= \forall(\lambda P.P))$

- Falsel: No rule!
- Rule *FalseE* and ND derivation

$$\frac{\forall P.P}{P} \text{spec}$$

Isabelle rule False ==> P.

False (Cont.)

- Rule *False_neq_True*

$$\textit{False} = \textit{True}$$

$$\frac{}{P} \textit{False_neq_True}$$

False (Cont.)

- Rule *False_neq_True* and ND derivation

$$\frac{\frac{False = True}{False} \text{eqTrueE}}{P} \text{FalseE}$$

Isabelle rule `False=True ==> P.`

- Similar:

$$\frac{True = False}{P} \text{True_neq_False}$$

Negation

$$\neg P = P \rightarrow \text{False}$$

- Rule *notI*

$$\frac{\begin{array}{c} [P] \\ \vdots \\ \text{False} \end{array}}{\neg P} \text{notI}$$

Negation

$$\neg P = P \rightarrow \text{False}$$

- Rule *notI*

$$\frac{\begin{array}{c} [P] \\ \vdots \\ \text{False} \end{array}}{P \rightarrow \text{False}} \text{notI}$$

Negation

$$\neg P = P \rightarrow False$$

- Rule *notI* and ND derivation

$$\frac{\begin{array}{c} [P] \\ \vdots \\ False \end{array}}{P \rightarrow False} \text{impl}$$

Isabelle rule $(P \implies False) \implies \sim P$.

Negation (2)

- Rule *notE*

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

Negation (2)

- Rule *notE*

$$\frac{P \rightarrow \text{False} \quad P}{R} \text{notE}$$

Negation (2)

- Rule *notE* and ND derivation

$$\frac{\frac{P \rightarrow False \quad P}{False} mp}{R} FalseE$$

Isabelle rule $[| \sim P; P |] ==> R.$

Negation (3)

- Rule *True_Not_False*

$$\frac{}{\neg(\textit{True} = \textit{False})} \textit{True_Not_False}$$

Negation (3)

- Rule *True_Not_False*

$$\frac{}{(True = False) \rightarrow False} \text{True_Not_False}$$

Negation (3)

- Rule *True_Not_False* and ND derivation

$$\frac{\frac{[True = False]^1}{False} \text{True_neq_False}}{(True = False) \rightarrow False} \text{notI}^1$$

Isabelle rule `True ~ = False`.

Existential Quantification

$$\exists P = P(\epsilon x.P(x))$$

- Rule *existsI*

$$\frac{P(x)}{\exists P} \text{ existsI}$$

Existential Quantification

$$\exists P = P(\epsilon x.P(x))$$

- Rule *existsI*

$$\frac{P(x)}{P(\epsilon x.P(x))} \text{ existsI}$$

Existential Quantification

$$\exists P = P(\epsilon x.P(x))$$

- Rule *existsI* and ND derivation

$$\frac{P(x)}{P(\epsilon x.P(x))} \text{ *selectI* }$$

Isabelle rule $P(x) \implies ?x :: 'a.P(x).$

Existential Quantification (Cont.)

- Rule *existsE*

$$\begin{array}{c} [P(x)] \\ \vdots \\ Q \end{array}$$

$$\frac{\exists P}{Q} \text{---} \text{existsE}$$

Inherits side condition from *allI* (just like in **FOL**).

Isabelle rule $[| \ ?x.P(x) ; \ !!x.P(x) \ ==> \ Q \ |] \ ==> \ Q.$

Existential Quantification (Cont.)

- Rule *existsE*

$$\begin{array}{c} [P(x)] \\ \vdots \\ Q \end{array}$$

$$\frac{P(\epsilon x.P(x))}{Q} \text{---} \text{existsE}$$

Inherits side condition from *a///* (just like in **FOL**).

Isabelle rule $[| \ ?x.P(x) ; \ !!x.P(x) \ ==> \ Q \ |] \ ==> \ Q.$

Existential Quantification (Cont.)

- Rule *existsE* and ND derivation

$$\begin{array}{c}
 [P(x)] \\
 \vdots \\
 Q \\
 \hline
 P(x) \rightarrow Q \quad \text{impl}^1 \\
 \hline
 \forall x.(P(x) \rightarrow Q) \quad \text{all} \\
 \hline
 P(\epsilon x.P(x)) \quad P(\epsilon x.P(x)) \rightarrow Q \quad \text{spec} \\
 \hline
 Q \quad \text{mp}
 \end{array}$$

Inherits side condition from *all* (just like in **FOL**).

Isabelle rule $[| \ ?x.P(x) ; \ !!x.P(x) \implies Q \ |] \implies Q$.

Conjunction

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

- Rule *conjI*

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

Conjunction

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

- Rule *conjI*

$$P$$
$$Q$$

$$\frac{}{\forall R. (P \rightarrow Q \rightarrow R) \rightarrow R} \text{conjI}$$

Conjunction

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

- Rule *conjI* and ND derivation

$$\begin{array}{c}
 \frac{[P \rightarrow Q \rightarrow R]^1 \quad P}{Q \rightarrow R} \text{mp} \quad \frac{Q}{R} \text{mp} \\
 \frac{R}{(P \rightarrow Q \rightarrow R) \rightarrow R} \text{impl}^1 \\
 \frac{(P \rightarrow Q \rightarrow R) \rightarrow R}{\forall R. (P \rightarrow Q \rightarrow R) \rightarrow R} \text{all}
 \end{array}$$

Isabelle rule `[| P; Q |] ==> P & Q.`

Conjunction (Cont.)

- Rule *conjEL*

$$P \wedge Q$$

$$\frac{}{P} \text{conjEL}$$

Conjunction (Cont.)

- Rule *conjEL*

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

$$\frac{}{P} \text{conjEL}$$

Conjunction (Cont.)

- Rule *conjEL* and ND derivation

$$\frac{\frac{\forall R.(P \rightarrow Q \rightarrow R) \rightarrow R}{(P \rightarrow Q \rightarrow P) \rightarrow P} \textit{spec} \quad \frac{\frac{[P]^1}{Q \rightarrow P} \textit{impl}}{P \rightarrow Q \rightarrow P} \textit{impl}^1}{P} \textit{mp}$$

Isabelle rule $P \ \& \ Q \ ==> \ P$.

Conjunction (Cont.)

- $P \wedge Q \implies Q$ (*conjER*)

Conjunction (Cont.)

- $P \wedge Q \Longrightarrow Q$ (*conjER*)
- $\llbracket P \wedge Q; \llbracket P; Q \rrbracket \Longrightarrow R \rrbracket \Longrightarrow R$ (*conjE*) (rule analogous to *disjE*)

Disjunction

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

- Rule *disjI*L

$$P$$

$$\frac{}{P \vee Q} \text{disjI}L$$

Disjunction

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

- Rule *disjI*L

$$P$$

$$\frac{}{\forall R. (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R} \text{disjI}L$$

Disjunction

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

- Rule *disjI*L and ND derivation

$$\begin{array}{c}
 \dfrac{[P \rightarrow R]^1 \quad P}{R} \textit{mp} \\
 \dfrac{\quad R}{(Q \rightarrow R) \rightarrow R} \textit{impl} \\
 \dfrac{\quad (Q \rightarrow R) \rightarrow R}{(P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R} \textit{impl}^1 \\
 \dfrac{\quad (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R}{\forall R. (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R} \textit{all}
 \end{array}$$

Isabelle rule $P \implies P \vee Q$.

Disjunction (Cont.)

- $Q \implies P \vee Q$ (*disjIR*) similar
- Rule *disjE*

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

Disjunction (Cont.)

- $Q \implies P \vee Q$ (*disjIR*) similar
- Rule *disjE*

$$\begin{array}{c}
 \forall R. (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R \\
 \begin{array}{ccc}
 & \begin{array}{c} [P] \\ \vdots \\ R \end{array} & \begin{array}{c} [Q] \\ \vdots \\ R \end{array} \\
 \hline
 & R & \text{disjE}
 \end{array}
 \end{array}$$

Disjunction (Cont.)

- $Q \implies P \vee Q$ (*disjIR*) similar
- Rule *disjE* and ND derivation

$$\begin{array}{c}
 \frac{\forall R. (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R}{(P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R} \text{spec} \quad \frac{\frac{[P]}{\vdots R} \quad P \rightarrow R}{R} \text{impl mp} \quad \frac{\frac{[Q]}{\vdots R} \quad Q \rightarrow R}{R} \text{impl mp} \\
 \hline
 (Q \rightarrow R) \rightarrow R
 \end{array}$$

Isabelle rule `[| P | Q; P ==> R; Q ==> R |] ==> R.`

Disjunction (Cont.)

- $Q \implies P \vee Q$ (*disjIR*) similar
- Rule *disjE* and ND derivation

$$\begin{array}{c}
 \frac{\forall R. (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R}{(P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R} \text{spec} \quad \frac{\frac{[P] \vdots R}{P \rightarrow R} \text{impl} \quad \frac{[Q] \vdots R}{Q \rightarrow R} \text{impl}}{(Q \rightarrow R) \rightarrow R} \text{mp} \quad \frac{(P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R \quad (Q \rightarrow R) \rightarrow R}{R} \text{mp}
 \end{array}$$

Isabelle rule `[| P | Q; P ==> R; Q ==> R |] ==> R.`

- $P \vee \neg P$ (*excluded middle*). Follows using *tof*.

Miscellaneous Definitions

See [HOL.thy](#)!

Typical example (if-then-else):

$$\begin{aligned} \text{If} = \lambda\phi^{bool}xy.\epsilon z. \quad & (\phi = \text{True} \rightarrow z = x) \\ & \wedge \quad (\phi = \text{False} \rightarrow z = y) \end{aligned}$$

The way rules are derived should now be clear. E.g.,

$$\frac{P = \text{True}}{\text{If } P \ x \ y = x} \qquad \frac{P = \text{False}}{\text{If } P \ x \ y = y}$$

Summary on Deriving Rules

HOL is very powerful in terms of what we can represent/derive:

- All well-known inference rules can be derived.
- Other “logical” syntax (e.g. **if-then-else**) can be defined.
- Rich theories can be obtained by a method we see **next lecture**.

Mathematics and Software Engineering in HOL

In coming weeks, we will see how Isabelle/HOL can be used as **foundation** for mathematics and software engineering.

Outline:

- The central method for making HOL scale up:
conservative extensions (< 1 week)
- How the different parts of mathematics are encoded in the
Isabelle/HOL library (several weeks)
- How software systems are embedded in Isabelle/HOL
(several weeks)

Outlook on Mathematics

After some historical background, we will look at how central parts of mathematics are encoded as Isabelle/HOL theories:

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Outlook on Software Engineering

Some weeks from now, we will look at case studies of how HOL can be applied in software engineering, i.e. how software systems can be **embedded** in Isabelle/HOL:

- Foundations, functional languages and denotational semantics
- Imperative languages, **Hoare logic**
- **Z** and data-refinement, **CSP** and process-refinement
- Object-oriented languages (Java-Light . . .)

Of the last three items, we want to treat only one in depth, depending on the audience's preferences.

Conservative Extensions: Motivation

But first, conservative extensions.

Stage of our course before studying HOL:

- fairly small theories,
- “intuitive” models, (e.g. *naïve set theory*),
- but *inconsistent* (due to foundational problems).

How can we use HOL to

- reason about a reasonably *large* part of mathematics and software engineering;
- *prevent* inconsistencies?

What Is Needed for Scaling up?

Several well-known structuring mechanisms are known.

Modularization: Isabelle supports **class** polymorphism and **theories**, however not **parametrized** theories.

Reuse: Isabelle supports **libraries** and **retrieval utilities**.

Safe, well-understood integration mechanisms: Isabelle supports **conservative theory extensions**.

Topic of **next** lecture.

More Detailed Explanations

Natural Deduction Derivation

We present most of those proofs by giving a **derivation tree** for it, but sometimes, we also give an Isabelle proof script.

Note also the **mix of syntaxes**.

Z, CSP

Z and CSP are specification languages. CSP stands for **communicating sequential processes**.

Conservative Theory Extensions

Outline

In the **previous lecture**, we have derived all well-known inference rules. There is now the need to scale up. Today we look at **conservative theory extensions**, an important method for this purpose.

Outline

In the **previous lecture**, we have derived all well-known inference rules. There is now the need to scale up. Today we look at **conservative theory extensions**, an important method for this purpose.

In the weeks to come, we will look at how mathematics is encoded in the Isabelle/HOL library.

Conservative Theory Extensions: Basics

Some definitions [GM93, Hué]

Definition 1 (theory):

A (syntactic) **theory** T is a triple (\mathcal{B}, Σ, A) , where \mathcal{B} is a **type signature**, Σ a **signature** and A and a set of **axioms**.

Definition 2 (theory extension):

A theory $T' = (\mathcal{B}', \Sigma', A')$ is an **extension** of a theory $T = (\mathcal{B}, \Sigma, A)$ iff $\mathcal{B} \subseteq \mathcal{B}'$ and $\Sigma \subseteq \Sigma'$ and $A \subseteq A'$.

Definitions (Cont.)

Definition 3 (conservative extension):

A theory extension $T' = (\mathcal{B}', \Sigma', A')$ of a theory $T = (\mathcal{B}, \Sigma, A)$ is **conservative** iff for the set of **derivable formulas** Th we have

$$Th(T) = Th(T') \mid_{\Sigma},$$

where \mid_{Σ} filters away all formulas not belonging to Σ .

Counterexample:

$$\frac{}{\forall f^{\alpha \rightarrow \alpha}. Y f = f (Y f)} \text{fix}$$

Consistency Preserved

Corollary 1 (consistency):

If T' is a conservative extension of T , then

$$\textit{False} \notin Th(T) \Rightarrow \textit{False} \notin Th(T').$$

Syntactic Schemata for Conservative Extensions

- Constant definition
- Type definition
- Constant specification
- Type specification

Will look at first two schemata now.

For the other two see [GM93].

Constant Definition

Definition 4 (constant definition):

A theory extension $T' = (\mathcal{B}', \Sigma', A')$ of a theory $T = (\mathcal{B}, \Sigma, A)$ is a **constant definition**, iff

- $\mathcal{B}' = \mathcal{B}$ and $\Sigma' = \Sigma \cup \{c : \tau\}$, where $c \notin \text{dom}(\Sigma)$;
- $A' = A \cup \{c = E\}$;
- E does not contain c and is closed;
- no subterm of E has a type containing a type variable that is not contained in the type of c .

Constant Definitions Are Conservative

Lemma 1 (constant definitions):

Constant definitions are conservative.

Proof Sketch:

- $Th(T) \subseteq Th(T') \mid_{\Sigma}$: trivial.
- $Th(T) \supseteq Th(T') \mid_{\Sigma}$: let π' be a proof for $\phi \in Th(T') \mid_{\Sigma}$. We unfold any subterm in π' that contains c via $c = E$ into π . Then π must be a proof in T , implying $\phi \in Th(T)$.

Side Conditions

Where are those **side conditions** needed? What goes wrong?

Side Conditions

Where are those *side conditions* needed? What goes wrong?
Very simple example: Let $E \equiv \exists x^\alpha y^\alpha. x \neq y$ and suppose σ is a type *inhabited* by only one term, and τ is a type *inhabited* by at least two terms. Then we would have:

$$\begin{aligned} c = c & \quad \text{holds by } \textit{refl} \\ \implies (\exists x^\sigma y^\sigma. x \neq y) &= (\exists x^\tau y^\tau. x \neq y) \\ \implies \textit{False} &= \textit{True} \\ \implies \textit{False} \end{aligned}$$

This explains *definition of True*. Other (standard) example later.

Constant Definition: Examples

Definitions of *True*, *False*, \wedge , \vee , \forall . . .

Here the **original** Isabelle syntax (**Ex_def** **changed**)

```

True_def:    "True      == ((%x::bool. x) = (%x. x))"
All_def:     "All(P)    == (P = (%x. True))"
Ex_def:      "Ex(P)     == P (SOME x. P x)"
False_def:   "False     == (!P. P)"
not_def:     "~ P       == P-->False"
and_def:     "P & Q      == !R. (P-->Q-->R) --> R"
or_def:      "P | Q      == !R. (P-->R) --> (Q-->R)
                                         --> R"

```

More Constant Definitions in Isabelle

Function application (Let), if-then-else, **unique existence**:

consts

Let :: [**'a**, **'a** => **'b**] => **'b**

If :: [**bool**, **'a**, **'a**] => **'a**

defs

Let_def "Let s f == f(s)"

if_def "If P x y == @z::**'a**. (P=True-->z=x) &
(P=False-->z=y)"

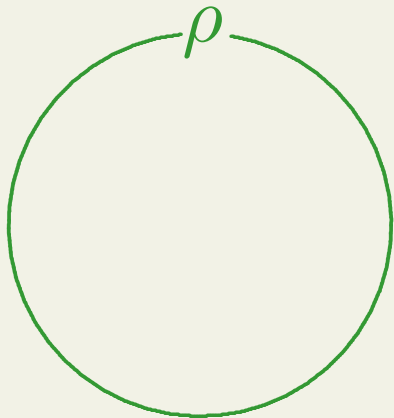
Ex1_def "Ex1(P) == ?x. P(x) & (!y. P(y) --> y=x)"

Recall: => is **function type arrow**; also recall **[] syntax**.

Type Definitions

Type definitions, explained intuitively: we have

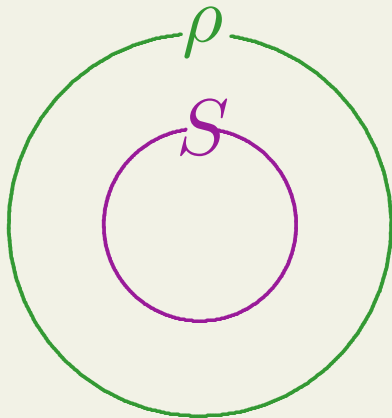
- an existing type ρ ;



Type Definitions

Type definitions, explained intuitively: we have

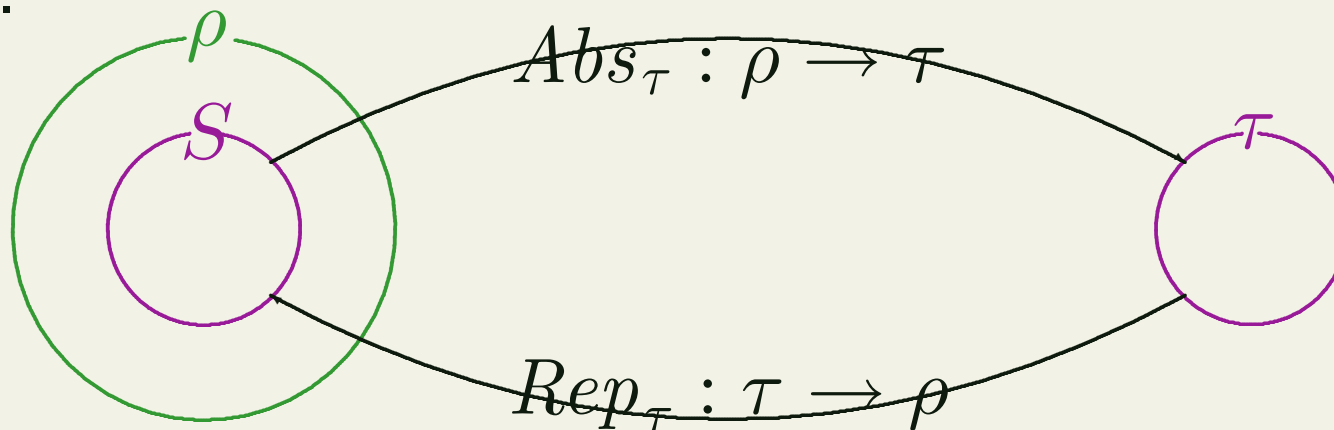
- an existing type ρ ;
- a predicate $S : \rho \rightarrow bool$, defining a non-empty “subset” of ρ ;



Type Definitions

Type definitions, explained intuitively: we have

- an existing type ρ ;
- a predicate $S : \rho \rightarrow \text{bool}$, defining a non-empty “subset” of ρ ;
- axioms stating an isomorphism between S and the new type τ .



Type Definition: Definition

Definition 5 (type definition):

Assume a theory $T = (\mathcal{B}, \Sigma, A)$ and a type ρ and a term S such that $\Sigma \vdash S : \rho \rightarrow \text{bool}$.

A theory extension $T' = (\mathcal{B}', \Sigma', A')$ of T is a **type definition** for type τ (where τ fresh), iff

$$\begin{aligned}\mathcal{B}' &= \mathcal{B} \uplus \{\tau\}, \\ \Sigma' &= \Sigma \cup \{Abs_\tau : \rho \rightarrow \tau, Rep_\tau : \tau \rightarrow \rho\} \\ A' &= A \cup \{\forall x. Abs_\tau(Rep_\tau x) = x, \\ &\quad \forall x. S x \rightarrow Rep_\tau(Abs_\tau x) = x\}\end{aligned}$$

Proof obligation $\exists x. S x$ can be proven inside HOL!

Type Definitions Are Conservative

Lemma 2 (type definitions):

Type definitions are conservative.

Proof see [GM93, pp.230].

HOL Is Rich Enough!

This may seem fishy: if a new type is always **isomorphic** to a **subset** of an **existing type**, how is this construction going to lead to a “rich” collection of types for large-scale applications?

HOL Is Rich Enough!

This may seem fishy: if a new type is always **isomorphic** to a **subset** of an **existing type**, how is this construction going to lead to a “rich” collection of types for large-scale applications?

But in fact, due to *ind* and \rightarrow , the types in HOL are already very rich.

HOL Is Rich Enough!

This may seem fishy: if a new type is always **isomorphic** to a **subset** of an **existing type**, how is this construction going to lead to a “rich” collection of types for large-scale applications?

But in fact, due to *ind* and \rightarrow , the types in HOL are already very rich.

We now give three examples to convince you.

Example: Typed Sets

General scheme,

$$\begin{aligned}\mathcal{B}' &= \mathcal{B} \uplus \{\tau\}, \\ \Sigma' &= \Sigma \cup \left\{ \begin{array}{l} Abs_\tau : \rho \longrightarrow \tau, \\ Rep_\tau : \tau \longrightarrow \rho \end{array} \right\} \\ A' &= A \cup \left\{ \begin{array}{l} \forall x. Abs_\tau (Rep_\tau x) = x, \\ \forall x. S x \longrightarrow Rep_\tau (Abs_\tau x) = x \end{array} \right\}\end{aligned}$$

Example: Typed Sets

General scheme, substituting $\rho \equiv \alpha \rightarrow \text{bool}$ (α is any **type variable**),

$$\mathcal{B}' = \mathcal{B} \uplus \{\tau\},$$

$$\Sigma' = \Sigma \cup \{ \text{Abs}_\tau : (\alpha \rightarrow \text{bool}) \rightarrow \tau, \\ \text{Rep}_\tau : \tau \rightarrow (\alpha \rightarrow \text{bool}) \}$$

$$A' = A \cup \{ \forall x. \text{Abs}_\tau (\text{Rep}_\tau x) = x, \\ \forall x. S x \rightarrow \text{Rep}_\tau (\text{Abs}_\tau x) = x \}$$

Example: Typed Sets

General scheme, substituting $\rho \equiv \alpha \rightarrow bool$ (α is any **type variable**), $\tau \equiv \alpha \text{ set}$ (or *set*),

$$\mathcal{B}' = \mathcal{B} \uplus \{set\},$$

$$\Sigma' = \Sigma \cup \{Abs_{set} : (\alpha \rightarrow bool) \rightarrow \alpha \text{ set}, \\ Rep_{set} : \alpha \text{ set} \rightarrow (\alpha \rightarrow bool)\}$$

$$A' = A \cup \{\forall x. Abs_{set}(Rep_{set} x) = x, \\ \forall x. S x \rightarrow Rep_{set}(Abs_{set} x) = x\}$$

Example: Typed Sets

General scheme, substituting $\rho \equiv \alpha \rightarrow bool$ (α is any **type variable**), $\tau \equiv \alpha \text{ set}$ (or *set*), $S \equiv \lambda x^{\alpha \rightarrow bool}. True$

$$\mathcal{B}' = \mathcal{B} \uplus \{set\},$$

$$\Sigma' = \Sigma \cup \{Abs_{set} : (\alpha \rightarrow bool) \rightarrow \alpha \text{ set}, \\ Rep_{set} : \alpha \text{ set} \rightarrow (\alpha \rightarrow bool)\}$$

$$A' = A \cup \{\forall x. Abs_{set}(Rep_{set} x) = x, \\ \forall x. True \rightarrow Rep_{set}(Abs_{set} x) = x\}$$

Example: Typed Sets

General scheme, substituting $\rho \equiv \alpha \rightarrow bool$ (α is any **type variable**), $\tau \equiv \alpha \text{ set}$ (or **set**), $S \equiv \lambda x^{\alpha \rightarrow bool}. True$

$$\mathcal{B}' = \mathcal{B} \uplus \{set\},$$

$$\Sigma' = \Sigma \cup \{Abs_{set} : (\alpha \rightarrow bool) \rightarrow \alpha \text{ set}, \\ Rep_{set} : \alpha \text{ set} \rightarrow (\alpha \rightarrow bool)\}$$

$$A' = A \cup \{\forall x. Abs_{set}(Rep_{set} x) = x, \\ \forall x. Rep_{set}(Abs_{set} x) = x\}$$

Simplification since $S \equiv \lambda x. True$. **Proof obligation:** $(\exists x. Sx)$ trivial since $(\exists x. True) = True$. **Inhabitation** crucial!

Sets: Remarks

Any function $r : \tau \rightarrow \text{bool}$ can be interpreted as a set of τ ; r is called **characteristic** function. That's what $Abs_{set} r$ does; Abs_{set} is a wrapper saying “interpret r as set”.

Sets: Remarks

Any function $r : \tau \rightarrow \text{bool}$ can be interpreted as a set of τ ; r is called **characteristic** function. That's what $\text{Abs}_{\text{set}} r$ does; Abs_{set} is a wrapper saying “interpret r as set”.
 $S \equiv \lambda x. \text{True}$ and so S is **trivial** in this case.

More Constants for Sets

For convenient use of sets, we define more constants:

$$\{x \mid f\ x\} = \textit{Collect}\ f = \textit{Abs}_{\textit{set}}\ f$$

$$x \in A = (\textit{Rep}_{\textit{set}}\ A)\ x$$

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

⋮

Consistent set theory adequate for most of mathematics and computer science !

Here, sets are just an **example** to demonstrate type definitions. **Later** we study them for their own sake.

Example: Pairs

Consider type $\alpha \rightarrow \beta \rightarrow bool$. We can regard a term $f : \alpha \rightarrow \beta \rightarrow bool$ as a representation of the pair (a, b) , where $a : \alpha$ and $b : \beta$, iff $f\ x\ y$ is true exactly for $x = a$ and $y = b$. Observe:

- For given a and b , there is **exactly one** such f (namely, $\lambda x^\alpha y^\beta. x = a \wedge y = b$).
- Some functions of type $\alpha \rightarrow \beta \rightarrow bool$ represent pairs and others don't (e.g., the function $\lambda xy. True$ does not represent a pair). The ones that do are exactly the ones that have the form $\lambda x^\alpha y^\beta. x = a \wedge y = b$, **for some** a and b .

Type Definition for Pairs

This gives rise to a type definition where S is non-trivial:

$$\rho \equiv \alpha \rightarrow \beta \rightarrow \text{bool}$$

$$S \equiv \lambda f^{\alpha \rightarrow \beta \rightarrow \text{bool}}. \exists ab. f = \lambda x^\alpha y^\beta. x = a \wedge y = b$$

$$\tau \equiv \alpha \times \beta \quad (\times \text{ infix})$$

It is convenient to define a constant `Pair_Rep` (not to be confused with Rep_\times) as $\lambda a^\alpha b^\beta. \lambda x^\alpha y^\beta. x = a \wedge y = b$. Then $\text{Pair_Rep } a b = \lambda x^\alpha y^\beta. x = a \wedge y = b$.

Now in Isabelle

Isabelle provides a special syntax for type definitions:

$$\begin{array}{l} \text{typedef } (T) \\ \quad \langle \textit{typevars} \rangle \text{ ``} T' \text{''} \langle \textit{fixity} \rangle \\ \quad = \text{ ``} \{x.\phi\} \text{''} \end{array}$$

Now in Isabelle

Isabelle provides a special syntax for type definitions:

$$\begin{aligned} &\text{typedef } (T) \\ &\quad \langle \text{typevars} \rangle \text{ } "T'" \langle \text{fixity} \rangle \\ &\quad = \text{ } "\{x.\phi\}" \end{aligned}$$

How is this linked to our **scheme**:

- the new type is **called** T' ;
- ρ is the type of x (**inferred**);
- S is $\lambda x.\phi$;
- **constants** Abs_T and Rep_T are automatically generated.

Isabelle Syntax for Pair Example

```
constdefs
```

```
Pair_Rep :: ['a, 'b] => ['a, 'b] => bool
```

```
"Pair_Rep == (%a b. %x y. x=a & y=b)"
```

```
typedef (Prod)
```

```
('a, 'b) "*" (infixr 20) =
```

```
"{f.?a b. f=Pair_Rep(a::'a)(b::'b)}"
```

The keyword `constdefs` introduces a constant definition.

The definition and use of `Pair_Rep` is for convenience.

There are “two names” `*` and `Prod`.

See `Product_Type.thy`.

Example: Sums

An element of (α, β) **sum** is either $Inl\ a$ where $a : \alpha$ or $Inr\ b$ where $b : \beta$.

So think of $Inl\ a$ and $Inr\ b$ as syntactic objects that we want to represent.

Consider type $\alpha \rightarrow \beta \rightarrow bool \rightarrow bool$. We can regard $f : \alpha \rightarrow \beta \rightarrow bool \rightarrow bool$ as a

representation of . . .	iff $f\ x\ y\ i$ is true for . . .
$Inl\ a$	$x = a$, y arbitrary, and $i = True$
$Inr\ b$	x arbitrary, $y = b$, and $i = False$.

Similar to **pairs**.

Isabelle Syntax for Sum Example

consts

```
Inl_Rep :: ['a, 'a, 'b, bool] => bool
```

```
"Inl_Rep == (%a. %x y p. x=a & p)"
```

```
Inr_Rep :: ['b, 'a, 'b, bool] => bool
```

```
"Inr_Rep == (%b. %x y p. y=b & ~p)"
```

typedef (Sum)

```
('a, 'b) "+" =
```

```
"{f. (?a. f = Inl_Rep(a::'a)) |  
      (?b. f = Inr_Rep(b::'b))}"
```

See [Sum_Type.thy](#).

How would you define a type even based on nat?

Summary on Conservative Extensions

We have seen two schemata:

- **Constant definition:** new constant must be defined using old constants. No **recursion**! Subtle **side conditions** concerning types.
- **Type definition:** new type must be isomorphic to a “subset” S of an existing type ρ . Not possible to define any type that is “structurally” richer than the types one already has. But HOL is **rich enough**. ►

More Detailed Explanations

Axioms or Rules

Isabelle-technically, those axioms are `thm`'s, and they may include Isabelle's metalevel implication \implies . For this reason, it is not required to mention `rules` explicitly.

But speaking more generally about HOL, not just its Isabelle implementation, one should better say “rules” here, i.e., objects with a horizontal line and zero or more `formulas` above the line and one `formula` below the line.

Derivable Formulas

The derivable formulas are terms of type *bool* derivable using the inference rules of HOL. We write $Th(T)$ for the derivable formulas of a theory T .

Closed Terms

A term is **closed** or **ground** if it does not contain any **free** variables.

Definition of *True* Is Type-Closed

True is defined as $\lambda x^{Bool}.x = \lambda x.x$ and not $\lambda x^\alpha.x = \lambda x.x$. The definition must be type-closed.

$\exists!$

We have never used **unique** existential quantification ($\exists!$) before.

$\exists!x_1, \dots, x_n. \phi(x_1, \dots, x_n)$ is defined as

$\exists x_1, \dots, x_n. \phi(x_1, \dots, x_n) \wedge (\forall y_1, \dots, y_n. \phi(y_1, \dots, y_n) \rightarrow x_1 = y_1 \wedge \dots \wedge x_n = y_n).$

Note that in general $\exists!x. (\exists!y. \phi)$ is not the same as $\exists!xy. \phi$.

Fixpoint Combinator

Given a function $f : \alpha \rightarrow \alpha$, a **fixpoint** of f is a term t such that $f t = t$. Now Y is supposed to be a fixpoint combinator, i.e., for any function f , the term $Y f$ should be a fixpoint of f . This is what the rule

$$\frac{}{\forall f^{\alpha \rightarrow \alpha}. Y f = f (Y f)} \text{fix}$$

says. Consider the example $f \equiv \neg$. Then the axiom allows us to infer $Y(\neg) = \neg(Y(\neg))$, and it is easy to derive *False* from this. This axiom is a standard example of a **non-conservative** extension of a theory.

It is not surprising that this goes wrong: Not every function has a fixpoint, so there cannot be a combinator returning a fixpoint of any function.

Nevertheless, fixpoints are important and must be realized in some way,

as we will see later.

Side Conditions

By **side conditions** we mean

- E does not contain c and is closed;
- no subterm of E has a type containing a type variable that is not contained in the type of c ;

in **the definition**.

The second condition also has a name: one says that the definition must be **type-closed**.

The notion of **having a type** is defined by the **type assignment calculus**. Since E is required to be closed, all variables occurring in E must be λ -bound, and so the type of those variables is given by the **type superscripts**.

Domains of Σ , Γ

The **domain** of Σ , denoted $dom(\Sigma)$, is $\{c \mid c : A \in \Sigma \text{ for some } A\}$.

Likewise, the **domain** of Γ , denoted $dom(\Gamma)$, is $\{x \mid x : A \in \Gamma \text{ for some } A\}$.

Note the **abuse of notation**.

constdefs

In Isabelle theory files, `consts` is the keyword preceding a sequence of constant declarations (i.e., this is where the Σ is defined), and `defs` is the keyword preceding the axioms that define these constants (i.e., this is where the A is defined).

`constdefs` combines the two, i.e. it allows for a sequence of both constant declarations and definitions. When the `constdefs` syntax is used to define a constant c , then the identifier c_def is generated automatically. E.g.

```
constdefs
  id :: "'a => 'a"
  "id == %x. x"
```

will bind `id_def` to $id \equiv \lambda x.x$.

A Predicate a Set?

Although a set is formally a different object than a predicate, it is standard to interpret a predicate a set: the set of terms for which the predicate returns true. We have done this [before](#).

S

Here, S is any “predicate”, i.e., term of type $\rho \rightarrow bool$, not necessarily a constant.

Fresh τ

The type constructor τ must not occur in \mathcal{B} .

What Is τ ?

A type definition is supposed to define a **type constructor** (where the arity and fixity are indicated in some way). We abuse notation here: we use τ to denote a type constructor, but also the type obtained by applying the type constructor to a vector of different **type variables** (as many as the type constructor requires).

So think of τ as either being a **type constructor** or a “**generic**” **type** (just a type constructor being applied to type variables).

We do the same in examples.



The symbol \uplus denotes disjoint union, so the expression $A \uplus B$ is well-formed only when A and B have no elements in common. One thus uses this notation to indicate this fact.

What Are Abs_τ and Rep_τ ?

Of course we are giving a schematic definition here, so any letters we use are metanotation.

Notice that Abs_τ and Rep_τ stand for new **constants**. For any new type τ to be defined, two such constants must be added to the signature to provide a generic way of obtaining terms of the new type. Since the new type is isomorphic to the “subset” S , whose members are of type ρ , one can say that Abs_τ and Rep_τ provide a type conversion between (the subset S of) ρ and τ .

So we have a new type τ , and we can obtain members of the new type by applying Abs_τ to a term t of type ρ for which $S t$ holds.

Isomorphism

The formulas

$$\forall x. Abs_{\tau}(Rep_{\tau} x) = x$$

$$\forall x. S x \rightarrow Rep_{\tau}(Abs_{\tau} x) = x$$

state that the “set” S and the new type τ are isomorphic. Note that Abs_{τ} should not be applied to a term not in “set” S . Therefore we have the premise $S x$ in the above equation.

Note also that S could be the “trivial filter” $\lambda x. True$. In this case, Abs_{τ} and Rep_{τ} would provide an isomorphism between the entire type ρ and the new type τ .

Proof Obligation

We have said [previously](#) that S should be a **non-empty** “subset” of τ . Therefore it must be proven that $\exists x. S\ x$. This is related to the [semantics](#).

Whenever a type definition is introduced in Isabelle, the proof obligation must be shown inside Isabelle/HOL. Isabelle provides the `typedef` syntax for type definitions, as we will see [later](#). Using this syntax, the “author” of a type definition can either explicitly provide a proof (see [Product_Type.thy](#)), or the proof is so easy that Isabelle can do it automatically (see [Sum_Type.thy](#)).

Concrete Symbols

set is a concrete type constructor symbol, since this is an example.

Inhabitation in the *set* Example

We have $S \equiv \lambda x^{\alpha \rightarrow \text{bool}}. \text{True}$, and so in $(\exists x. Sx)$, the variable x has type $\alpha \rightarrow \text{bool}$. The proposition $(\exists x. Sx)$ is true since the type $\alpha \rightarrow \text{bool}$ is inhabited, e.g. by the term $\lambda x^{\alpha}. \text{True}$ or $\lambda x^{\alpha}. \text{False}$.

Beware of a confusion: This does not mean that the new type $\alpha \text{ set}$, defined by this construction, is the type of **non-empty** sets. There is a term for the empty set: The empty set is the term $\text{Abs}_{\text{set}} (\lambda x. \text{False})$. Recall a **previous argument** for the importance of inhabitation.

Trivial S

We said that in the general formalism for defining a new type, there is a term S of type $\rho \rightarrow bool$ that defines a “subset” of a type ρ . In other words, it filters some terms from type ρ . Thus the idea that a predicate can be interpreted as a set is present in the general formalism for defining a new type.

Now we are talking about a particular example, the type αset . Having the idea “predicates are sets” in mind, one is tempted to think that in the particular example, S will take the role of defining particular sets, i.e., terms of type αset . This is not the case!

Rather, S is $\lambda x. True$ and hence trivial in this example. Moreover, in the example, ρ is $\alpha \rightarrow bool$, and any term r of type ρ defines a set whose elements are of type α ; $Abs_{set} r$ is that set.

Collect

We have seen *Collect* before in the theory file NSet.thy (naïve set theory).

Collect f is the set whose characteristic function is f . There is also a concrete (i.e., according to mathematical practice) syntax $\{x \mid f\ x\}$. It is called set comprehension. The correspondence between the HOAS *Collect* f and the concrete syntax $\{x \mid f\ x\}$ also makes it clear that set comprehension is a binding operator, as we learned some time ago.

Note also that *Collect* is the same as Abs_{set} here, so there is no need to have them as separate constants, and this is in fact what is done in the Isabelle theory file Set.thy: One only has *Collect*.

The file Set.thy should be contained in your Isabelle distribution. In Freiburg it is

```
/usr/local/isabelle/Isabelle/src/HOL/Set.thy
```

The \in -Sign

We define

$$x \in A = (Rep_{set} A) x$$

Since Rep_{set} has type $\alpha set \rightarrow (\alpha \rightarrow bool)$, this means that x is of type α and A is of type $(\alpha \rightarrow bool)$. Therefore \in is of type $\alpha \rightarrow (\alpha set) \rightarrow bool$ (but written *infix*).

In the Isabelle theory file `Set.thy`, you will indeed find that the constant `:` (Isabelle syntax for \in) has type $\alpha \rightarrow (\alpha set) \rightarrow bool$.

However, you will not find anything directly corresponding to Rep_{set} .

One can see that this setup is equivalent to the one we have here (which was presented like that for the sake of generality). There are two axioms

in `Set.thy`:

axioms

```
mem_Collect_eq [iff]: "(a : {x. P(x)}) = P(a)"
Collect_mem_eq [simp]: "{x. x:A} = A"
```

Let us translate these axioms using the [definitions](#).

$$a \in \{x \mid P x\} = P a \rightsquigarrow$$

$$a \in (\text{Collect } P) = P a \rightsquigarrow$$

$$a \in (\text{Abs}_{\text{set}} P) = P a \rightsquigarrow$$

$$\text{Rep}_{\text{set}}(\text{Abs}_{\text{set}} P) a = P a \rightsquigarrow \text{Rep}_{\text{set}}(\text{Abs}_{\text{set}} P) = P$$

The last step uses [extensionality](#).

Now the second one:

$$\begin{aligned}\{x \mid x \in A\} &= A \rightsquigarrow \\ \{x \mid (\text{Rep}_{\text{set}} A) x\} &= A \rightsquigarrow \\ \text{Collect}(\text{Rep}_{\text{set}} A) &= A\end{aligned}$$

Ignoring some universal quantifications (these are implicit in Isabelle), these are the **isomorphy axioms for *set***.

Consistent Set Theory

Typed set theory is a conservative extension of HOL and hence consistent.

Recall the problems with untyped set theory.

“Exactly one” Term

When we say that there is “exactly one” f , this is meant modulo equality in HOL. This means that e.g. $\lambda x^\alpha y^\beta. y = b \wedge x = a$ is also such a term since $(\lambda x^\alpha y^\beta. x = a \wedge y = b) = (\lambda x^\alpha y^\beta. y = b \wedge x = a)$ is derivable in HOL.

$$Rep_{\times}$$

Rep_{\times} would be the generic name for one of the two isomorphism-defining functions.

Since Rep_{\times} looks funny, the definition scheme for type definitions in Isabelle is such that it provides two names for a type, one if the type is used as such, and one for the purpose of generating the names of the isomorphism-defining functions.

Funny Iteration of λ 's

We write $\lambda a^\alpha b^\beta . \lambda x^\alpha y^\beta . x = a \wedge y = b$ rather than $\lambda a^\alpha b^\beta x^\alpha y^\beta . x = a \wedge y = b$ to emphasize the idea that one first applies *Pair_Rep* to a and b , and the result is a function representing a pair, which can then be applied to x and y .

Product_Type.thy

This file should be contained in your Isabelle distribution. In Freiburg it is

`/usr/local/isabelle/Isabelle/src/HOL/Product_Type.thy`

Sum Types

Idea of **sum** or **union** type: t is in the sum of τ and σ if t is either in τ or in σ . To do this formally in our **type system**, and also in the type system of functional programming languages like ML, t must be wrapped to signal if it is of type τ or of type σ .

For example, in ML one could define

$$\text{datatype } (\alpha, \beta) \text{ sum} = \text{Inl } \alpha \mid \text{Inr } \beta$$

So an element of (α, β) sum is either $\text{Inl } a$ where $a : \alpha$ or $\text{Inr } b$ where $b : \beta$.

Sum_Type.thy

This file should be contained in your Isabelle distribution. In Freiburg it is

`/usr/local/isabelle/Isabelle/src/HOL/Sum_Type.thy`

Defining even

Suppose we have a type `nat` and a constant `+` with the expected meaning. We want to define a type `even` of even numbers. What is an even number?

Defining even

Suppose we have a type `nat` and a constant `+` with the expected meaning. We want to define a type `even` of even numbers. What is an even number?

The following choice of S is adequate:

$$S \equiv \lambda x. \exists n. x = n + n$$

Using the Isabelle scheme, this would be

```
typedef (Even)  
  even = "{x. ?y. x=y+y}"
```

We could then go on by defining an operation `PLUS` on `even`, say as

follows:

```
constdefs
  PLUS :: [even, even] => even (infixl 56)
  PLUS_def "opPLUS ==
            %xy. Abs_Even (Rep_Even(x)+Rep_Even(x))"
```

Note that we chose to use names `even` and `Even`, but we could have used the same name twice as well.

Mathematics in the Isabelle/HOL Library: Introduction

Isabelle/HOL at Work

We have seen how the mechanism of conservative extensions works in principle.

For several lectures, we will now look at theories of the Isabelle/HOL library, all built by conservative extensions and modelling significant portions of mathematics.

Sets: The Basis of Principia Mathematica

Sets are ubiquitous in mathematics:

- 17th century: geometry can be reduced to numbers [Des16, vL16].
- 19th century: numbers can be reduced to sets [Can18, Pea18, Fre93, Fre03].
- 20th century: sets can be represented in logics [Zer19, Frä19, WR25, Göd31, Ber19, Chu40].

We call this the **Principia Mathematica Structure** [WR25].

The libraries of theorem provers follow this Principia Mathematica Structure — in reverse order!

The Roadmap

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes ►

More Detailed Explanations

Why in Reverse Order?

It is not surprising that the logical built-up of theorem prover is reversed w.r.t. to the historical development of mathematics and logics. Research usually starts from applications and the intuition and works its way back to the foundations.

Orders

The Roadmap

We are looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

The Roadmap

We are looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Three Order Classes

We first define a **syntactic class** ord. It is the class of types for which symbols $<$ and \leq exist.

Three Order Classes

We first define a **syntactic class** `ord`. It is the class of types for which symbols $<$ and \leq exist.

We then define two **axiomatic classes** `order` and `linorder` for which $<$ and \leq are required to have certain properties, that of being a **partial order**, or a **linear order**, resp.

Orders (in HOL.thy)

axclass

ord < type

consts

"op <" :: ['a::ord, 'a] => bool

"op <=" :: ['a::ord, 'a] => bool

constdefs

min :: "['a::ord, 'a] => 'a"

"min a b == (if a <= b then a else b)"

max :: "['a::ord, 'a] => 'a"

"max a b == (if a <= b then b else a)"

Recall `constdefs` syntax and note two uses of `<`.

Orders in HOL.thy (Cont.)

```
axclass order < ord
  order_refl      "x <= x"
  order_trans     "[| x <= y; y <= z |] ==> x <= z"
  order_antisym  "[| x <= y; y <= x |] ==> x = y"
  order_less_le  "x < y = (x <= y & x ~= y)"
%
axclass linorder < order
  linorder_linear "x <= y | y <= x"
```

Least Elements

In `Ord.thy`, `least elements` used to be defined as:

```
Least :: "('a::ord => bool) => 'a"
```

```
Least_def "Least P == @x. P(x) &  
  (ALL y. P(y) => x <= y)"
```

Now it is done without using the `Hilbert operator`.

Monotonicity

In `Ord.thy`, `monotonicity` used to be defined as:

```
mono      :: ['a::ord => 'b::ord] => bool
mono_def  "mono(f) ==
           (!A B. A <= B --> f(A) <= f(B))"
```

Now it is done using a completely different syntax, but one can still use `monotonicity` as before.

Some Theorems about Orders

<code>monoI</code>	$(\bigwedge AB. A \leq B \implies f A \leq f B) \implies \text{mono } f$
<code>monoD</code>	$\llbracket \text{mono } f; A \leq B \rrbracket \implies f A \leq f B$
<code>order_eq_refl</code>	$x = y \implies x \leq y$
<code>order_less_irrefl</code>	$\neg x < x$
<code>order_le_less</code>	$(x \leq y) = (x < y \vee x = y)$
<code>linorder_less_linear</code>	$x < y \vee x = y \vee y < x$
<code>linorder_neq_iff</code>	$(x \neq y) = (x < y \vee y < x)$
<code>min_same</code>	$\min x x = x$
<code>le_min_iff_conj</code>	$(z \leq \min x y) = (z \leq x \wedge z \leq y)$

Summary on Orders

Type classes are a structuring mechanism in Isabelle:

- **Syntactic classes** (e.g. $t :: \alpha :: \text{ord}$ as in Haskell [HHPW96]): merely a mechanism to structure visibility of operations.

Summary on Orders

Type classes are a structuring mechanism in Isabelle:

- **Syntactic classes** (e.g. $t :: \alpha :: \text{ord}$ as in Haskell [HHPW96]): merely a mechanism to structure visibility of operations.
- **Axiomatic classes** (e.g. $t :: \alpha :: \text{order}$): a mechanism for structuring semantic knowledge in types (foundation to be discussed later). ►

More Detailed Explanations

Where Are Orders Defined?

In previous versions of Isabelle, there used to be a theory file `Ord.thy`. Nowadays orders are defined in `HOL.thy`.

Different uses of $<$

The line

```
axclass order < ord
```

in the theory file states that `order` is a [subclass](#) of `ord`.

The line

```
"op <" :: ['a::ord, 'a] => bool ("(_ < _)" [50, 51] 50)
```

in the theory file declares a constant $<$ with a certain type.

`type` is the class containing all types. In [previous versions of Isabelle](#), it used to be called `term`.

Theorems of what?

In the rest of the course, we will mostly be dealing with Isabelle HOL, and so when we speak of a theorem, we usually mean an Isabelle theorem, i.e., a theorem in Isabelle's metalogic, what we also call a thm. Such theorems may contain the meta-level implication \implies and universal quantifier \bigwedge .

So they are not theorems within HOL. Logically, this is not a big deal as one switch between object and meta-level by the introduction and elimination rules for \rightarrow and \forall . But technically (for the proof procedures), it makes a difference.

To see a theorem displayed in Isabelle, simply type the name of the theorem followed by “;”.

Semantic Classes for Semantic Knowledge

The Isabelle type system records for any type variable what **class constraints** there are for this type variable. These class constraints may arise from the types of the constants used in an expression, or they may be given explicitly by the user in a goal. E.g. one might type

```
Goal "(x::'a::order)<y ==> x<=y";
```

to specify that x must be of a type in the type class `order`.

The axioms of an axiomatic class can only be applied if any constant declared in the axiomatic class (or a syntactic superclass) is applied to arguments of a type in the axiomatic class. E.g. `order_refl` can only be used to prove $y <= y$ if the type of y is in the type class `order`.

In this sense the type information (y is of type in class `order`) is semantic knowledge ($y <= y$ holds).

Sets

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Set.thy

```
theory Set = HOL:
typedec1 'a set
instance set :: (type) ord ..
consts
  "{}"      :: 'a set ("{}")
  UNIV      :: 'a set
  insert    :: ['a, 'a set] => 'a set
  Collect   :: ('a => bool) => 'a set
  "op :"    :: "'a => 'a set => bool"
```

Note that Collect and ":" correspond to Abs_{set} and Rep_{set} .

Set.thy: More Constant Declarations

```
Un, Int      :: ['a set, 'a set] => 'a set
Ball, Bex    :: ['a set, 'a => bool] => bool
UNION, INTER:: ['a set, 'a => 'b set] => 'b set
Union, Inter:: (('a set) set) => 'a set
Pow          :: 'a set => 'a set set
"image"      :: ['a => 'b, 'a set] => ('b set)
```

We use **old syntax** here but only since it is more concise.

Set.thy: Constant Definitions

```

empty_def:      "{} == {x. False}"
UNIV_def:       "UNIV == {x. True}"
Un_def:         "A Un B == {x. x:A | x:B}"
Int_def:        "A Int B == {x. x:A & x:B}"
insert_def:     "insert a B == {x. x=a} Un B"
Ball_def:       "Ball A P == ALL x. x:A --> P(x)"
Bex_def:        "Bex A P == EX x. x:A & P(x)"

```

Nice syntax:

```

{x, y, z}      for  insert x (insert y (insert z {}))
ALL x : A. Sx   for  Ball A S
EX x : A. Sx    for  Bex A S

```

Set.thy: Constant Definitions (2)

subset_def: " $A \leq B \iff \text{ALL } x:A. x:B$ "

Compl_def: " $- A \iff \{x. \sim x:A\}$ "

set_diff_def: " $A - B \iff \{x. x:A \ \& \ \sim x:B\}$ "

UNION_def: " $\text{UNION } A \ B \iff \{y. \text{EX } x:A. y: B(x)\}$ "

INTER_def: " $\text{INTER } A \ B \iff \{y. \text{ALL } x:A. y: B(x)\}$ "

Note use of \leq instead of \subseteq !

Nice syntax:

$\text{UN } x \in A. S \ x$ or $\bigcup_{x \in A}. S \ x$ for $\text{UNION } A \ S$

$\text{INT } x \in A. S \ x$ or $\bigcap_{x \in A}. S \ x$ for $\text{INTER } A \ S$

Set.thy: Constant Definitions (3)

Union_def: "Union $S == (\text{UN } x:S. x)$ "

Inter_def: "Inter $S == (\text{INT } x:S. x)$ "

Pow_def: "Pow $A == \{B. B \leq A\}$ "

image_def: " $f' A == \{y. \text{EX } x:A. y = f(x)\}$ "

Nice syntax:

$\bigcup S$ for Union S

$\bigcap S$ for Inter S

Some Theorems in Set.thy

CollectI	$P\ a \Longrightarrow a \in \{x.P\ x\}$
CollectD	$a \in \{x.P\ x\} \Longrightarrow P\ a$
set_ext	$(\bigwedge x.(x \in A) = (x \in B)) \Longrightarrow A = B$
subsetI	$(\bigwedge x.x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B$
eqset_imp_iff	$A = B \Longrightarrow (x \in A) = (x \in B)$
UNIV_I	$x \in \text{UNIV}$
subset_UNIV	$A \subseteq \text{UNIV}$
empty_subsetI	$\{\} \subseteq A$
Pow_iff	$(A \in \text{Pow}\ B) = (A \subseteq B)$
IntI	$\llbracket c \in A; c \in B \rrbracket \Longrightarrow c \in A \cap B$

More Theorems in Set.thy

insert_iff $(a \in \text{insert } b \ A) = (a = b \vee a \in A)$

image_Un $f'(A \cup B) = f'A \cup f'B$

Inter_lower $B \in A \implies \bigcap A \subseteq B$

Inter_greatest $(\bigwedge X. X \in A \implies C \subseteq X) \implies C \subseteq \bigcap A$

Summary on Sets

Rich and powerful set theory available in HOL:

- No problems with **consistency**
- **Weaker** than **ZFC** (since typed set-theory:) there is no “union of sets”; but: **complement-closed**
- Good **mechanical support** for many set tautologies
(**Fast_tac**, `fast_tac set_cs`, `fast_tac eq_cs`,
... `simp_tac set_ss` ...)
- Powerful basis for many problems in modeling ►

More Detailed Explanations

\leq instead of \subseteq

Sets are an instance of the type class `ord`, where the generic constant \leq is the subset relation in this particular case.

In fact, the subset relation is reflexive, transitive and anti-symmetric, and so sets are an instance of the `axiomatic class` `order`. This is non-obvious and must be proven, which is done not in `Set.thy` itself but in `Fun.thy`, [later](#). This is a technicality of Isabelle.

Union of Arbitrary Sets?

In typed set theory (what we have here in HOL), it is not possible to form the union of two sets of different type. This is in contrast to [ZFC](#).

Typed Sets Are Complement-Closed

The complement of a typed set A , i.e.

$$\{x \mid x \notin A\}$$

is again a set, whose type is the same as the type of A . In [ZFC](#), the complement construction is not generally allowed since it opens the door to [Russell's Paradox](#).

Functions

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders
- Sets
- **Functions**
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Fun.thy

The theory `Fun.thy` defines some important notions on functions, such as concatenation, the identity function, the image of a function, etc.

We look at it briefly.

Two Extracts from Fun.thy

Composition and the identity function:

```
constdefs
```

```
  id :: "'a => 'a"
```

```
"id == %x. x"
```

```
%
```

```
  comp :: "[ 'b => 'c, 'a => 'b, 'a ] => 'c"
```

```
"f o g == %x. f(g(x))"
```

Recall `constdefs` syntax.

Instantiating an Axiomatic Class

Sets are **partial orders**: `set` is an **instance** of the axiomatic class `order`.

For **some reason**, this is proven in `Fun.thy`.

```
instance set :: (type) order
  by (intro_classes,
      (assumption | rule subset_refl
        subset_trans subset_antisym psubset_eq)+)
```

- **Axiomatic classes** result in **proof obligations**.
- These are **discharged** whenever instance is stated.
- **Type-checking** has access to the established properties.

Conclusion of Orders, Sets, Functions

- Theory says: **conservative extensions** can be used to build consistent libraries.

Conclusion of Orders, Sets, Functions

- Theory says: **conservative extensions** can be used to build consistent libraries.
- **Sets** as **one important package** of Isabelle/HOL library:
 - Set theory is typed, but **very rich** and **powerfully supported**.

Conclusion of Orders, Sets, Functions

- Theory says: **conservative extensions** can be used to build consistent libraries.
- **Sets** as **one important package** of Isabelle/HOL library:
 - Set theory is typed, but **very rich** and **powerfully supported**.
 - Sets are instance of **ord** and **order** type class, demonstrates type classes as structuring mechanism in Isabelle.

Conclusion of Orders, Sets, Functions

- Theory says: **conservative extensions** can be used to build consistent libraries.
- **Sets** as **one important package** of Isabelle/HOL library:
 - Set theory is typed, but **very rich** and **powerfully supported**.
 - Sets are instance of **ord** and **order** type class, demonstrates type classes as structuring mechanism in Isabelle.
- Will see more examples: Isabelle/HOL contains some 10000 **thm**'s. ►

More Detailed Explanations

Fun.thy

This file should be contained in your Isabelle distribution. In Freiburg it is

```
/usr/local/isabelle/Isabelle/src/HOL/Fun.thy
```

Fun.thy builds on [Set.thy](#), and it is here that it is proven and used that sets are an instance of the type class order.

Proof Obligations

To claim that a type is an instance of an **axiomatic class**, it has to be proven that the axioms (in the case of order: `order_refl`, `order_trans`, `order_antisym`, and `order_less_le` are indeed fulfilled by that type.

Discharge Obligations

The Isabelle mechanism is such that the line

```
instance set :: (type) order
  by (intro_classes,
      (assumption | rule
       subset_refl subset_trans subset_antisym psubset_eq)+)
```

instructs Isabelle to prove the **axioms** using the previously proven **theorems** `subset_refl`, `subset_trans`, `subset_antisym`, and `psubset_eq`.

Background: Recursion, Induction, and Fixpoints

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Recursion Based on Set Theory

Current stage of our course:

- On the basis of conservative extensions, set theory can be built safely.

Recursion Based on Set Theory

Current stage of our course:

- On the basis of **conservative extensions**, **set theory** can be built safely.
- But: our mathematical world is still quite small and quite remote from computer science: we have no means of introducing recursive definitions (recursive programs, recursive set equations, . . .).

Recursion Based on Set Theory

Current stage of our course:

- On the basis of **conservative extensions**, **set theory** can be built safely.
- But: our mathematical world is still quite small and quite remote from computer science: we have no means of introducing recursive definitions (recursive programs, recursive set equations, . . .).

How can we benefit from set theory to introduce recursion?

Recursion and General Fixpoints

Naïve Approach: One could **axiomatize** fixpoint combinator Y as

$$\overline{Y = \lambda F. F(Y F)}^{\text{fix}}$$

This axiom is not a **constant definition**.

Then we could easily derive

$$\forall F^{\alpha \rightarrow \alpha}. Y F = F (Y F).$$

Recursion and General Fixpoints

Naïve Approach: One could **axiomatize** fixpoint combinator Y as

$$\overline{Y = \lambda F. F(Y F)}^{\text{fix}}$$

This axiom is not a **constant definition**.

Then we could easily derive

$$\forall F^{\alpha \rightarrow \alpha}. Y F = F (Y F).$$

- Why are we interested in Y ?
- What's the problem with such a definition?

Why Are We Interested in Y ?

First, why are we interested in **recursion** (solutions to recursive equations)?

Why Are We Interested in Y ?

First, why are we interested in **recursion** (solutions to recursive equations)?

- Recursively defined **functions** are solutions of such equations (example: *fac*).
- Inductively defined **sets** are solutions of such equations (example: *Fin* A , all finite subsets of A).

Why Are We Interested in Y ?

First, why are we interested in **recursion** (solutions to recursive equations)?

- Recursively defined **functions** are solutions of such equations (example: *fac*).
- Inductively defined **sets** are solutions of such equations (example: *Fin A*, all finite subsets of A).

We are interested in Y because it is the mother of all recursions. With Y , recursive axioms can be **converted** into constant definitions.

What's the Problem with such an Axiom?

Such a definition would lead to **inconsistency**.

This is not surprising because not all functions have a fixpoint.

Therefore we only consider **special forms** of fixpoint combinators.

We consider two approaches: **Least fixpoints** (Tarski) and **well-founded orderings**. ►

More Detailed Explanations

Axiom is not a Definition

The axiom

$$Y = \lambda F.F(Y F)$$

is not a **constant definition**, since Y occurs again on the right-hand side.

$$\forall F^{\alpha \rightarrow \alpha}. Y F = F (Y F)$$

In words, this says that $Y F$ is a fixpoint of F .

Recursive Equation

By a recursive equation, we mean an equation of the form

$$f = e$$

where f occurs in e . A fortiori, such an equation does not qualify as constant definition.

Converting Recursive Axioms

Any recursive function can be defined by an expression (functional) which is not itself recursive, but instead relies on the recursive equation defining Y .

Consider *fac* or *Fin A* as an example.

Least Fixpoints

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

First Approach: Least Fixpoints (Tarski)

- Restriction: F is not of arbitrary type $(\alpha \rightarrow \alpha)$ but of set type $(\alpha \text{ set} \rightarrow \alpha \text{ set})$.
- Instead of Y define lfp by an equation which is not recursive.
- lfp is fixpoint combinator, but only under additional condition that F is monotone, and: this is not obvious (requires non-trivial proof)!

This leads us towards recursion and induction.

Lfp.thy

Lfp = Product_Type +
constdefs

```
lfp :: ['a set => 'a set] => 'a set  
"lfp(f) == Inter({u. f(u) <= u})"
```

Note:

- \Rightarrow is function type arrow.
- \leq (\subseteq) is a partial order.
- Inter (\bigcap) gives a “minimum”: $\forall A \in S. (\bigcap S) \subseteq A$.

Lfp.thy

Lfp = Product_Type +
constdefs

```
lfp :: ['a set => 'a set] => 'a set  
"lfp(f) == Inter({u. f(u) <= u})"
```

Note:

- \Rightarrow is function type arrow.
- \leq (\subseteq) is a partial order.
- `Inter` (\cap) gives a “minimum”: $\forall A \in S. (\cap S) \subseteq A$.
- Definition of *lfp* is conservative. That’s fine. But is it a fixpoint combinator?

Tarski's Fixpoint Theorem

Theorem 1 (Tarski):

If f is **monotone**, then $\text{lfp } f = f (\text{lfp } f)$.

In Isabelle, the **theorem** is shown in **Lfp.ML** and called `lfp_unfold`.

We show the theorem using mathematical notation and a graphical illustration to help intuition.

The proof has four steps.

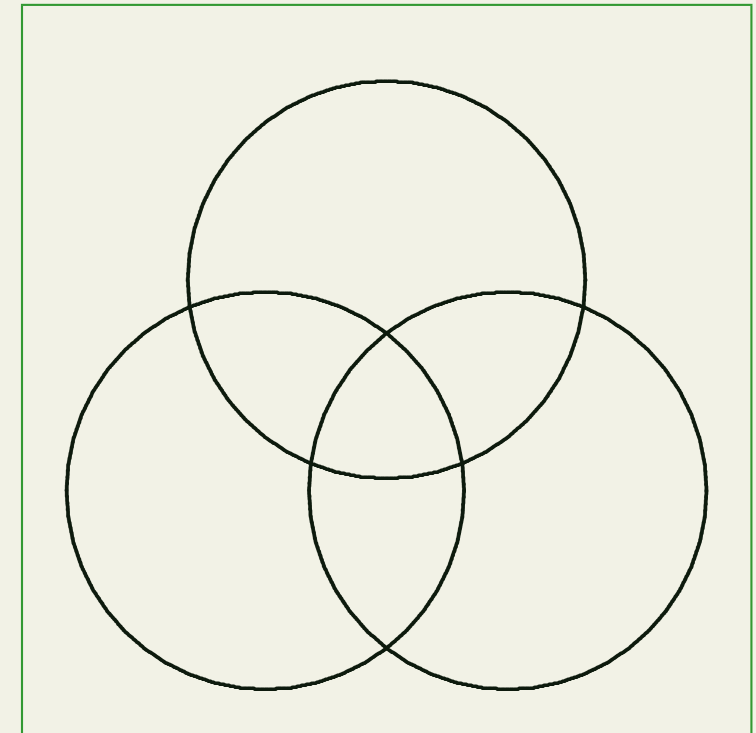
Tarski's Fixpoint Theorem (1)

Claim 1 (“*lfp* lower bound”): If $f \restriction A \subseteq A$ then $\text{lfp } f \subseteq A$.

Tarski's Fixpoint Theorem (1)

Claim 1 (“lfp lower bound”): If $f A \subseteq A$ then $\text{lfp } f \subseteq A$.

The **box** denotes “the set” α . The **three circles** denote the sets A for which $f A \subseteq A$.

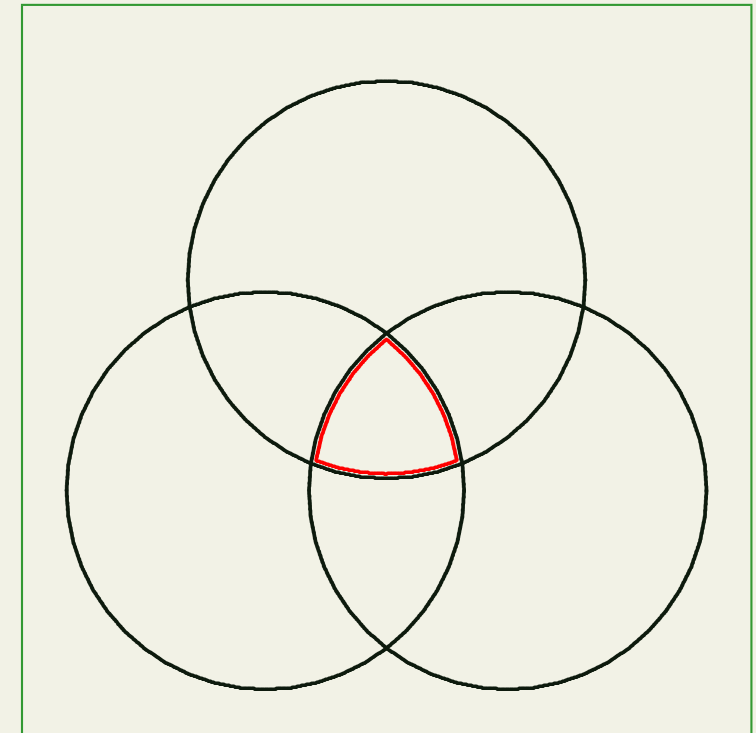


Tarski's Fixpoint Theorem (1)

Claim 1 (“*lfp* lower bound”): If $f A \subseteq A$ then $\text{lfp } f \subseteq A$.

The **box** denotes “the set” α . The **three circles** denote the sets A for which $f A \subseteq A$.

By definition, ***lfp* f** is the **intersection**.



Tarski's Fixpoint Theorem (1)

Claim 1 (“*lfp* lower bound”): If $f A \subseteq A$ then $\text{lfp } f \subseteq A$.

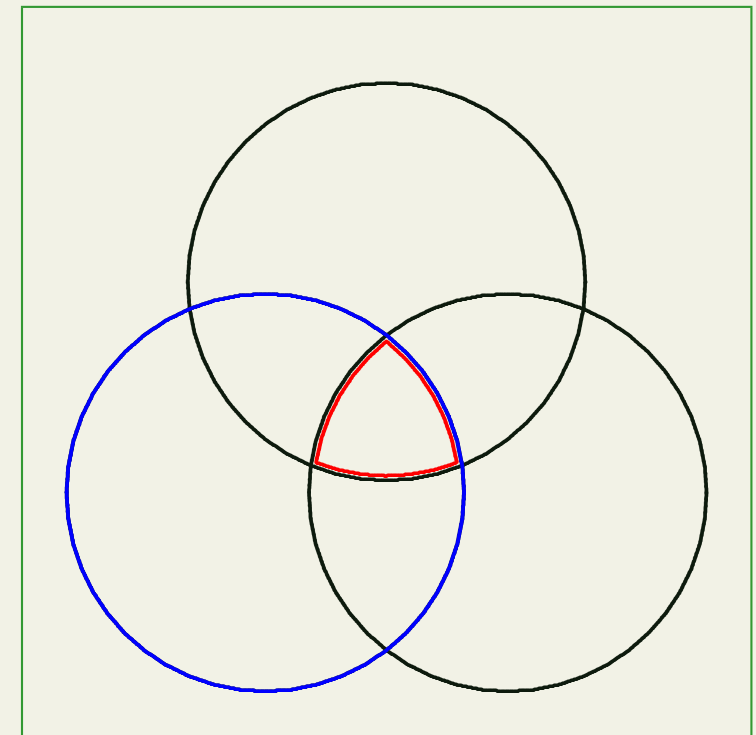
The **box** denotes “the set” α . The **three circles** denote the sets A for which $f A \subseteq A$.

By definition, ***lfp* f** is the **intersection**.

Pick an **A** for which $f A \subseteq A$.

Clearly, ***lfp* f** $\subseteq A$.

Or as **proof tree**.



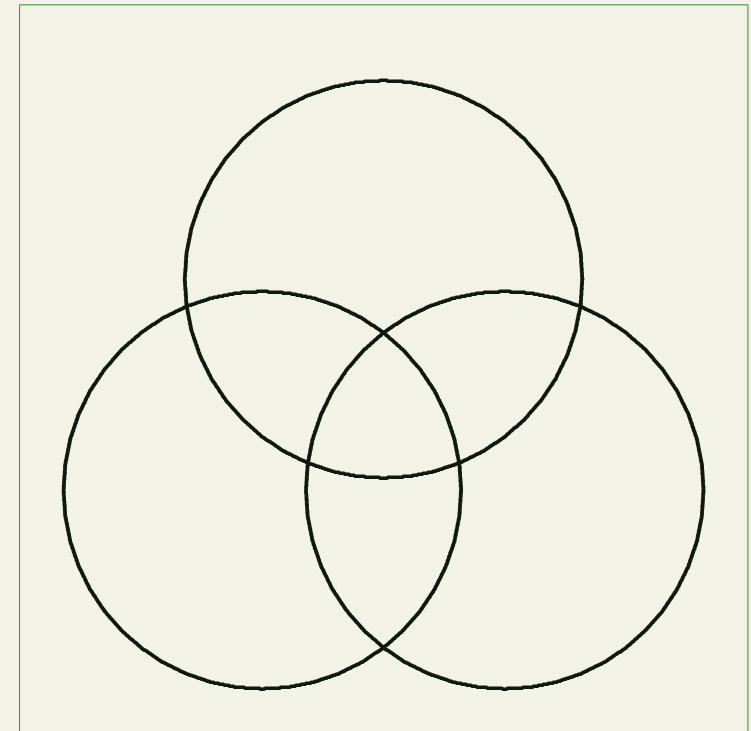
Tarski's Fixpoint Theorem (2)

Claim 2 (“*lfp* greatest”): For all A , if
for all U , $f U \subseteq U$ implies $A \subseteq U$, then $A \subseteq \text{lfp } f$.

Tarski's Fixpoint Theorem (2)

Claim 2 (“*lfp* greatest”): For all A , if
for all U , $f U \subseteq U$ implies $A \subseteq U$, then $A \subseteq \text{lfp } f$.

The three circles denote the sets U
for which $f U \subseteq U$.

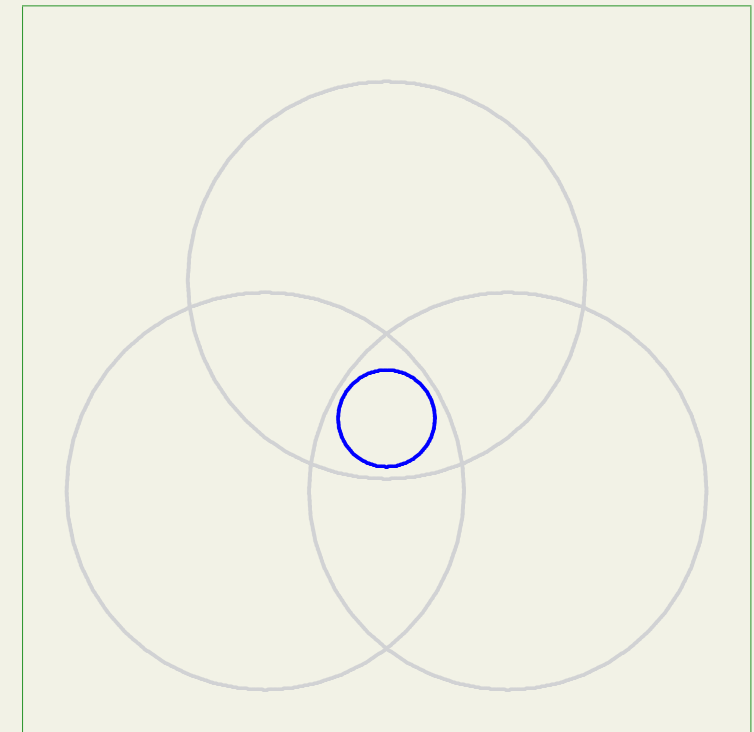


Tarski's Fixpoint Theorem (2)

Claim 2 (“*lfp* greatest”): For all A , if
for all U , $f U \subseteq U$ implies $A \subseteq U$, then $A \subseteq \text{lfp } f$.

The three circles denote the sets U
for which $f U \subseteq U$.

By hypothesis, $A \subseteq U$ for each U

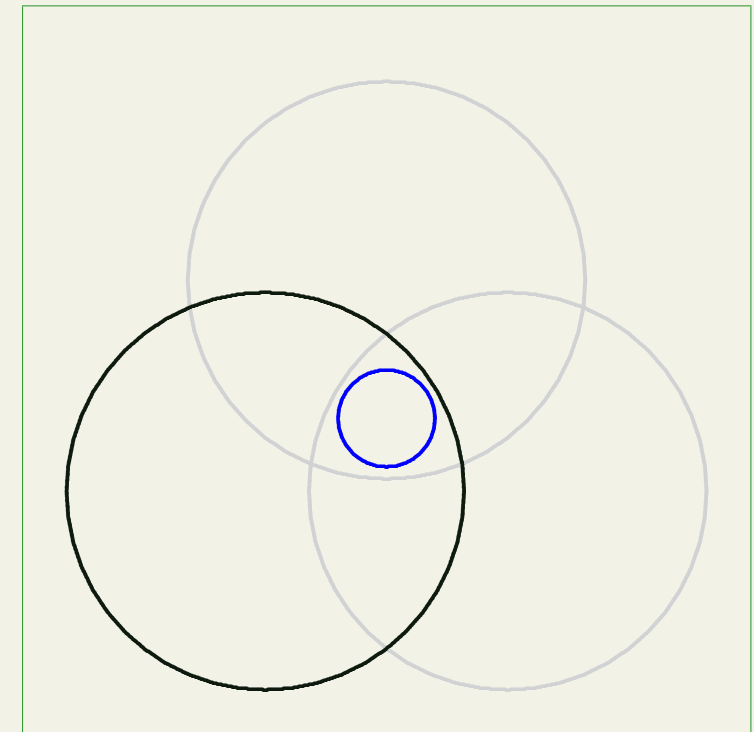


Tarski's Fixpoint Theorem (2)

Claim 2 (“*lfp* greatest”): For all A , if
for all U , $f U \subseteq U$ implies $A \subseteq U$, then $A \subseteq \text{lfp } f$.

The three circles denote the sets U
for which $f U \subseteq U$.

By hypothesis, $A \subseteq U$ for each U
(1st,

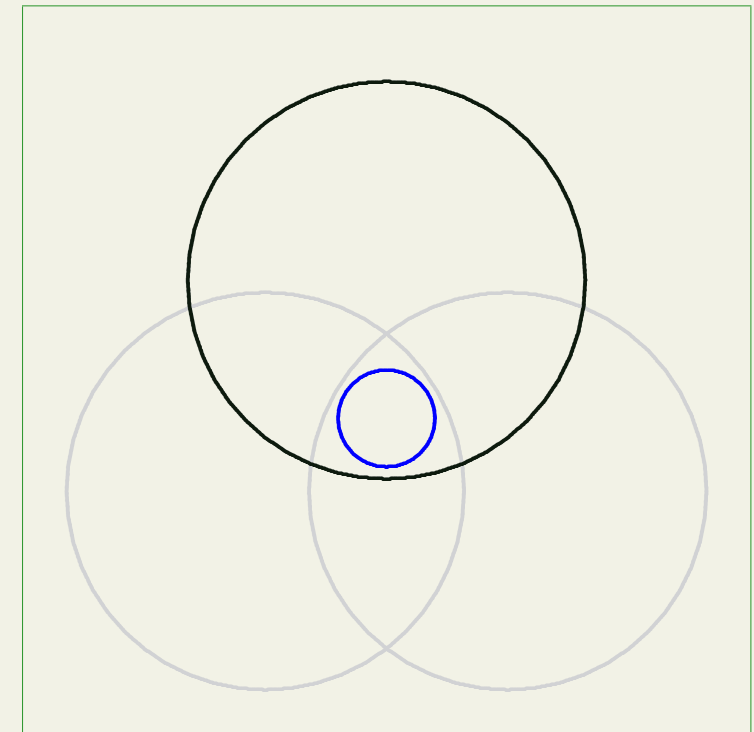


Tarski's Fixpoint Theorem (2)

Claim 2 (“*lfp* greatest”): For all A , if
for all U , $f U \subseteq U$ implies $A \subseteq U$, then $A \subseteq \text{lfp } f$.

The three circles denote the sets U
for which $f U \subseteq U$.

By hypothesis, $A \subseteq U$ for each U
(1st, 2nd,

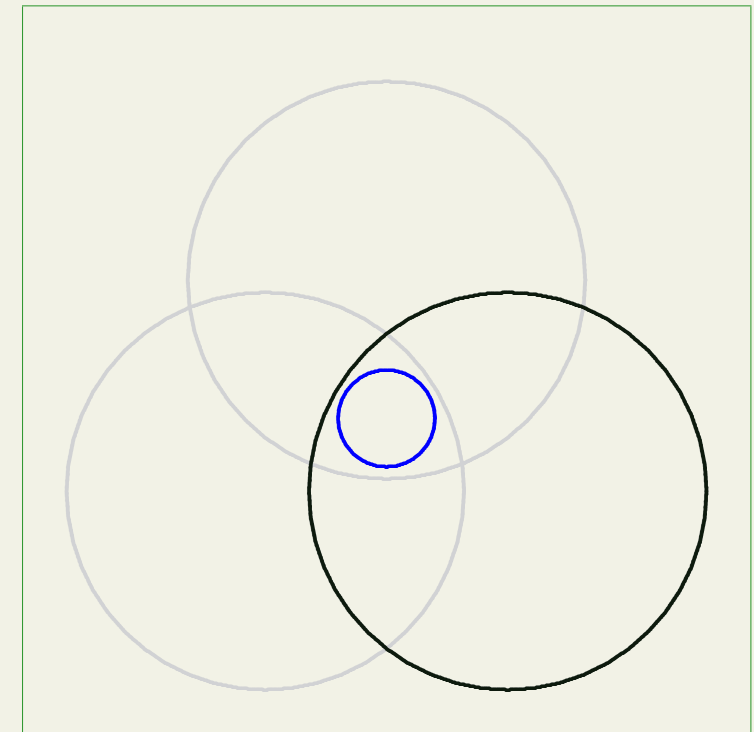


Tarski's Fixpoint Theorem (2)

Claim 2 (“*lfp* greatest”): For all A , if
for all U , $f U \subseteq U$ implies $A \subseteq U$, then $A \subseteq \text{lfp } f$.

The three circles denote the sets U
for which $f U \subseteq U$.

By hypothesis, $A \subseteq U$ for each U
(1st, 2nd, 3rd . . .).



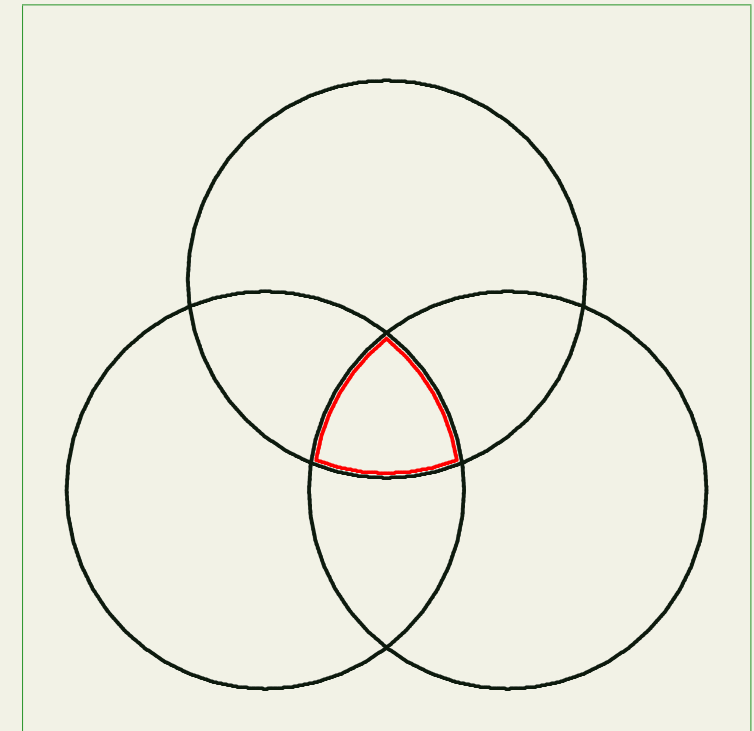
Tarski's Fixpoint Theorem (2)

Claim 2 (“*lfp* greatest”): For all A , if
for all U , $f U \subseteq U$ implies $A \subseteq U$, then $A \subseteq \text{lfp } f$.

The three circles denote the sets U
for which $f U \subseteq U$.

By hypothesis, $A \subseteq U$ for each U
(1st, 2nd, 3rd . . .).

By definition, $\text{lfp } f$ is the intersec-
tion.



Tarski's Fixpoint Theorem (2)

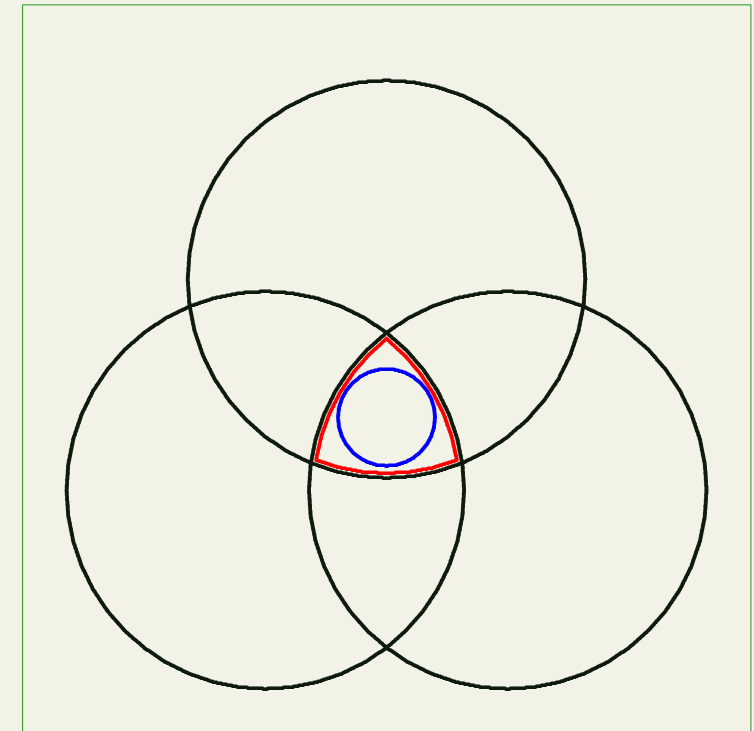
Claim 2 (“*lfp* greatest”): For all A , if
for all U , $f U \subseteq U$ implies $A \subseteq U$, then $A \subseteq \text{lfp } f$.

The three circles denote the sets U
for which $f U \subseteq U$.

By hypothesis, $A \subseteq U$ for each U
(1st, 2nd, 3rd . . .).

By definition, $\text{lfp } f$ is the intersec-
tion.

Clearly, $A \subseteq \text{lfp } f$.



Tarski's Fixpoint Theorem (2)

Claim 2 (“*lfp* greatest”): For all A , if
for all U , $f U \subseteq U$ implies $A \subseteq U$, then $A \subseteq \text{lfp } f$.

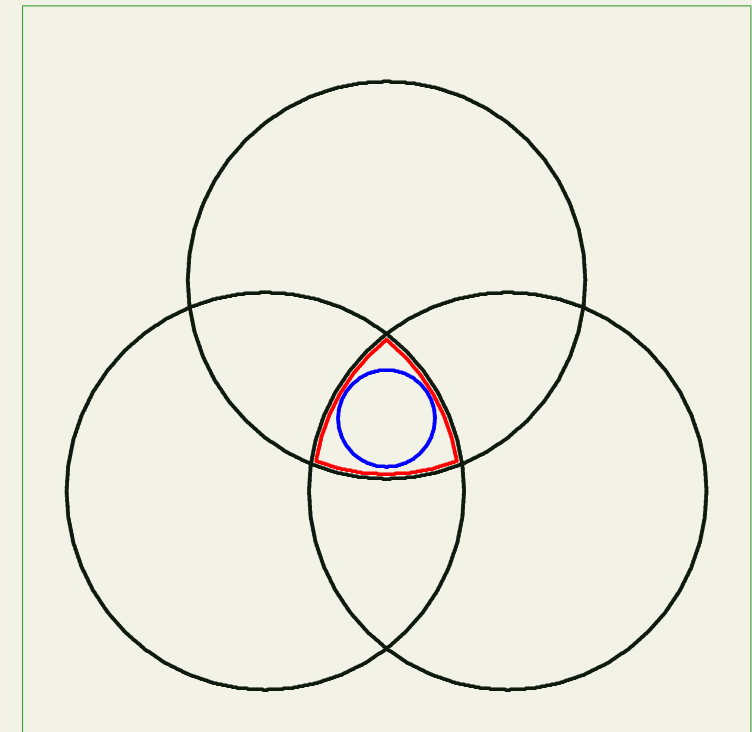
The three circles denote the sets U
for which $f U \subseteq U$.

By hypothesis, $A \subseteq U$ for each U
(1st, 2nd, 3rd . . .).

By definition, $\text{lfp } f$ is the intersec-
tion.

Clearly, $A \subseteq \text{lfp } f$.

Or as proof tree.



Tarski's Fixpoint Theorem (3)

Claim 3: If f is monotone then $f(\text{lfp } f) \subseteq \text{lfp } f$.

Tarski's Fixpoint Theorem (3)

Claim 3: If f is **monotone** then $f(\text{lfp } f) \subseteq \text{lfp } f$.

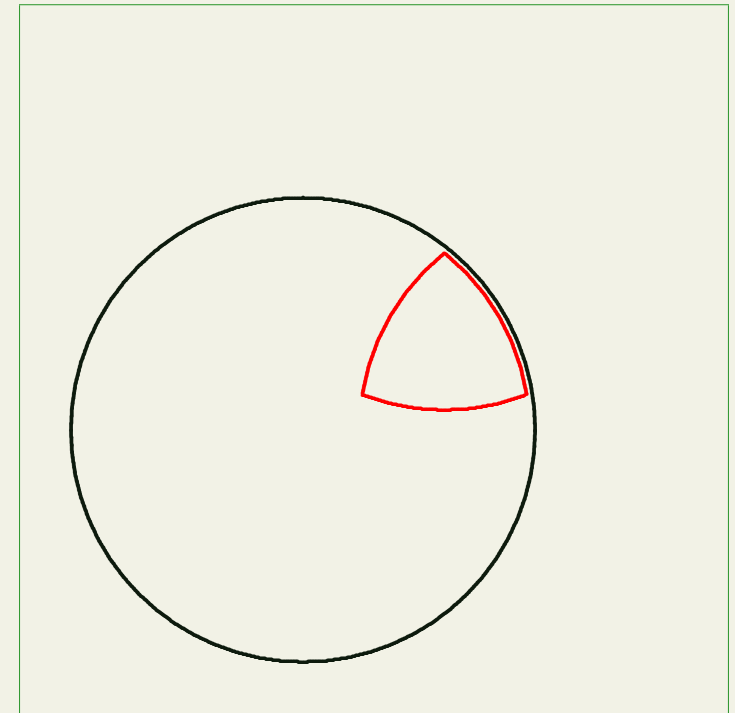
First show Claim 3*: $f U \subseteq U$ implies $f(\text{lfp } f) \subseteq U$.

Tarski's Fixpoint Theorem (3)

Claim 3: If f is **monotone** then $f(\text{lfp } f) \subseteq \text{lfp } f$.

First show Claim 3*: $f U \subseteq U$ implies $f(\text{lfp } f) \subseteq U$.

Let the circle be such a U . By **Claim 1**, $\text{lfp } f \subseteq U$.



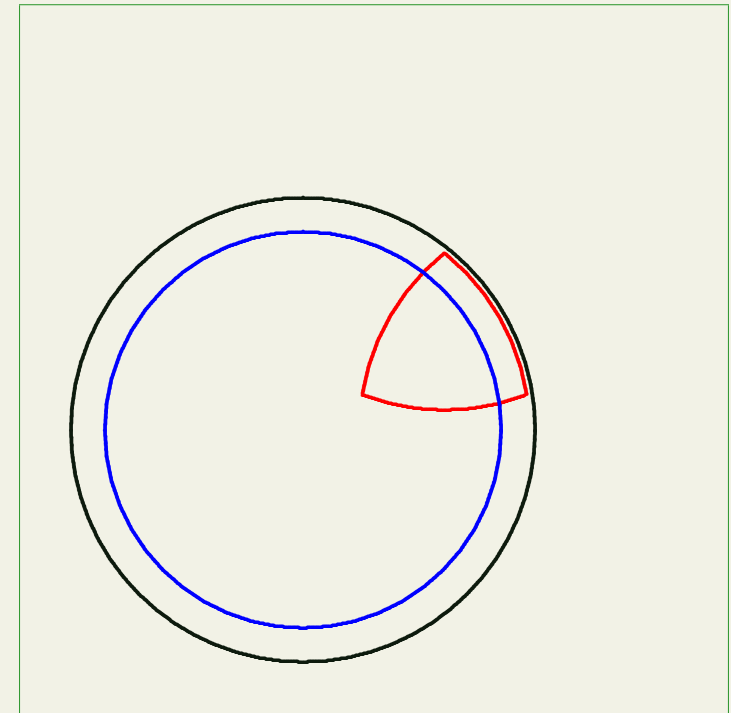
Tarski's Fixpoint Theorem (3)

Claim 3: If f is **monotone** then $f(\text{lfp } f) \subseteq \text{lfp } f$.

First show Claim 3*: $f U \subseteq U$ implies $f(\text{lfp } f) \subseteq U$.

Let the circle be such a U . By **Claim 1**, $\text{lfp } f \subseteq U$.

$f U \subseteq U$ (hypothesis).



Tarski's Fixpoint Theorem (3)

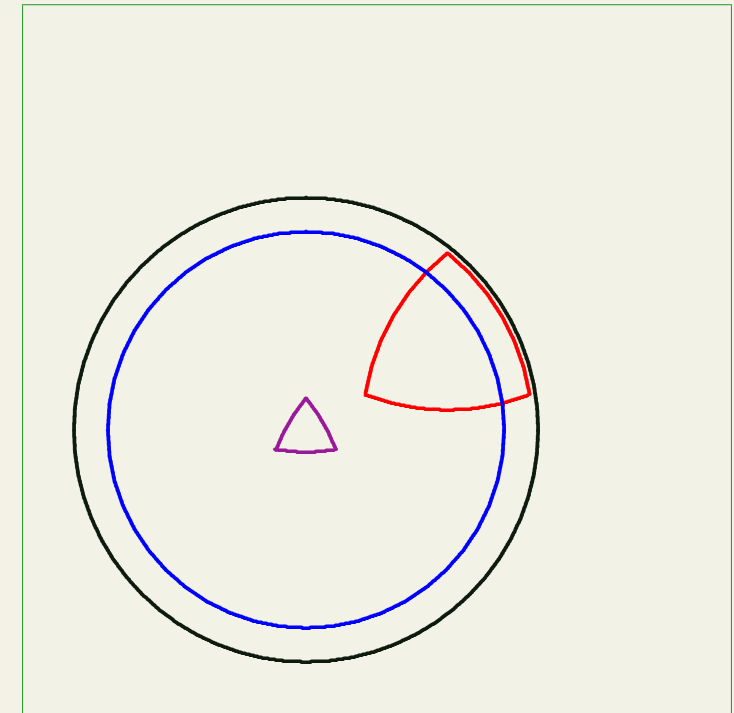
Claim 3: If f is **monotone** then $f(\text{lfp } f) \subseteq \text{lfp } f$.

First show Claim 3*: $f U \subseteq U$ implies $f(\text{lfp } f) \subseteq U$.

Let the circle be such a U . By Claim 1, $\text{lfp } f \subseteq U$.

$f U \subseteq U$ (hypothesis).

$f(\text{lfp } f) \subseteq f U$ (monotonicity).



Tarski's Fixpoint Theorem (3)

Claim 3: If f is **monotone** then $f(\text{lfp } f) \subseteq \text{lfp } f$.

First show Claim 3*: $f U \subseteq U$ implies $f(\text{lfp } f) \subseteq U$.

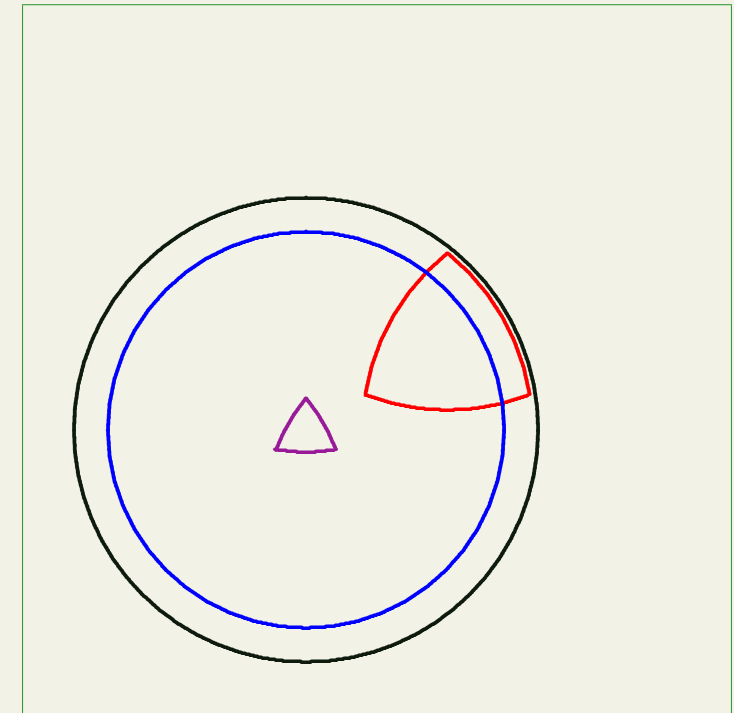
Let the circle be such a U . By Claim 1, $\text{lfp } f \subseteq U$.

$f U \subseteq U$ (hypothesis).

$f(\text{lfp } f) \subseteq f U$ (monotonicity).

$f(\text{lfp } f) \subseteq U$ (transitivity of \subseteq).

Claim 3* shown.



Tarski's Fixpoint Theorem (3)

Claim 3: If f is **monotone** then $f(\text{lfp } f) \subseteq \text{lfp } f$.

First show Claim 3*: $f U \subseteq U$ implies $f(\text{lfp } f) \subseteq U$.

Let the circle be such a U . By Claim 1, $\text{lfp } f \subseteq U$.

$f U \subseteq U$ (hypothesis).

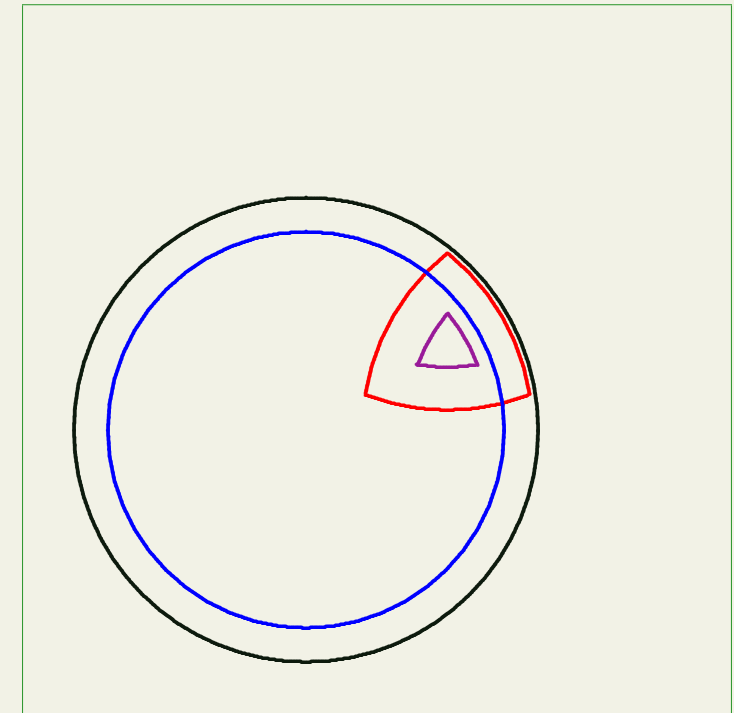
$f(\text{lfp } f) \subseteq f U$ (monotonicity).

$f(\text{lfp } f) \subseteq U$ (transitivity of \subseteq).

Claim 3* shown.

By Claim 2 (letting $A := f(\text{lfp } f)$),

$f(\text{lfp } f) \subseteq \text{lfp } f$.



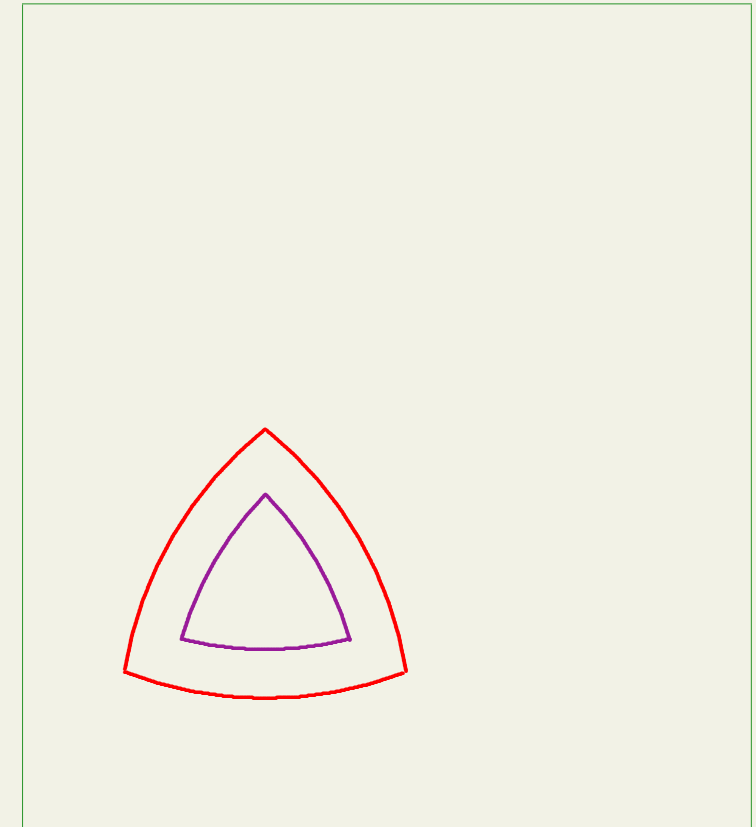
Tarski's Fixpoint Theorem (4)

Claim 4: If f is monotone then $\text{lfp } f \subseteq f(\text{lfp } f)$.

Tarski's Fixpoint Theorem (4)

Claim 4: If f is monotone then $\text{lfp } f \subseteq f(\text{lfp } f)$.

By Claim 3, $f(\text{lfp } f) \subseteq \text{lfp } f$.

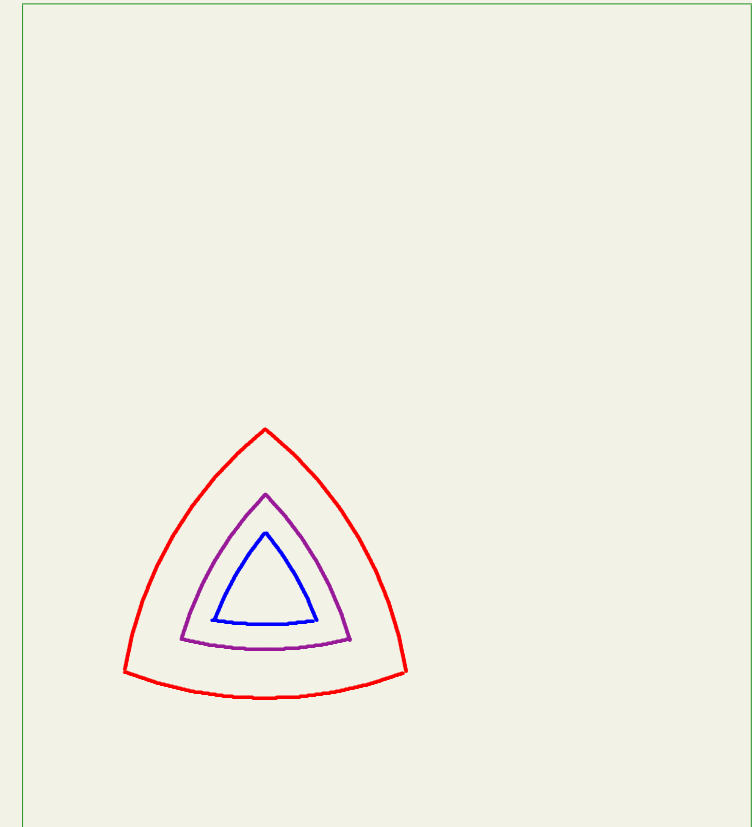


Tarski's Fixpoint Theorem (4)

Claim 4: If f is monotone then $\text{lfp } f \subseteq f(\text{lfp } f)$.

By Claim 3, $f(\text{lfp } f) \subseteq \text{lfp } f$.

By monotonicity, $f(f(\text{lfp } f)) \subseteq f(\text{lfp } f)$.



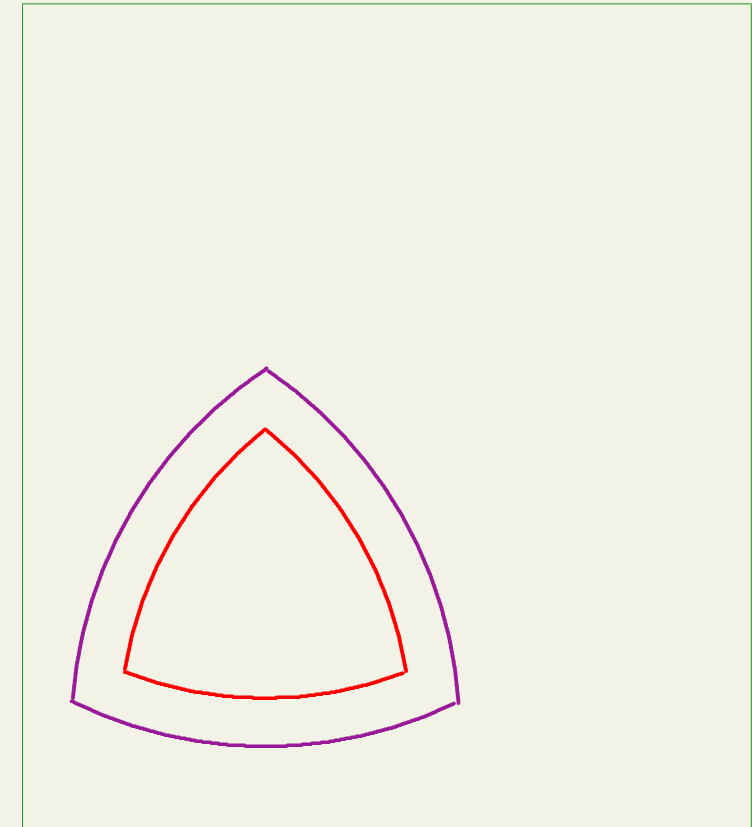
Tarski's Fixpoint Theorem (4)

Claim 4: If f is monotone then $\text{lfp } f \subseteq f(\text{lfp } f)$.

By Claim 3, $f(\text{lfp } f) \subseteq \text{lfp } f$.

By monotonicity, $f(f(\text{lfp } f)) \subseteq f(\text{lfp } f)$.

By Claim 1 (letting $A := f(\text{lfp } f)$),
 $\text{lfp } f \subseteq f(\text{lfp } f)$.



Tarski's Fixpoint Theorem (4)

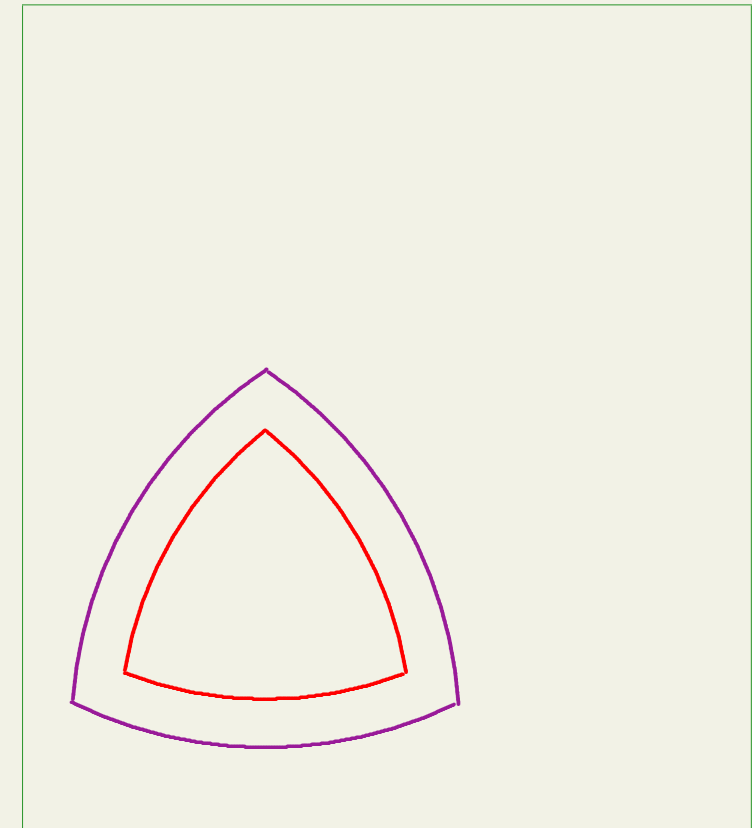
Claim 4: If f is monotone then $\text{lfp } f \subseteq f(\text{lfp } f)$.

By Claim 3, $f(\text{lfp } f) \subseteq \text{lfp } f$.

By monotonicity, $f(f(\text{lfp } f)) \subseteq f(\text{lfp } f)$.

By Claim 1 (letting $A := f(\text{lfp } f)$),
 $\text{lfp } f \subseteq f(\text{lfp } f)$.

Or as proof tree.



Tarski's Fixpoint Theorem: QED

Claim 3 ($\text{lfp } f \subseteq f(\text{lfp } f)$) and Claim 4 ($f(\text{lfp } f) \subseteq \text{lfp } f$) together give the result:

If f is monotone, then $\text{lfp } f = f(\text{lfp } f)$.

So under appropriate conditions, lfp is a fixpoint combinator.
We will later reuse Claim 1.

Alternative: A Natural-Deduction Style Proof

The proof can also be presented in natural deduction style.

Tarski's Fixpoint Theorem (1)

Claim 1 (“*lfp* lower bound”): If $f A \subseteq A$ then $\text{lfp } f \subseteq A$.

$$\begin{array}{c}
 \frac{[f A \subseteq A]^1}{A \in \{u. fu \subseteq u\}} \text{CollectI} \\
 \frac{A \in \{u. fu \subseteq u\}}{\bigcap \{u. fu \subseteq u\} \subseteq A} \text{Inter_lower} \\
 \frac{\bigcap \{u. fu \subseteq u\} \subseteq A}{\text{lfp } f \subseteq A} \text{Def. } \text{lfp} \\
 \frac{\text{lfp } f \subseteq A}{f A \subseteq A \rightarrow \text{lfp } f \subseteq A} \rightarrow\text{-I}^1
 \end{array}$$

Tarski's Fixpoint Theorem (2)

Claim 2 (“*lfp* greatest”): For all A , if for all U , $f U \subseteq U$ implies $A \subseteq U$, then $A \subseteq \text{lfp } f$.

$$\begin{array}{c}
 \frac{[\forall x. f x \subseteq x \rightarrow A \subseteq x]^1}{\forall x. x \in \{u. f u \subseteq u\} \rightarrow A \subseteq x} \text{subst, CollectI} \\
 \frac{\forall x. x \in \{u. f u \subseteq u\} \rightarrow A \subseteq x}{A \subseteq \cap \{u. f u \subseteq u\}} \text{Inter_greatest} \\
 \frac{A \subseteq \cap \{u. f u \subseteq u\}}{A \subseteq \text{lfp } f} \text{Def. lfp} \\
 \frac{A \subseteq \text{lfp } f}{(\forall x. f x \subseteq x \rightarrow A \subseteq x) \rightarrow A \subseteq \text{lfp } f} \rightarrow\text{-I}^1
 \end{array}$$

Tarski's Fixpoint Theorem (3)

Claim 3: If f is **monotone** then $f(\text{lfp } f) \subseteq \text{lfp } f$.

$$\begin{array}{c}
 \frac{[fx \subseteq x]^2}{[mono\ f]^1 \quad \text{lfp } f \subseteq x} \\
 \hline
 \frac{f(\text{lfp } f) \subseteq f\ x \quad [fx \subseteq x]^2}{f(\text{lfp } f) \subseteq x} \text{order_trans} \\
 \hline
 \frac{f(\text{lfp } f) \subseteq x}{\forall x. fx \subseteq x \rightarrow f(\text{lfp } f) \subseteq x} \forall\text{-I}, \rightarrow\text{-I}^2 \\
 \hline
 \frac{\forall x. fx \subseteq x \rightarrow f(\text{lfp } f) \subseteq x}{f(\text{lfp } f) \subseteq \text{lfp } f} \text{lfp_greatest}, \rightarrow\text{-E} \\
 \hline
 \frac{f(\text{lfp } f) \subseteq \text{lfp } f}{mono\ f \rightarrow f(\text{lfp } f) \subseteq \text{lfp } f} \rightarrow\text{-I}^1
 \end{array}$$

Tarski's Fixpoint Theorem (4)

Claim 4: If f is **monotone** then $lfp\ f \subseteq f(lfp\ f)$.

$$\begin{array}{c}
 \frac{[mono\ f]^1 \quad \frac{f(lfp\ f) \subseteq lfp\ f}{\text{Claim 3, } \rightarrow -E}}{\text{monoD}} \\
 \frac{f(f(lfp\ f)) \subseteq f(lfp\ f)}{lfp\ f \subseteq f(lfp\ f)} \text{ lfp_lowerbound, } \rightarrow -E \\
 \frac{lfp\ f \subseteq f(lfp\ f)}{mono\ f \rightarrow lfp\ f \subseteq f(lfp\ f)} \rightarrow -I^1
 \end{array}$$

Completing Proof Tree

$$\begin{array}{c}
 \frac{[mono\ f]^1}{lfp\ f \subseteq f(lfp\ f)} \text{ Claim 4} \quad \frac{[mono\ f]^1}{f(lfp\ f) \subseteq lfp\ f} \text{ Claim 3} \\
 \hline
 lfp\ f = f(lfp\ f) \\
 \hline
 mono\ f \rightarrow lfp\ f = f(lfp\ f) \rightarrow -/1
 \end{array}$$

equality

Induction Based on Lfp.thy

Theorem 2 (lfp induction):

If

- f is **monotone**, and
 - $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \{x \mid P x\}$,
- then $\text{lfp } f \subseteq \{x \mid P x\}$.

Induction Based on Lfp.thy

Theorem 2 (lfp induction):

If

- f is **monotone**, and
 - $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \{x \mid P x\}$,
- then $\text{lfp } f \subseteq \{x \mid P x\}$.

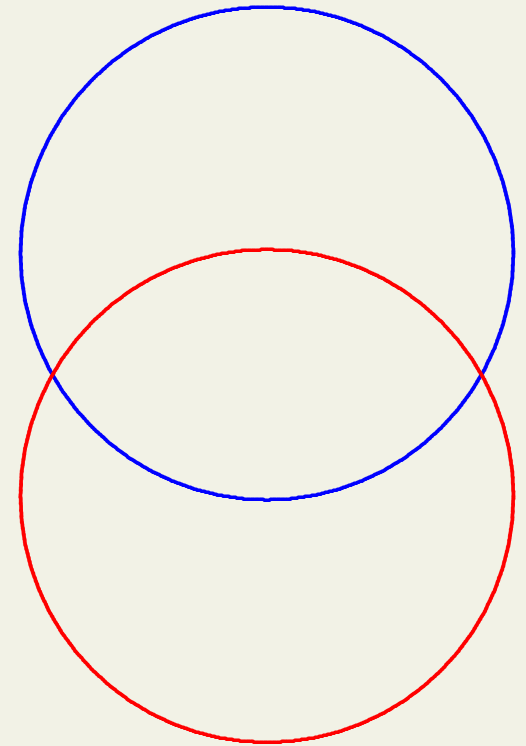
In **Isabelle**, it is called `lfp_induct`:

$$\llbracket a \in \text{lfp } f; \text{mono } f; \bigwedge x. x \in f(\text{lfp } f \cap \{x. P x\}) \implies P x \rrbracket \\ \implies P a$$

We now show the theorem similarly as **Tarski's Theorem**.

Showing lfp_induct

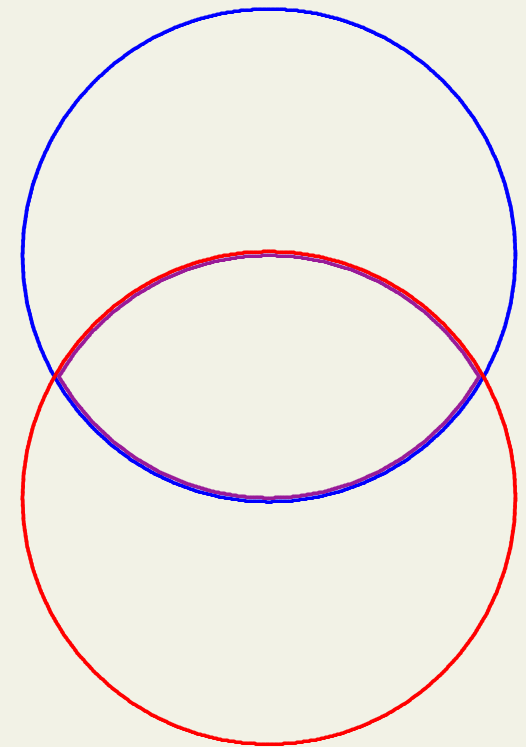
Circles denote $\text{lfp } f$ and $\{x \mid P x\}$.



Showing lfp_induct

Circles denote $\text{lfp } f$ and $\{x \mid P x\}$.

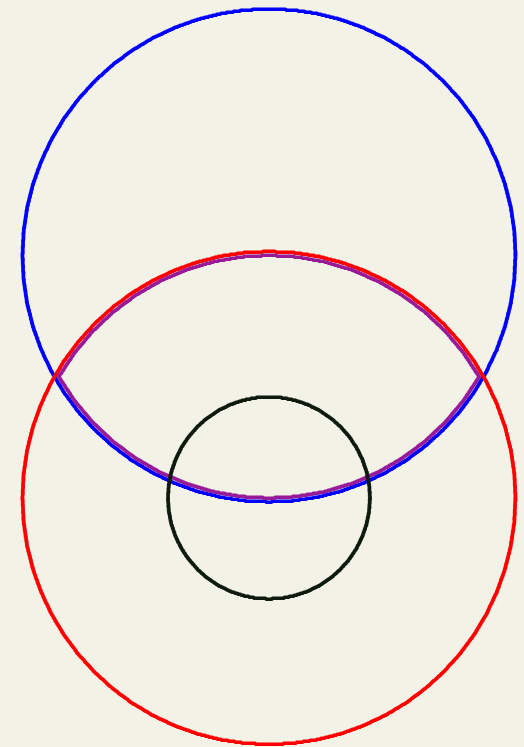
By monotonicity, $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq f(\text{lfp } f)$. By Tarski, $\text{lfp } f = f(\text{lfp } f)$.



Showing lfp_induct

Circles denote $\text{lfp } f$ and $\{x \mid P x\}$.

By monotonicity, $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq f(\text{lfp } f)$. By Tarski, $\text{lfp } f = f(\text{lfp } f)$. Hence $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \text{lfp } f$.

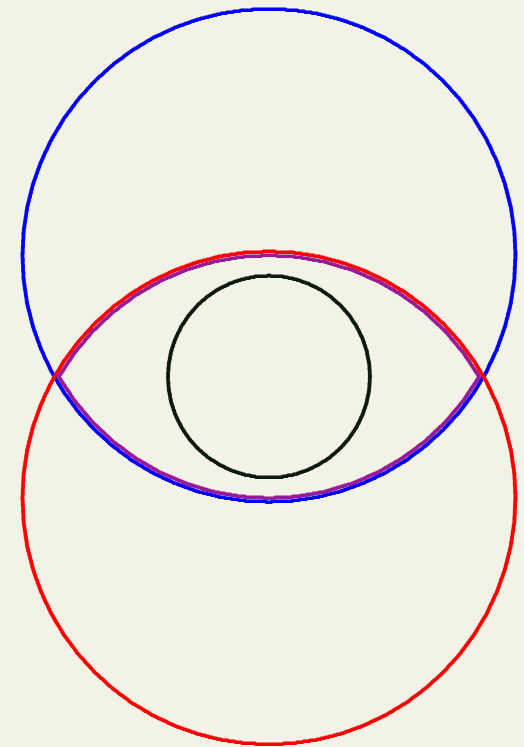


Showing lfp_induct

Circles denote $\text{lfp } f$ and $\{x \mid P x\}$.

By monotonicity, $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq f(\text{lfp } f)$. By Tarski, $\text{lfp } f = f(\text{lfp } f)$. Hence $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \text{lfp } f$.

By hypothesis, $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \{x \mid P x\}$, and so we must adjust picture: $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \text{lfp } f \cap \{x \mid P x\}$.



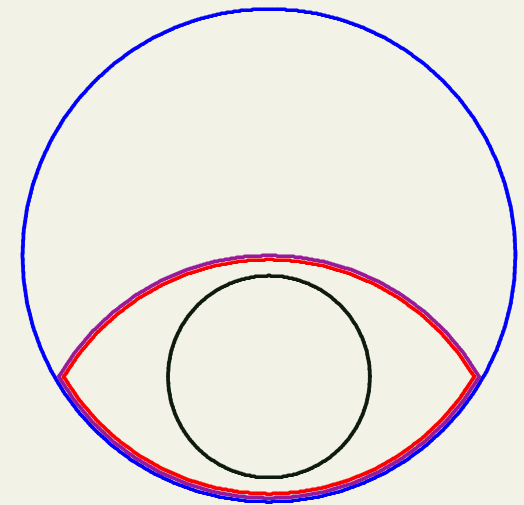
Showing lfp_induct

Circles denote $\text{lfp } f$ and $\{x \mid P x\}$.

By monotonicity, $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq f(\text{lfp } f)$. By Tarski, $\text{lfp } f = f(\text{lfp } f)$. Hence $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \text{lfp } f$.

By hypothesis, $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \{x \mid P x\}$, and so we must adjust picture: $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \text{lfp } f \cap \{x \mid P x\}$.

By Claim 1, $\text{lfp } f \subseteq \text{lfp } f \cap \{x \mid P x\}$ and so $\text{lfp } f = \text{lfp } f \cap \{x \mid P x\}$.



Showing lfp_induct

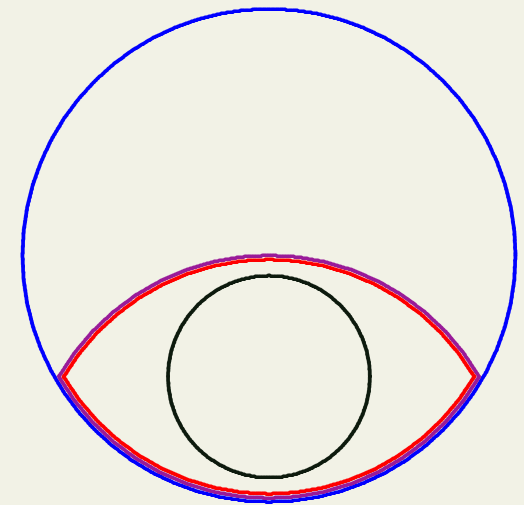
Circles denote $\text{lfp } f$ and $\{x \mid P x\}$.

By monotonicity, $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq f(\text{lfp } f)$. By Tarski, $\text{lfp } f = f(\text{lfp } f)$. Hence $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \text{lfp } f$.

By hypothesis, $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \{x \mid P x\}$, and so we must adjust picture: $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \text{lfp } f \cap \{x \mid P x\}$.

By Claim 1, $\text{lfp } f \subseteq \text{lfp } f \cap \{x \mid P x\}$ and so $\text{lfp } f = \text{lfp } f \cap \{x \mid P x\}$.

Conclusion: $\text{lfp } f \subseteq \{x \mid P x\}$.



Approximating Fixpoints

Looking ahead: Suppose we have the set \mathbb{N} of natural numbers (the **type** is formally introduced **later**). The theorem approx

$$(\forall S. f(\bigcup S) = \bigcup (f \cdot S)) \implies \bigcup_{n \in \mathbb{N}} (f^n \{\}) = \text{lfp } f$$

shows a way of approximating lfp , which is important for **algorithmic solutions** (e.g. in program analysis).

You will show this as an exercise.

Where Are We Going? Induction and Recursion

Let's step back: What is an **inductive definition** of a **set** S ?

Where Are We Going? Induction and Recursion

Let's step back: What is an **inductive definition** of a **set** S ?

It has the form: S is the smallest set such that:

- $\emptyset \subseteq S$ (just mentioned for emphasis);
- if $S' \subseteq S$ then $F(S') \subseteq S$ (for some appropriate F).

Where Are We Going? Induction and Recursion

Let's step back: What is an **inductive definition** of a **set** S ?

It has the form: S is the smallest set such that:

- $\emptyset \subseteq S$ (just mentioned for emphasis);
- if $S' \subseteq S$ then $F(S') \subseteq S$ (for some appropriate F).

At the same time, S is the smallest solution of the **recursive equation** $S = F(S)$.

Where Are We Going? Induction and Recursion

Let's step back: What is an **inductive definition** of a **set** S ?

It has the form: S is the smallest set such that:

- $\emptyset \subseteq S$ (just mentioned for emphasis);
- if $S' \subseteq S$ then $F(S') \subseteq S$ (for some appropriate F).

At the same time, S is the smallest solution of the **recursive equation** $S = F(S)$.

Induction and recursion are two faces of the same coin.

Lfp.thy for Inductive Definitions

Least fixpoints are for building inductive definitions of **sets** in a **definitional way**: $S := \text{lfp } F$.

This is **obviously** well-defined, so why this fuss about **monotonicity** and **Tarski**?

Lfp.thy for Inductive Definitions

Least fixpoints are for building inductive definitions of **sets** in a **definitional way**: $S := \text{lfp } F$.

This is **obviously** well-defined, so why this fuss about **monotonicity** and **Tarski**?

Tarski allows us to exploit the equation $\text{lfp } f = f(\text{lfp } f)$ in **proofs** about S ! That's what lfp is all about.

Example (Revisited)

The set of all finite subsets of a set A :

$$\textit{Fin } A = \textit{lfp } F$$

where $F = \lambda X. \{\{\}\} \cup \bigcup x \in A. ((\textit{insert } x) ' X)$.

Example (Revisited)

The set of all finite subsets of a set A :

$$\text{Fin } A = \text{lfp } F$$

where $F = \lambda X. \{\{\}\} \cup \bigcup x \in A. ((\text{insert } x) ' X)$.

Thus we can do using lfp what we would have wanted to do using Y .

To show: F is monotone!

In the Isabelle library, this is done a bit differently.

There will be an exercise on this.

The Package for Inductive Sets

Since monotonicity proofs can be automated, Isabelle has special proof support for inductive definitions. Example:

```
consts Fin :: 'a set => 'a set set
inductive "Fin(A)"
  intrs
    emptyI   "{} : Fin(A)"
    insertI  "[| a: A;  b: Fin(A) |] ==>
               insert a b : Fin(A)"
```

Translated into expression using *lfp*.

Package relies on proven lemma *lfp_unfold*.

Technical Support for Inductive Definitions

Support important in practice since many constructions are based on inductively defined sets (**datatypes**, . . .). Support provided for:

- Automatic proof of monotonicity
- Automatic proof of **induction rule**, for example:

$$\llbracket xa \in Fin\ A; P\ \{\}; \bigwedge a\ b. \llbracket a \in A; b \in Fin\ A; P\ b \rrbracket \implies \\ P\ (insert\ a\ b) \rrbracket \implies P\ xa$$

This works also for **mutually recursive** definitions, **co-inductive** definitions, . . .

Example: Program Semantics

This is an Isabelle/HOL add-on: Can reason about programming languages in Isabelle/HOL.

In **imperative programming**, program statements such as assignments or if-statements transform a program state (which consists of the current values of the program variables) into a new state.

This transformation is defined inductively.

We will **later** study a small imperative programming language.

Summary on Least Fixpoints

We are interested in recursion because **inductively defined sets** and **recursively defined functions** are solutions to recursive equations.

We cannot have general fixpoint operator Y , but we have, by **conservative extension**, least fixpoints for **defining sets**.

There is an induction scheme (**lfp induction**) for proving theorems about an inductively defined set.

Restriction of F to **set type** ($\alpha \text{ set} \rightarrow \alpha \text{ set}$) means that least fixpoints are not generally suitable for defining **functions** . . . ►

More Detailed Explanations

Defining *fac*

In the following explanations, any constants like 1 or + or **if-then-else** are intended to have their usual meaning.

A **fixpoint combinator** is a function Y that returns a fixpoint of a function F , i.e., Y must fulfill the equation $YF = F(YF)$. Doing λ -abstraction over F on both sides and η -conversion (backwards) on the left-hand side, we have

$$Y = \lambda F.F(YF)$$

This is a recursive equation. We will now demonstrate how a definition of a function *fac* (factorial) using a recursive equation can be transformed to a definition that uses Y instead of using recursion directly.

In a functional programming language we might define

$$fac\ n = (\text{if } n = 0 \text{ then } 1 \text{ else } n * fac\ (n - 1)).$$

We now massage this equation a bit. Doing λ -abstraction on both sides we get

$$\lambda n. \text{fac } n = (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fac}(n - 1))$$

which is the η -conversion of

$$\text{fac} = (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fac}(n - 1))$$

which in turn is a β -reduction of

$$\text{fac} = \underline{((\lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n - 1)) \text{ fac})} \quad (3)$$

We are looking for a solution to (3). We abbreviate the underlined expression by Fac . We claim $\text{fac} = Y \text{ Fac}$, i.e., it is a solution to (3).

Simply replacing fac with $Y\ Fac$ in (3) we get

$$Y\ Fac = Fac\ (Y\ Fac)$$

which holds by the definition of Y .

Thus we see that a recursive definition of a function can be transformed so that the function is the fixpoint of an appropriate functional (a function taking a function as argument).

Defining *Fin* *A*

We want to define a function *Fin* such that *Fin* *A* is the set of all finite subsets of *A*.

How do you construct the set of all finite subsets of *A*? The following pseudo-code suggests what you have to do:

```
S := {{}};  
forever do  
  foreach a ∈ A do  
    foreach B ∈ S do  
      add (a ∪ B) to S  
    od od od
```

This means that you have to add new sets forever (however, when you actually do this construction for a **finite** set *A*, it will indeed reach a

fixpoint, i.e., adding new sets won't change anything).

Generally (even if A is infinite), $Fin\ A$ is a set such that adding new sets as suggested by the pseudo-code won't change anything. Written as recursive equation:

$$Fin\ A = \{\{\}\} \cup \bigcup_{x \in A}.((\text{insert } x) ' Fin\ A)$$

Recall that $'$ is nice syntax for *image*, defined in `Set.thy`.

The above is a β -reduction of

$$Fin\ A = \underbrace{(\lambda X. \{\{\}\} \cup \bigcup_{x \in A}.((\text{insert } x) ' X))}_{FA} (Fin\ A) \quad (4)$$

We are looking for a solution to (4). We abbreviate the underlined expression by FA . We claim

$$Fin\ A = Y\ FA,$$

i.e., it is a solution to (4). Simply replacing $Fin\ A$ with $Y\ FA$ in (4) we get

$$Y\ FA = FA(Y\ FA),$$

which holds by the definition of Y .

You should compare this to what we said about *fac*. Note that in this example, there is no such thing as a recursive call to a “smaller” argument as in *fac* example.

Lfp.thy

This file should be contained in your Isabelle distribution. In Freiburg it is

`/usr/local/isabelle/Isabelle/src/HOL/Lfp.thy`

Lfp.ML

This file should be contained in your Isabelle distribution. In Freiburg it is

`/usr/local/isabelle/Isabelle/src/HOL/Lfp.ML`

Algorithmic Version of lfp

The theorem

$$(\forall S. f(\bigcup S) = \bigcup (f \cdot S)) \implies \bigcup_{n \in \mathbb{N}} (f^n \{\}) = lfp f$$

says that under a certain condition, $lfp f$ can be computed by applying f to the empty set over and over again:

- although the expression uses the union over all natural numbers, which is an infinite set, this can sometimes effectively be computed. Under certain conditions, there exists a k such that $f^k \{\} = f^{k+1} \{\}$.
- Even if $\bigcup_{n \in \mathbb{N}} f^n \{\}$ cannot be effectively computed, it can still be **approximated**: for any k , we know that $\bigcup_{i \leq k} f^i \{\} \subseteq \bigcup_{n \in \mathbb{N}} f^n \{\}$.

Monotone Functions

A function f is monotone w.r.t. a **partial order** \leq if the following holds:
 $A \leq B$ implies $f(A) \leq f(B)$.

In particular, we consider the order given by the subset relation.

“The Set” α

α is not a set but a type (variable). But we can consider the set of all terms of that type (UNIV of type α).

The polymorphic constant UNIV was defined in [Set.thy](#). UNIV of type τ set is the set containing all terms of type τ .

Three Circles?

In general, needless to say, there could be any number of such sets, but the picture is to be understood in the sense that the three circles are all the sets A with the property $f A \subseteq A$.

Different Phrasings

The theorem is phrased a bit differently in the “mathematical” version we give here and in the Isabelle version (see [Lfp.ML](#)). This is convenient for the graphical illustration of the proof.

The “mathematical phrasing” corresponding closely to the Isabelle version would be the following:

Theorem 3 (Induct (alternative)):

If

- $a \in \text{lfp } f$, and
- f is **monotone**, and
- for all x , $x \in f(\text{lfp } f \cap \{x \mid P x\})$ implies $P x$

then $P a$ holds.

Other phrasings, which may help to get some intuition about the

theorem:

Theorem 4 (Induct (alternative)):

If

- $a \in \text{lfp } f$, and
- f is **monotone**, and
- $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \{x \mid P x\}$

then $P a$ holds.

Theorem 5 (Induct (alternative)):

If

- f is **monotone**, and
- $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \{x \mid P x\}$

then for all x in $\text{lfp } f$, we have $P x$.

Detail on Monotonicity

$lfp\ f \cap \{x \mid P\ x\} \subseteq lfp\ f$, so by monotonicity,
 $f(lfp\ f \cap \{x \mid P\ x\}) \subseteq f(lfp\ f)$.

Use of Claim 1

We have just seen $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \text{lfp } f \cap \{x \mid P x\}$.

By Claim 1 (setting $A := \text{lfp } f \cap \{x \mid P x\}$), this implies

$\text{lfp}(f) \subseteq \text{lfp } f \cap \{x \mid P x\}$.

Antisymmetry of \subseteq

We have $\text{lfp } f \cap \{x \mid P x\} \subseteq \text{lfp}(f)$ and $\text{lfp}(f) \subseteq \text{lfp } f \cap \{x \mid P x\}$, and so $\text{lfp}(f) = \text{lfp } f \cap \{x \mid P x\}$ by the antisymmetry of \subseteq .

Recursion in a Definitional Way

Recall why we were interested in fixpoints.

The problem with Y is that it leads to inconsistency (and of course, the definition of Y is not a constant definition/conservative extension.).

The definition of lfp is conservative.

And in appropriate situations, it can be used to define recursive functions.

Compared to Y , the type of lfp is restricted.

This restriction means that there is no obvious way to use lfp for defining recursive numeric functions such as fac .

Finite Sets in Isabelle

Above, we defined the set of finite subsets of a set A . Alternatively, one could define “the set of all finite sets whose elements have type τ ”. In this case, no fixed set A is involved, and it is closer to what actually happens in Isabelle. In `Finite_Set.thy` a constant *Finities* is defined. It has polymorphic type $\alpha \text{ set set}$. We have $A \in \text{Finities}$ if and only if A is a finite set. However, it would be wrong to think of *Finities* as one single set that contains all finite sets. Instead, for each τ , there is a **polymorphic instance** of *Finities* of type $\tau \text{ set set}$ containing all finite sets of element type τ .

In `Finite_Set.thy` we find the lines

```
inductive "Finites"
  intros
    emptyI [simp, intro!]: "{} : Finites"
    insertI [simp, intro!]: "A : Finites ==>
      insert a A : Finites"
```

The Isabelle mechanism of interpreting the keyword `inductive` translates this into the following definition: $Finites = lfp\ G$ where

$$G \equiv \lambda S. \{x \mid x = \{\} \vee (\exists A\ a. x = insert\ a\ A \wedge A \in S)\}$$

You can see this by typing in your proof script:

```
open Finites;
defs;
```

Talking (ML-)technically, `Finities` is a **structure** (module), and `defs` is a value (component) of this **structure**.

As a sanity-check, consider **the type** of this expression. The expression `insert a A` forces A to be of type $\tau \text{ set}$ for some τ and a to be of type τ . Next, `insert a A` is of type $\tau \text{ set}$, and hence x is also of type $\tau \text{ set}$. Moreover, the expression `$A \in S$` forces S to be of type $\tau \text{ set set}$. The expression $\{x \mid x = \{\} \vee (\exists A a. x = \text{insert } a A \wedge A \in S)\}$ is of type $\tau \text{ set set}$. Next, G is of type $\tau \text{ set set} \rightarrow \tau \text{ set set}$, and so finally, `Finities` is of type $\tau \text{ set set}$. But actually, since τ is arbitrary, we can replace it by a type variable α .

Note that there is a convenient syntactic translation

translations `"finite A"` == `"A : Finities"`

When does Isabelle generate ML-structures, and what are the names of

those structures?

This question is highly Isabelle-technical, related to different formats used for writing theory files, which is in turn partly due to mere historic reasons.

It used to be the case that for a theory file called $F.thy$, a structure F would be generated. Certain keywords in $F.thy$ such as `inductive`, `recursive`, and `datatype`, would trigger the creation of substructures, so for example `inductive I` would call for the creation of a substructure I .

For a newer format of theory files, this is no longer the case.

The treatment of the keyword `constdefs`, followed by the declaration and definition of a constant C , also depends on the format used for writing theory files.

- Sometimes (when an older format is used), it will automatically

generate a `thm` called C_def which is the definition of C .

- Sometimes (when a newer format is used), it will insert the definition of C into a database which can be accessed by a function called `thm` taking a string as argument. In this case, not C_def would be the definition of C , but rather

`thm "C_def"`

You should be aware of such problems, but we do not treat them in this course.

Finite_Set.thy

This file should be contained in your Isabelle distribution. In Freiburg it is

`/usr/local/isabelle/Isabelle/src/HOL/Finite_Set.thy`

F is monotone

This proof is of course done in Isabelle.

Induction for Finite Sets

The theorem

$$\begin{aligned} & \llbracket xa \in \text{Fin } A; P \ \{\}; \bigwedge ab. \llbracket a \in A; b \in \text{Fin } A; P \ b \rrbracket \implies P \ (\text{insert } a \ b) \rrbracket \\ & \implies P \ x a \end{aligned}$$

is an instance of the general **induction scheme**. That is to say, if we take the general induction scheme **lfp_induct**

$$\llbracket a \in \text{lfp } f; \text{mono } f; \bigwedge x. x \in f(\text{lfp } f \cap \{x.P \ x\}) \implies P \ x \rrbracket \implies P \ a$$

and instantiate f to $\lambda X. \{\{\}\} \cup \bigcup x \in A. ((\text{insert } x) ' X)$ then some massaging using the definitions will give us the first theorem.

Note here that monotonicity has disappeared from the assumptions. This is because the **monotonicity of F** is shown by Isabelle **once and for all**.

This is one aspect of what we mean by **special proof support for inductive definitions**.

The least fixpoint of the functional is $\text{Fin } A$ (the set of finite subsets of A) in this case.

Mutually Recursive Definitions

Two functions f and g are **mutually recursive** if f is defined in terms of g and vice versa.

Co-induction

Co-induction is a construction analogous to induction but using **greatest** fixpoints.

Well-Founded Recursion

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL](#) library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Well-Founded Recursion

After least fixpoints, **well-founded recursion** is our second concept of recursion (and fixpoint combinator).

Idea: Modeling “terminating” recursive functions, i.e. recursive definitions that use “smaller” arguments for the recursive call.

Prerequisite: Relations

We need some standard operations on **binary relations** (sets of **pairs**), such as converse, composition, image of a set and a relation, the identity relation, . . .

These are provided by **Relation.thy**.

Relation.thy (Fragment)

constdefs

```
converse :: "('a * 'b) set => ('b * 'a) set"
"r-1 == {(y, x). (x, y):r}"
rel_comp  :: "[('b * 'c) set, ('a * 'b) set] =>
               ('a * 'c) set"
"r ∘ s == {(x,z). EX y. (x, y):s & (y, z):r}"
Image :: "[('a * 'b) set, 'a set] => 'b set"
"r `` s == {y. EX x:s. (x,y):r}"
Id      :: "('a * 'a) set"
"Id == {p. EX x. p = (x,x)}"
```

Somewhat similar to [Fun.thy](#).

Prerequisite: Closures

We need the transitive, as well as the reflexive transitive closure of a relation.

These are provided by `Transitive_Closure.thy`.

How would you define those inductively, ad-hoc?

Transitive_Closure.thy (Fragment)

consts

```
rtranc1 :: "('a * 'a) set => ('a * 'a) set"  
          ("(_^*)" [1000] 999)
```

inductive "r^*"

intros

```
rtranc1_refl [...]: "(a, a) : r^*"
```

```
rtranc1_into_rtranc1 [...]: "(a, b) : r^* ==>  
    (b, c) : r ==> (a, c) : r^*"
```


Transitive_Closure.thy (Fragment Cont.)

consts

```
tranc1 :: "('a * 'a) set => ('a * 'a) set"  
        ("(_^+)" [1000] 999)
```

inductive "r^+"

intros

```
r_into_tranc1 [...]: "(a, b) : r ==>  
                        (a, b) : r^+"
```

```
tranc1_into_tranc1 [...]: "(a, b) : r^+ ==>  
                            (b, c) : r ==> (a, c) : r^+"
```

Well-Founded Orderings

Defined in `Wellfounded_Recursion.thy`.

`Wellfounded_Recursion = Transitive_Closure +
constdefs`

```
wf                :: "('a * 'a) set => bool"  
"wf(r) == (!P. (!x. (!y. (y,x):r --> P(y))  
                --> P(x)) --> (!x. P(x)))"
```

What does this mean?

Well-Founded Orderings

Defined in `Wellfounded_Recursion.thy`.

`Wellfounded_Recursion = Transitive_Closure +
constdefs`

```
wf                :: "('a * 'a) set => bool"  
"wf(r) == (!P. (!x. (!y. (y,x):r --> P(y))  
                --> P(x)) --> (!x. P(x)))"
```

What does this mean? A relation r is **well-founded** if well-founded (Noetherian) induction based on r is a valid proof scheme.

Example: Is \emptyset well-founded?

Intuition of Well-Foundedness

Intuition of wf : All descending chains are finite.

But: Cannot express infinity; must look for alternatives.

- Not symmetric:

Intuition of Well-Foundedness

Intuition of wf : All descending chains are finite.

But: Cannot express infinity; must look for alternatives.

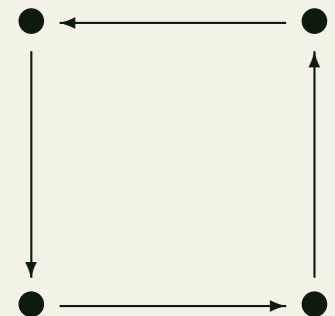
- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?

Intuition of Well-Foundedness

Intuition of wf : All descending chains are finite.

But: Cannot express infinity; must look for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?



Intuition of Well-Foundedness

Intuition of wf : All descending chains are finite.

But: Cannot express infinity; must look for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles:

Intuition of Well-Foundedness

Intuition of wf : All descending chains are finite.

But: Cannot express infinity; must look for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?

Intuition of Well-Foundedness

Intuition of wf : All descending chains are finite.

But: Cannot express infinity; must look for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?



Intuition of Well-Foundedness

Intuition of wf : All descending chains are finite.

But: Cannot express infinity; must look for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?
- r has minimal element:

Intuition of Well-Foundedness

Intuition of wf : All descending chains are finite.

But: Cannot express infinity; must look for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?
- r has minimal element: $\exists x. \forall y. (y, x) \notin r$?

Note: Trivial for $r = \emptyset$.

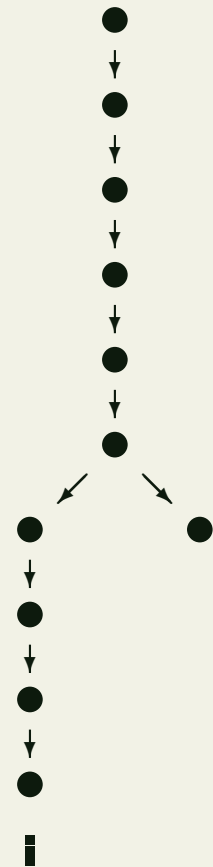
Intuition of Well-Foundedness

Intuition of wf : All descending chains are finite.

But: Cannot express infinity; must look for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?
- r has minimal element: $\exists x. \forall y. (y, x) \notin r$?

Note: Trivial for $r = \emptyset$.



Intuition of Well-Foundedness

Intuition of wf : All descending chains are finite.

But: Cannot express infinity; must look for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?
- r has minimal element: $\exists x. \forall y. (y, x) \notin r$?

Note: Trivial for $r = \emptyset$.

- Any subrelation must have minimal element:

Intuition of Well-Foundedness

Intuition of wf : All descending chains are finite.

But: Cannot express infinity; must look for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?
- r has minimal element: $\exists x. \forall y. (y, x) \notin r$?

Note: Trivial for $r = \emptyset$.

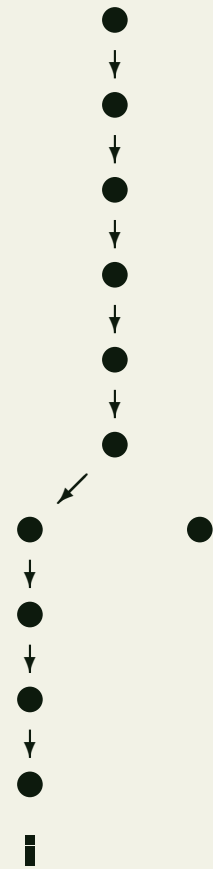
- Any subrelation must have minimal element:
 $\forall p. p \subseteq r \rightarrow \exists x. \forall y. (y, x) \notin p$?

Intuition of Well-Foundedness

Intuition of wf : All descending chains are finite.

But: Cannot express infinity; must look for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?
- r has minimal element: $\exists x. \forall y. (y, x) \notin r$?
Note: Trivial for $r = \emptyset$.
- Any subrelation must have minimal element:
 $\forall p. p \subseteq r \rightarrow \exists x. \forall y. (y, x) \notin p$? “Minimal element” badly formalized (already in previous point).



The Characterisation

All these attempts are just **necessary** but not **sufficient** conditions for well-foundedness.

Here is a characterization:

$$(\forall Q. x \in Q \rightarrow (\exists z \in Q. \forall y. (y, z) \in r \rightarrow y \notin Q))$$

Here is an alternative characterization (exercise):

$$(\forall r. r \neq \{\} \wedge r \subseteq p \rightarrow (\exists x \in \text{Domain } r. \forall y. (y, x) \notin r))$$

Let's see some theorems to confirm our intuition, including the statements just shown.

A Theorem on the Empty Set

$\text{wf_empty} \quad \text{wf } \{\}$

Proof sketch:

wf_empty : substitute r into definition, simplify.

A Theorem for Induction

By *massage* of the definition of well-foundedness

$$\forall P. (\forall x. (\forall y. (y, x) \in r \rightarrow P y) \rightarrow P x) \rightarrow (\forall x. P x)$$

one obtains the theorem `wf_induct`

$$\llbracket wf\ r; \bigwedge x. \forall y. (y, x) \in r \rightarrow P y \implies P x \rrbracket \implies P a.$$

This is a form suitable for doing induction proofs in Isabelle.

Induction Theorem as Proof Rule

The Isabelle theorem `wf_induct`

$$\llbracket wf\ r; \bigwedge x. \forall y. (y, x) \in r \rightarrow P\ y \Longrightarrow P\ x \rrbracket \Longrightarrow P\ a.$$

as proof rule:

$$\frac{wf\ r \quad \begin{array}{c} [\forall y. (y, x) \in r \rightarrow P\ y] \\ \vdots \\ P\ x \end{array}}{P\ a} wf_induct$$

A Theorem on Antisymmetry

$$\text{wf_not_sym} \quad \llbracket \text{wf } r; (a, x) \in r \rrbracket \implies (x, a) \notin r$$

Proof sketch:

$$\frac{\begin{array}{c} \text{wf } r \\ \forall y. (y, x) \in r \longrightarrow (\forall z. (y, z) \in r \longrightarrow (z, y) \notin r) \\ \vdots \\ \forall z. (x, z) \in r \longrightarrow (z, x) \notin r \end{array}}{\forall z. (a, z) \in r \longrightarrow (z, a) \notin r} \text{wf_induct}$$

Rest routine though not so trivial (needs **classical reasoning**).
A variation will be done as exercise.

Theorems on Absence of Cycles

`wf_not_refl` $wf\ r \implies (a, a) \notin r$

`wf_trancl` $wf\ r \implies wf\ (r^+)$

`wf_acyclic` $wf\ r \implies acyclic\ r$

$(acyclic\ r \equiv \forall x. (x, x) \notin r^+)$

Proof sketch:

`wf_not_refl`: Corollary of `wf_not_sym`.

`wf_trancl`: Uses induction.

`wf_acyclic`: Apply `wf_not_refl` and `wf_trancl`

Ergo: Definition of *wf* really meets our intuition of “no cycles”.

Another Theorem (“Exists Minimal Element”)

`wf_minimal` $wf\ r \implies \exists x. \forall y. (y, x) \notin r^+$

Proof sketch, writing $\phi \equiv (\exists x. \forall y. (y, x) \notin r^+)$:

$wf(r)$

_____ ϕ `wf_minimal`

This is what we must construct.

Another Theorem (“Exists Minimal Element”)

$$\text{wf_minimal} \quad \text{wf } r \implies \exists x. \forall y. (y, x) \notin r^+$$

Proof sketch, writing $\phi \equiv (\exists x. \forall y. (y, x) \notin r^+)$:

$$\forall w. (w, v) \in r^+ \rightarrow \phi$$

$$\frac{\text{wf}(r) \quad \text{wf}(r^+) \quad \phi}{\phi} \text{wf_induct}$$

Note “special case”: w and v do **not occur** in ϕ !

Another Theorem (“Exists Minimal Element”)

`wf_minimal` $wf\ r \implies \exists x. \forall y. (y, x) \notin r^+$

Proof sketch, writing $\phi \equiv (\exists x. \forall y. (y, x) \notin r^+)$:

$$\begin{array}{l} \forall w. (w, v) \\ \in r^+ \rightarrow \phi \end{array}$$

$$\frac{\frac{wf(r)}{wf(r^+)} \bullet \quad \phi}{\phi} wf_induct$$

This is `wf_tranc1`.

$$\text{wf_minimal} \quad wf \, r \implies \exists x. \forall y. (y, x) \notin r^+$$

Proof sketch, writing $\phi \equiv (\exists x. \forall y. (y, x) \notin r^+)$:

$$\begin{array}{c}
\frac{\text{wf}(r)}{\text{wf}(r^+)} \bullet \quad \frac{\phi \vee \neg\phi \quad \phi \quad \phi}{\phi} \text{disjE} \\
\hline
\phi \quad \text{wf_induct}
\end{array}$$

We now try a proof by **case distinction** on ϕ .

Another Theorem (“Exists Minimal Element”)

$\text{wf_minimal} \quad \text{wf } r \implies \exists x. \forall y. (y, x) \notin r^+$

Proof sketch, writing $\phi \equiv (\exists x. \forall y. (y, x) \notin r^+)$:

$$\begin{array}{c}
 \text{wf}(r) \quad \overline{\phi \vee \neg\phi} \quad \phi \quad \phi \quad \phi \\
 \hline
 \text{wf}(r^+) \quad \phi \quad \phi \quad \phi \quad \phi \quad \phi \\
 \hline
 \phi \quad \text{wf_induct}
 \end{array}
 \quad
 \begin{array}{c}
 \neg\phi \quad \neg\phi \quad \forall w. (w, v) \in r^+ \rightarrow \phi \\
 \hline
 \phi
 \end{array}
 \quad
 \text{disjE}$$

Classical reasoning.

Another Theorem (“Exists Minimal Element”)

`wf_minimal` $wf\ r \implies \exists x. \forall y. (y, x) \notin r^+$

Proof sketch, writing $\phi \equiv (\exists x. \forall y. (y, x) \notin r^+)$:

$$\begin{array}{c}
 \frac{\neg\phi \quad \forall w. (w, v) \in r^+ \rightarrow \phi}{\forall x. \exists y. (y, x) \in r^+} \dots \\
 \frac{\frac{wf(r)}{wf(r^+)} \quad \frac{\phi \vee \neg\phi \quad \phi}{\phi} \text{disjE}}{\phi} wf_induct
 \end{array}$$

Using some elementary equivalences.

Another Theorem (“Exists Minimal Element”)

$\text{wf_minimal} \quad \text{wf } r \implies \exists x. \forall y. (y, x) \notin r^+$

Proof sketch, writing $\phi \equiv (\exists x. \forall y. (y, x) \notin r^+)$:

$$\begin{array}{c}
 \frac{\neg\phi}{\forall x. \exists y. (y, x) \in r^+} \dots \frac{\neg\phi \quad \frac{\forall w. (w, v) \in r^+ \rightarrow \phi}{\exists w. (w, v) \notin r^+}}{\bullet} \\
 \\
 \frac{\frac{\text{wf}(r)}{\text{wf}(r^+)} \bullet \quad \frac{\phi \vee \neg\phi \quad \bullet}{\phi} \quad \phi}{\phi} \text{disjE} \\
 \frac{\phi}{\phi} \text{wf_induct}
 \end{array}$$

This step works for **any** ϕ . Think semantically or check!

Another Theorem (“Exists Minimal Element”)

$\text{wf_minimal} \quad \text{wf } r \implies \exists x. \forall y. (y, x) \notin r^+$

Proof sketch, writing $\phi \equiv (\exists x. \forall y. (y, x) \notin r^+)$:

$$\begin{array}{c}
 \frac{\text{wf}(r)}{\text{wf}(r^+)} \bullet \quad \frac{\overline{\phi \vee \neg \phi} \bullet \quad \phi}{\phi} \quad \frac{\frac{\text{False}}{\phi} \text{ FalseE}}{\phi} \text{ disjE} \\
 \hline
 \phi \quad \text{wf_induct}
 \end{array}$$

$\frac{\neg \phi}{\forall x. \exists y. (y, x) \in r^+} \dots \quad \frac{\neg \phi \quad \forall w. (w, v) \in r^+ \rightarrow \phi}{\exists w. (w, v) \notin r^+} \bullet$

It is routine to derive *False*.

Another Theorem (“Exists Minimal Element”)

$\text{wf_minimal} \quad \text{wf } r \implies \exists x. \forall y. (y, x) \notin r^+$

Proof sketch, writing $\phi \equiv (\exists x. \forall y. (y, x) \notin r^+)$:

$$\begin{array}{c}
 \frac{\text{wf}(r)}{\text{wf}(r^+)} \bullet \quad \frac{\phi \vee \neg \phi}{\phi} \bullet \quad \frac{[\phi]^2}{\phi} \quad \frac{\frac{[\neg \phi]^2}{\forall x. \exists y. (y, x) \in r^+} \dots \quad \frac{[\neg \phi]^2 \quad \forall w. (w, v) \in r^+ \rightarrow \phi}{\exists w. (w, v) \notin r^+} \bullet}{\phi} \text{disjE}^2 \\
 \frac{\text{wf}(r^+) \quad \phi}{\phi} \text{wf_induct}
 \end{array}$$

This completes the proof by case distinction . . .

Another Theorem (“Exists Minimal Element”)

$\text{wf_minimal} \quad \text{wf } r \implies \exists x. \forall y. (y, x) \notin r^+$

Proof sketch, writing $\phi \equiv (\exists x. \forall y. (y, x) \notin r^+)$:

$$\begin{array}{c}
 \frac{\frac{\text{wf}(r)}{\text{wf}(r^+)} \bullet \quad \frac{\phi \vee \neg \phi \bullet \quad [\phi]^2}{\phi} \quad \frac{\frac{\frac{[\neg \phi]^2}{\forall x. \exists y. (y, x) \in r^+} \dots \quad \frac{[\neg \phi]^2 \quad [\forall w. (w, v) \in r^+ \rightarrow \phi]^1}{\exists w. (w, v) \notin r^+} \bullet}{\dots} \quad \frac{\text{False}}{\phi} \text{FalseE}}{\phi} \text{disjE}^2 \\
 \hline
 \phi \quad \text{wf_induct}^1
 \end{array}$$

... and the proof by induction.

Remarks on the Proof

We used an **instance** of **wf_induct**, where we instantiated x by v , y by w , and P by $\lambda w.(\exists x.\forall y.(y, x) \notin r^+)$. I.e., ϕ does **not contain** the “induction variables” w and v .

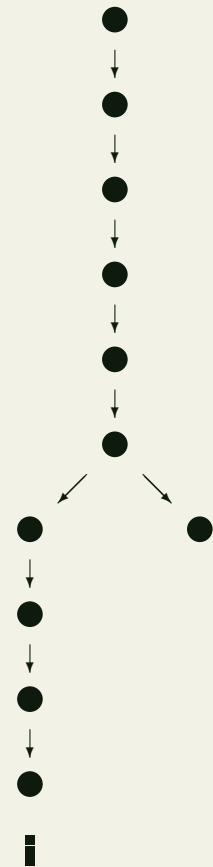
Still this is a “proper” induction proof: Although ϕ does not contain the “induction variables”, the proof **does depend** on the actual form of ϕ ! (Try doing it without induction . . .)

Scoping of quantifiers (e.g., in general $(\forall w.(w, v) \in r^+ \rightarrow \phi) \not\equiv (\forall w.(w, v) \in r^+) \rightarrow \phi$) and **side conditions** are very subtle in this proof. Underlines the importance of machine-checked proofs.

Remarks on wf_minimal

Ergo: Definition of **wf** fulfills the condition corresponding to our **first attempt** of characterizing well-foundedness using minimal elements.

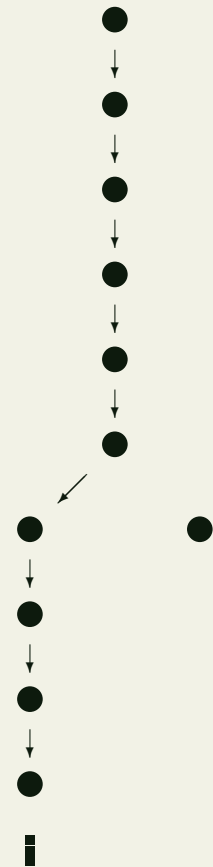
However, this formalization had a problem: there could be **local minima**, and



Remarks on `wf_minimal`

Ergo: Definition of `wf` fulfills the condition corresponding to our `first attempt` of characterizing well-foundedness using minimal elements.

However, this formalization had a problem: there could be **local minima**, and **isolated points** are also always minima. In particular, if r is empty, then any element is trivially a minimum.



A Characterization

The theorem `wf_eq_minimal` is characterization of well-foundedness.:

$$wf\ r = (\forall Q . x \in Q \rightarrow (\exists z \in Q . \forall y . (y, z) \in r \rightarrow y \notin Q))$$

Proof uses `split =`, `wf_def`, rest routine.

Ergo: Definition of `wf` meets textbook definitions “every non-empty set Q has a minimal element in r ” (more or less standard textbook).

A Theorem on Subsets

`wf_subset` $\llbracket wf\ r; p \subseteq r \rrbracket \implies wf\ p$

Proof sketch:

`wf_subset`: simplification tactic using `wf_eq_minimal`.

A Theorem on Subrelations

`wf_subrel`

$$wf\ r \implies \forall p. p \subseteq r \rightarrow \exists x. \forall y. (y, x) \notin p^+$$

Proof sketch:

Combine `wf_minimal` and `wf_subset`.

This implies $wf\ r \implies \forall p. p \subseteq r \rightarrow \exists x. \forall y. (x, y) \notin p$.

Ergo: Definition of `wf` fulfills the condition corresponding to our `second attempt` of characterizing well-foundedness using minimal elements.

A Theorem on Subrelations

`wf_subrel`

$$wf\ r \implies \forall p. p \subseteq r \rightarrow \exists x. \forall y. (y, x) \notin p^+$$

Proof sketch:

Combine `wf_minimal` and `wf_subset`.

This implies $wf\ r \implies \forall p. p \subseteq r \rightarrow \exists x. \forall y. (x, y) \notin p$.

Ergo: Definition of `wf` fulfills the condition corresponding to our `second attempt` of characterizing well-foundedness using minimal elements.

However, this formalization `still` had a problem: The minimum could be an `isolated` element, unrelated to the subrelation.

Defining Recursive Functions

Idea of **well-founded recursion**: Wish to define f by **recursive equation** $f = e$, e.g.

$$fac = (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * fac(n - 1))$$

Define $F = \lambda f. e$, e.g.

$$Fac = (\lambda fac. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * fac(n - 1))$$

Defining Recursive Functions

Idea of **well-founded recursion**: Wish to define f by recursive equation $f = e$, e.g.

$$fac = (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * fac(n - 1))$$

Define $F = \lambda f. e$, e.g. (α -conversion of what you have seen)

$$Fac = (\lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n - 1))$$

We say: F is the **functional defining** f .

Recall that $Y F$ would solve $f = e$, but we don't have Y , so what can we do?

Coherent Functionals

A functional F is **coherent w.r.t. $<$** if all recursive calls are with arguments “smaller” than the original argument. This means that if F has the form

$$\lambda f. \lambda n. e'$$

then for any $(f\ m)$ occurring in e' , we have $m < n$.

Here $<$ could be any relation (although the idea is that it should be a well-founded ordering).

(Simplification, assumes that recursion is on the first argument of f .)

Using Bad f 's

Let $f|_{<a}$ be a function that is like f on all values $< a$, and arbitrary elsewhere. $f|_{<a}$ is an approximation, a “bad” f .

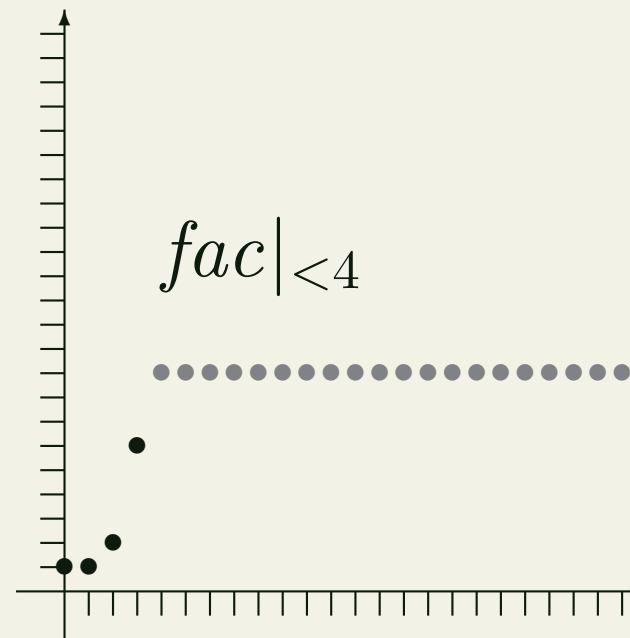
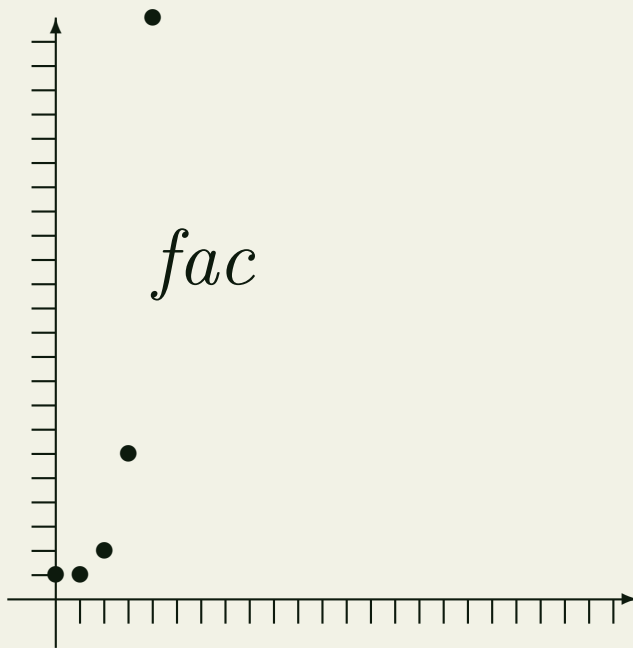
If F is **coherent**, then we would expect that for any a ,

$$f\ a = (F\ f)\ a = (F\ f|_{<a})\ a. \quad (5)$$

It's not that we are ultimately interested in constructing such a “bad” f , but our formalization of well-founded recursion **defines** coherence by the fact that one **could** use such a “bad” f , i.e., via (5).

“Bad” f ’s: Example

Consider fac . On the right-hand side, we show **one possibility** for $fac|_{<4}$:



cut (in `Wellfounded_Recursion.thy`)

constdefs

```
cut    :: "('a => 'b) => ('a * 'a) set =>
          'a => 'a => 'b"
```

```
"cut f r x ==
```

```
  (%y. if (y,x):r then f y else arbitrary)"
```

cut $f\ r\ x$ is what we denoted by $f|_{<x}$ (taking $<$ for r).

`arbitrary` is defined in `HOL.thy`.

cut (in `Wellfounded_Recursion.thy`)

constdefs

```
cut      :: "('a => 'b) => ('a * 'a) set =>
              'a => 'a => 'b"
```

```
"cut f r x ==
```

```
  (%y. if (y,x):r then f y else arbitrary)"
```

cut $f\ r\ x$ is what we denoted by $f|_{<x}$ (taking $<$ for r).

`arbitrary` is defined in `HOL.thy`.

The function cut $f\ r\ x$ is **unspecified** for arguments y where $(y, x) \notin r$, but for each such argument, $(\text{cut } f\ r\ x)\ y$ must be the same (in any particular **model**).

Theorems Involving cut

$$\begin{array}{ll} \text{cuts_eq} & (cut\ f\ r\ x = cut\ g\ r\ x) = \\ & (\forall y. (y, x) \in r \rightarrow f\ y = g\ y) \\ \text{cut_apply} & (x, a) \in r \implies cut\ f\ r\ a\ x = f\ x \end{array}$$

Or, using the more intuitive notation:

$$\begin{array}{ll} \text{cuts_eq} & (f|_{<x} = g|_{<x}) = (\forall y. y < x \rightarrow f\ y = g\ y) \\ \text{cut_apply} & x < a \implies f|_{<a}\ x = f\ x \end{array}$$

wfrec_rel (**in** Wellfounded_Recursion.thy)

construction: “approximate” f by a **relation** $wfrec_rel\ R\ F$.

```
wfrec_rel :: "('a * 'a) set =>
```

```
(( 'a => 'b) => 'a => 'b) => ( 'a * 'b) set"
```

```
inductive "wfrec_rel R F"
```

```
intrs
```

```
wfrecI "ALL z. (z, x) : R --> (z, g z) :
```

```
wfrec_rel R F ==> (x, F g x) : wfrec_rel R F"
```

wfrec_rel Explained

$$\forall z. (z, x) \in R \rightarrow (z, g\ z) \in wfrec_rel\ R\ F \implies \\ (x, F\ g\ x) \in wfrec_rel\ R\ F$$

- For R and F arbitrary, $wfrec_rel\ R\ F$ is defined but we wouldn't want to know what it is.
- But if R is well-founded and F is coherent, $wfrec_rel\ R\ F$ defines a recursive “function”.

Try to show that $(4, 24) \in (wfrec_rel\ ' < ' Fac)$.

Now let us really turn $wfrec_rel\ R\ F$ into a function . . .

wfrec (**in** Wellfounded_Recursion.thy)

```
wfrec :: "('a * 'a) set =>
```

```
((('a => 'b) => 'a => 'b) => 'a => 'b)"
```

```
"wfrec R F == %x. THE y.
```

```
(x, y) : wfrec_rel R (%f x. F (cut f R x) x)"
```

THE $x.P\ x$ picks the unique a such that $P\ a$ holds, if it exists.
We don't care what it does otherwise (see [HOL.thy](#)).

wfrec Explained

$$\begin{aligned} wfrec\ R\ F &\equiv \\ \lambda x. \text{THE } y. (x, y) &\in wfrec_rel\ R\ (\lambda f x. \textcolor{red}{F}\ (\textcolor{red}{cut}\ f\ R\ x)\ x) \end{aligned}$$

We don't care what this means for arbitrary R and F .

But if R is well-founded and F is coherent, then

$F\ (\textcolor{red}{cut}\ f\ R\ x)\ x = F\ f\ x$ (by (5)), and so

$\lambda f x. \textcolor{red}{F}\ (\textcolor{red}{cut}\ f\ R\ x)\ x = F$, and so

$\lambda x. \text{THE } y. (x, y) \in wfrec_rel\ R\ (\lambda f x. \textcolor{red}{F}\ (\textcolor{red}{cut}\ f\ R\ x)\ x)$ is the function defined by $wfrec_rel\ R\ F$ in the obvious way.

$wfrec\ R\ F$ is the recursive function defined by functional F .

The “Fixpoint” Theorem

$$\text{wfrec} \quad wf \ r \implies wfrec \ r \ H \ a = H(\text{cut}(wfrec \ r \ H) \ r \ a) \ a$$

Note that `wfrec` is used here both as a name of a constant (defined [above](#)) and a theorem.

So if r is well-founded and H is coherent, we have

$$wfrec \ r \ H \ a = H(wfrec \ r \ H) \ a$$

Theorem states that $wfrec$ is like a fixpoint combinator (disregarding the additional argument r).

Thus we can do using $wfrec$ what we would have liked to do using Y .

Example for *wfrec*: Natural Numbers

The constant *wfrec* provides **the** mechanism/support for defining recursive functions. We illustrate this using `nat`, the type of natural numbers (**pretending we have it**).

wfrec is applied to a well-founded order and a functional to define a function.

Example for *wfrec*: Natural Numbers

The constant *wfrec* provides **the** mechanism/support for defining recursive functions. We illustrate this using `nat`, the type of natural numbers (**pretending we have it**).

wfrec is applied to a well-founded order and a functional to define a function.

First, define predecessor relation:

```
constdefs
```

```
  pred_nat :: "(nat * nat) set"
```

```
  pred_nat_def "pred_nat == {(m,n). n = Suc m}"
```

How would you define **addition or subtraction**?

Defining Division and Modulus

```
div :: ['a::div, 'a] => 'a           (infixl 70)
"m div n == wfrec (pred_nat^+)
(%f j. if j<n | n=0 then 0 else Suc (f (j-n))) m"
```

```
mod :: ['a::div, 'a] => 'a           (infixl 70)
"m mod n == wfrec (pred_nat^+)
(%f j. if j<n | n=0 then j else f (j-n)) m"
```

Here, `div` is a syntactic class for which division is defined (don't worry about it). We **know how to define** `—`.

The functions are recursive in **one** argument (**just like add**).

Theorems of the Example

`wf_pred_nat` *wf pred_nat*

`mod_if` $m \bmod n =$
 (if $m < n$ then m else $(m - n) \bmod n$)

`div_if` $0 < n \implies m \operatorname{div} n =$
 (if $m < n$ then 0 else $\operatorname{Suc}((m - n) \operatorname{div} n)$)

Theorems of the Example

`wf_pred_nat` *wf pred_nat*

`mod_if` $m \bmod n =$
 (if $m < n$ then m else $(m - n) \bmod n$)

`div_if` $0 < n \implies m \operatorname{div} n =$
 (if $m < n$ then 0 else $\operatorname{Suc}((m - n) \operatorname{div} n)$)

This is very similar to functional programming code and hence lends itself to real **computations** (rewriting), as opposed to only doing **proofs**.

Conclusion on Well-founded Recursion

Well-founded recursion allows us to define recursive functions in HOL and thus **reason about** computations.

But: We have used the **THE** operator (similar to the **ϵ -operator**): it is not clear that there is always an effective way to compute the recursive functions.

Conclusion on Well-founded Recursion

Well-founded recursion allows us to define recursive functions in HOL and thus **reason about** computations.

But: We have used the **THE** operator (similar to the **ϵ -operator**): it is not clear that there is always an effective way to compute the recursive functions.

It is not clear that we can actually **do** computations. This is only **possible in particular cases**, where we derived recursive **theorems** that can be used for rewriting just like in a functional programming language.

HOL is a logic, not a programming language.

Isabelle Package for Primitive Recursion

For **primitive recursion**, finding a well-founded ordering is simple enough for **automation**!

Isabelle Package for Primitive Recursion

For **primitive recursion**, finding a well-founded ordering is simple enough for **automation**!

Examples (use **nat** and **case-syntax**): . . .

Recursion and Arithmetic

primrec

add_0: "0 + n = n"

add_Suc: "Suc m + n = Suc (m + n)"

primrec

diff_0: "m - 0 = m"

diff_Suc: "m - Suc n =

(case m - n of 0 => 0 | Suc k => k)"

primrec

mult_0: "0 * n = 0"

mult_Suc: "Suc m * n = n + (m * n)"

Conclusion on Recursion and Induction

We are interested in recursion because **inductively defined sets** and **recursively defined functions** are solutions to recursive equations.

We cannot have general fixpoint operator Y , but we have, by conservative extension:

- Least fixpoints for defining sets;
- well-founded orders for defining functions.

Both concepts come with induction schemes (**lfp induction** and **definition of well-foundedness**) for proving properties of the defined objects.

Summary: Proof Support

The methodological overhead can be faced by powerful mechanical support, since many proof-tasks are routine. Isabelle offers extensibility for such routine task constructions. ►

More Detailed Explanations

Packages Rely on Lemmas

If you look around in the ML-files of the Isabelle/HOL library, you might not find any uses of `lfp_unfold`, so you may wonder: why is it important then? But you must bear in mind that the `package for inductive sets` relies on these lemmas.

This is a general insight about proven results in the library: Even though you might not find them being used in other ML-files, special packages of Isabelle/HOL might use those results.

Relation.thy

This file should be contained in your Isabelle distribution. In Freiburg it is

`/usr/local/isabelle/Isabelle/src/HOL/Relation.thy`

Wellfounded_Recursion.thy

This file should be contained in your Isabelle distribution. In Freiburg it is

`/usr/local/isabelle/Isabelle/src/HOL/Wellfounded_Recursion.thy`

but in older versions it used to be called

`/usr/local/isabelle/Isabelle/src/HOL/WF.thy`

Transitive_Closure.thy

This file should be contained in your Isabelle distribution. In Freiburg it is

`/usr/local/isabelle/Isabelle/src/HOL/Transitive_Closure.thy`

Defining r^* and r^+

r^* is the smallest set such that:

- $Id \subseteq r^*$;
- if $r' \subseteq r^*$ then $r' \cup r \circ r' \subseteq r^*$.

Or, in line with the schema for inductive definitions:

- $\emptyset \subseteq r^*$;
- if $r' \subseteq r^*$ then $(\lambda s. Id \cup (r \circ s))r' \subseteq r^*$.

The latter form corresponds to the definition in `Transitive_Closure.thy`.

The definition of r^+ is similar.

One Way of Understanding wf

For a moment, forget everything you have ever heard about proofs using induction! The definition of wf has the form

$$wf(r) \equiv \forall P. \phi(r, P) \rightarrow \forall x. P(x)$$

That is, it says: a relation r is well-founded if a certain scheme ϕ can be used to show a property P that holds for all x .

By the fact that this is a **constant definition** (conservative extension), it is immediately clear that this gives us a **correct method** of proving $\forall x. P(x)$. To prove $\forall x. P(x)$ for some given P , find some r such that $\forall P. \phi(r, P) \rightarrow \forall x. P(x)$ holds, and show $\phi(r, P)$.

Once again, this method is correct regardless of what ϕ is. Forget about induction!

But how is that possible? How is it ensured that only true statements can be proven, if the method is correct for any old ϕ ?

The point is this: The method is correct in principle, but it will typically not work unless ϕ is something sensible, e.g. an induction scheme as in the **actual definition of wf** . It will not work simply because we will fail to show either $\forall P. \phi(r, P) \rightarrow \forall x. P(x)$ or $\phi(r, P)$.

Is \emptyset well-founded?

The definition of wf is:

$$wf(r) \equiv (\forall P. (\forall x. (\forall y. (y, x) \in r \rightarrow P(y)) \rightarrow P(x)) \rightarrow (\forall x. P(x)))$$

Is \emptyset well-founded?

The definition of wf is:

Let's instantiate r to \emptyset .

$$wf(\emptyset) \equiv (\forall P. (\forall x. (\forall y. (y, x) \in \emptyset \rightarrow P(y)) \rightarrow P(x)) \rightarrow (\forall x. P(x)))$$

Is \emptyset well-founded?

The definition of wf is:

Let's instantiate r to \emptyset .

$$wf(\emptyset) \equiv (\forall P. (\forall x. (\forall y. False \rightarrow P(y)) \rightarrow P(x)) \rightarrow (\forall x. P(x)))$$

Is \emptyset well-founded?

The definition of wf is:

Let's instantiate r to \emptyset .

$$wf(\emptyset) \equiv (\forall P. (\forall x. (\forall y. True \rightarrow P(y)) \rightarrow P(x)) \rightarrow (\forall x. P(x)))$$

Is \emptyset well-founded?

The definition of wf is:

Let's instantiate r to \emptyset .

$$wf(\emptyset) \equiv (\forall P. (\forall x. \quad \text{True} \quad \rightarrow P(x)) \rightarrow (\forall x. P(x)))$$

Is \emptyset well-founded?

The definition of wf is:

Let's instantiate r to \emptyset .

$$wf(\emptyset) \equiv (\forall P. (\forall x. P(x) \rightarrow (\forall x. P(x))))$$

Is \emptyset well-founded?

The definition of wf is:

Let's instantiate r to \emptyset .

$$wf(\emptyset) \equiv (\forall P. True \quad)$$

Is \emptyset well-founded?

The definition of wf is:

Let's instantiate r to \emptyset .

$$wf(\emptyset) \equiv \text{True}$$

So the empty set is well-founded.

Expressing Absence of Infinite Chains

We will now try some ideas, work out their formalization as a formula, and then illustrate why the condition is either too weak or too strong, using an example. Finally, we will give the correct condition.

Bad Formalization of “Minimal Element”

In this attempt, we formalized the “minimal element **in** p ” as an x such that there is no y with $(x, y) \in p$. But this is a bad formalization since an **isolated element**, i.e., one that is completely unrelated to p , or even to r , would meet the definition.

In fact, this problem was already present for the previous attempt where we just required $\exists x. \forall y. (y, x) \notin r$ (i.e., r has a minimal element).

No Infinite Descending Chains

The final condition

$$(\forall Q . x \in Q \rightarrow (\exists z \in Q . \forall y . (y, z) \in r \rightarrow y \notin Q))$$

expresses the absence of infinite descending chains without explicitly using the concept of infinity.

It is a characterization of well-foundedness. One could say that the above formula expresses what well-foundedness **is**, while the “official” definition is somewhat indirect since it defines well-foundedness by **what one can do with it**.

Theorems of Wellfounded_Recursion.ML

The **theorems** we present here are proven in Wellfounded_Recursion.ML.

This file should be contained in your Isabelle distribution. In Freiburg it is

```
/usr/local/isabelle/Isabelle/src/HOL/Wellfounded_Recursion.M
```

but in older versions it used to be called

```
/usr/local/isabelle/Isabelle/src/HOL/WF.ML
```

induct_wf

As far as the induction principle is concerned, `induct_wf` states the same as the very [definition of `wf`](#). All that happens is that some explicit universal object-level quantifiers are [removed](#) and the according variables are (implicitly) universally quantified on the meta-level, and some [shifting](#) from object-level implications to meta-level implications using `mp`. This is why we dare say “logical massage”. See [Wellfounded_Recursion.ML](#).

Elementary Equivalences

For example $\neg\forall x.\phi = \exists x.\neg\phi$ or $\neg\neg\phi = \phi$, which hold because our reasoning is **classical**.

$\neg \exists w.(w, v) \in r^+$ in Detail

In the proof of $\exists x.\forall y.(y, x) \notin r^+$ we had the sub-proof

$$\frac{\neg \phi \quad \forall w.(w, v) \in r^+ \rightarrow \phi}{\neg \exists w.(w, v) \in r^+}$$

This sub-proof does not actually depend on ϕ , it would hold no matter what ϕ is (unlike the entire proof)

In detail, the sub-proof looks as follows:

$$\frac{\forall w.(w, v) \in r^+ \rightarrow \phi}{(w, v) \in r^+ \rightarrow \phi} \textit{spec}$$

In detail, the sub-proof looks as follows:

$$(w, v) \in r^+ \quad \frac{\forall w.(w, v) \in r^+ \rightarrow \phi}{(w, v) \in r^+ \rightarrow \phi} \textit{spec}$$

In detail, the sub-proof looks as follows:

$$\frac{(w, v) \in r^+ \quad \frac{\forall w.(w, v) \in r^+ \rightarrow \phi}{(w, v) \in r^+ \rightarrow \phi} \text{spec}}{\phi} \text{mp}$$

In detail, the sub-proof looks as follows:

$$\begin{array}{c} \exists w.(w, v) \in r^+ \quad \frac{(w, v) \in r^+ \quad \frac{\forall w.(w, v) \in r^+ \rightarrow \phi}{(w, v) \in r^+ \rightarrow \phi} \textit{spec}}{\phi} \textit{mp} \end{array}$$

In detail, the sub-proof looks as follows:

$$\begin{array}{c}
 \frac{\frac{\frac{\exists w.(w, v) \in r^+}{\phi} \quad \frac{\frac{[(w, v) \in r^+]^2 \quad \frac{\forall w.(w, v) \in r^+ \rightarrow \phi}{(w, v) \in r^+ \rightarrow \phi} \text{spec}}{\phi} \text{mp}}{\phi} \text{exists}E^2}
 \end{array}$$

In detail, the sub-proof looks as follows:

$$\begin{array}{c}
 \neg\phi \quad \frac{\exists w.(w, v) \in r^+}{\phi} \quad \frac{\frac{[(w, v) \in r^+]^2 \quad \frac{\forall w.(w, v) \in r^+ \rightarrow \phi}{(w, v) \in r^+ \rightarrow \phi} \text{spec}}{\phi} \text{mp}}{\phi} \text{exists}E^2
 \end{array}$$

In detail, the sub-proof looks as follows:

$$\begin{array}{c}
 \frac{\neg\phi}{\text{False}} \quad \frac{\frac{\frac{\exists w.(w, v) \in r^+}{\phi} \text{existsE}^2 \quad \frac{\frac{[(w, v) \in r^+]^2 \quad \frac{\frac{\forall w.(w, v) \in r^+ \rightarrow \phi}{(w, v) \in r^+ \rightarrow \phi} \text{spec}}{(w, v) \in r^+ \rightarrow \phi} \text{mp}}{\phi} \text{mp}}{\phi} \text{existsE}^2}}{\phi} \text{notE}
 \end{array}$$

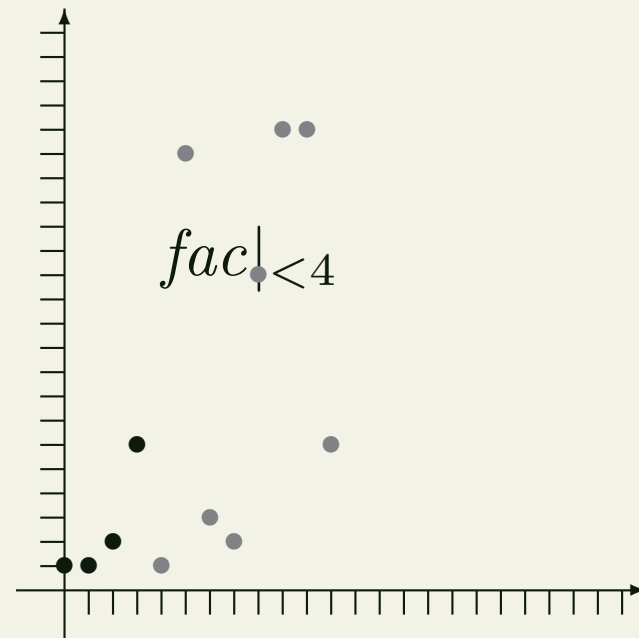
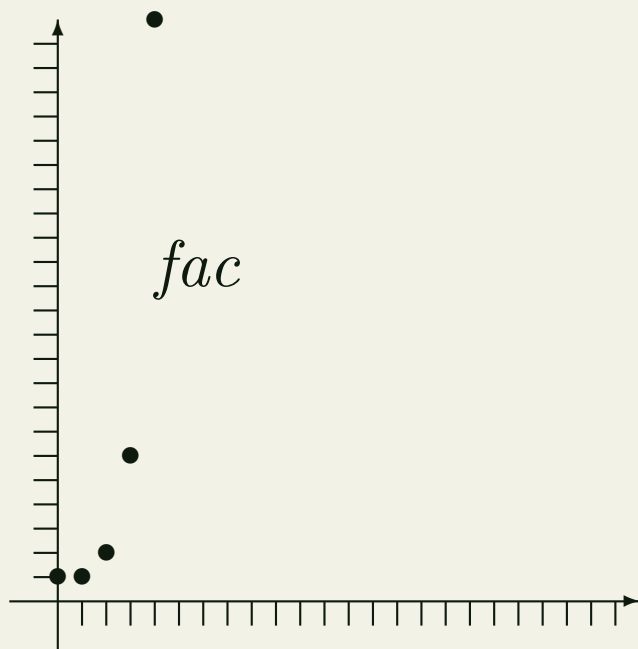
$$\begin{array}{c}
\frac{\forall w.(w, v) \in r^+ \rightarrow \phi}{(w, v) \in r^+ \rightarrow \phi} \textit{spec} \\
\frac{[(w, v) \in r^+]^2 \quad (w, v) \in r^+ \rightarrow \phi}{\phi} \textit{mp} \\
\frac{[\exists w.(w, v) \in r^+]^1 \quad \phi}{\phi} \textit{existsE}^2 \\
\frac{\neg \phi \quad \phi}{False} \textit{notE} \\
\frac{False}{\neg \exists w.(w, v) \in r^+} \textit{notI}^1
\end{array}$$

Splitting =

By this we simply mean to split a proof of $\phi = \psi$ into two proofs $\phi \implies \psi$ and $\psi \implies \phi$.

What is arbitrary?

For the construction we have in mind, it would be fine that $f|_{<a}$ be a function that is like f on all values $< a$, and arbitrary elsewhere. E.g., $fac|_{<4}$ could be



However, such a $fac|_{<4}$ could not be in a **model** for HOL (with the extensions we consider here). The way that arbitrary elements are

formalized in `HOL.thy`, it turns out that in any model and for each type, there must be **one specific** domain element for the constant `arbitrary` (you don't have to understand why this is so). That is, in different models we could have different ones, but within each model the element must be a specific one. Since the value of $fac|_{<4}$ is “arbitrary” for all arguments ≥ 4 , this means that in each model, this value must be the same for all arguments ≥ 4 , ruling out the function above.

Of course, these are considerations taking place only in our heads. In the actual deduction machinery, one never constructs these “arbitrary” terms.

Relation Is Function

When we say that a binary relation $r : \tau \times \sigma$ is in fact a function, we mean that for $t : \tau$, there is exactly one $s : \sigma$ such that $(t, s) \in r$.

THE

The operator THE is similar to the [Hilbert operator](#), but it returns the unique element having a certain property rather than an arbitrary one. The Isabelle formalization of HOL nowadays heavily relies on THE rather than the Hilbert operator.

Define Addition and Subtraction

```
add :: [nat, nat] => nat    (infixl 70)
"m add n == wfrec (pred_nat^+)
  (%f j. if j=0 then n else Suc (f (pred j))) m"
```

Here we suppose that we have a predecessor function `pred`. The implementation in Isabelle [is different](#), but conceptually, the above is a definition of the `add` function.

Note that `add` is a function of type $\text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$ (written infix), but it is only recursive in one argument, namely the first one.

You may be confused about this and wonder: how do I know that it is the first? Is this some Isabelle mechanism saying that it is always the first? The answer is: no. You must look at the two sides in isolation. On

the right-hand side, we have

```
wfrec (pred_nat^+)  
  (%f j. if j=0 then n else Suc (f (pred j)))
```

By the definitions (of *wfrec* most importantly), this expression is a function of type $nat \rightarrow nat$, namely the function that adds n (which is not known looking at this expression alone; it occurs on the left-hand side) to its argument. The function is recursive in its argument (and hence not in n). Now, this function is applied to m . Therefore we say that the final function `add` is recursive in m but not in n .

Now look at subtraction:

```
subtract :: [nat, nat] => nat    (infixl 70)  
"m subtract n == wfrec (pred_nat^+)  
  (%f j. if j=0 then m else pred (f (pred j))) n"
```

Note that `subtract` is recursive in its **second** argument, simply because the right-hand side of the defining equation was constructed in a different way than for `add`.

Similar considerations apply for other binary functions defined by recursion in one argument.

Primitive Recursion

A function is **primitive recursive** if the recursion is based on the immediate predecessor w.r.t. the well-founded order used (e.g., the predecessor on the natural numbers, as opposed to any arbitrary smaller numbers).

This is not the same concept as used in the context of computation theory, where **primitive recursive** is in contrast to μ -recursive [LP81].

Automating Recursion

The `primrec` syntax provides a convenient front-end for defining **primitive recursive** functions.

Isabelle will guess a well-founded ordering to use. E.g. for functions on the natural numbers, it will use the usual $<$ ordering.

Arithmetic

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Motivation

Current stage of our course:

- On the basis of conservative embeddings, **set theory** can be built safely.
- **Inductive sets** can be defined using **least fixpoints** and suitably **supported by Isabelle**.
- **Well-founded orderings** can be defined without referring to **infinity**. Recursive functions can be based on these. Needs inductive sets though. **Support by Isabelle** provided.

Motivation

Current stage of our course:

- On the basis of conservative embeddings, **set theory** can be built safely.
- **Inductive sets** can be defined using **least fixpoints** and suitably **supported by Isabelle**.
- **Well-founded orderings** can be defined without referring to **infinity**. Recursive functions can be based on these. Needs **inductive sets** though. **Support by Isabelle** provided.

Next important topic: **arithmetic**.

Which Approach to Take?

- Purely definitional?

Which Approach to Take?

- Purely definitional?

Not possible with eight basic rules (cannot enforce infinity of HOL model)!

Which Approach to Take?

- Purely definitional?
Not possible with eight basic rules (cannot enforce infinity of HOL model)!
- Heavily axiomatic? I.e., we state natural numbers by **Peano axioms** and claim analogous axioms for any other number type?

Which Approach to Take?

- Purely definitional?
Not possible with eight basic rules (cannot enforce infinity of HOL model)!
- Heavily axiomatic? I.e., we state natural numbers by **Peano axioms** and claim analogous axioms for any other number type?
Insecure!

Which Approach to Take?

- Purely definitional?
Not possible with eight basic rules (cannot enforce infinity of HOL model)!
- Heavily axiomatic? I.e., we state natural numbers by **Peano axioms** and claim analogous axioms for any other number type?
Insecure!
- Minimally axiomatic? We construct an infinite set, and define numbers etc. as **inductive subset**?

Which Approach to Take?

- Purely definitional?

Not possible with eight basic rules (cannot enforce infinity of HOL model)!

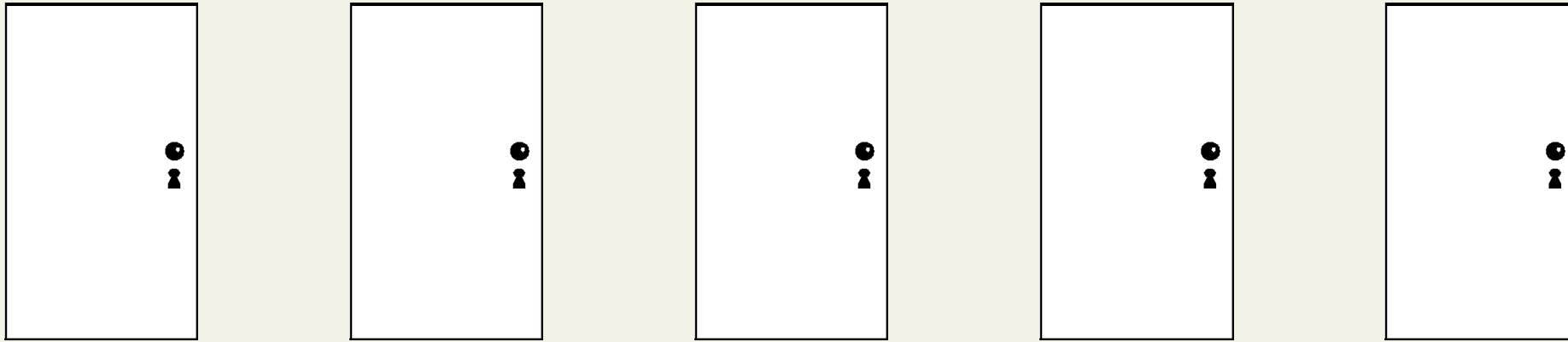
- Heavily axiomatic? I.e., we state natural numbers by **Peano axioms** and claim analogous axioms for any other number type?

Insecure!

- Minimally axiomatic? We construct an infinite set, and define numbers etc. as **inductive subset**?

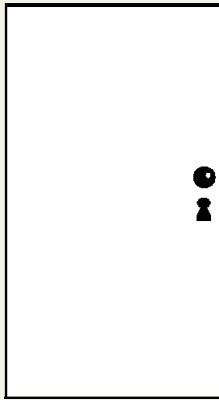
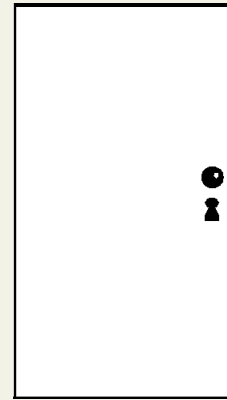
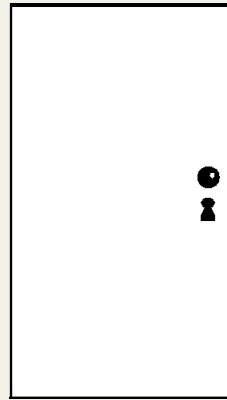
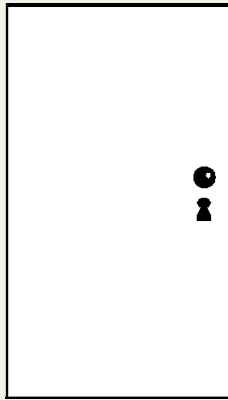
Yes. Finally use **infinity** axiom.

What is Infinity? Cantor's Hotel



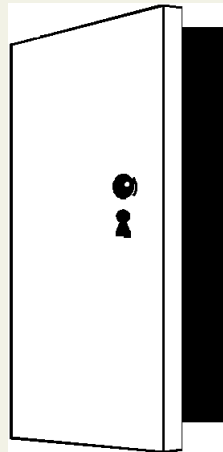
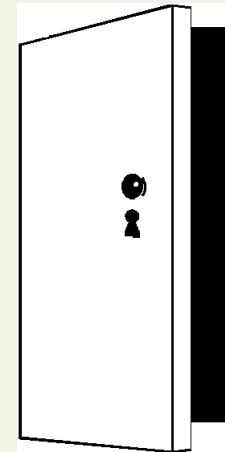
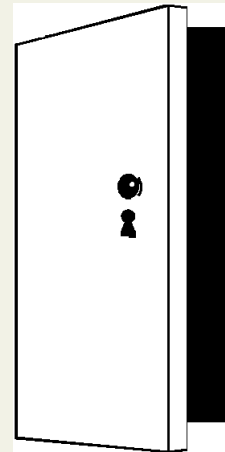
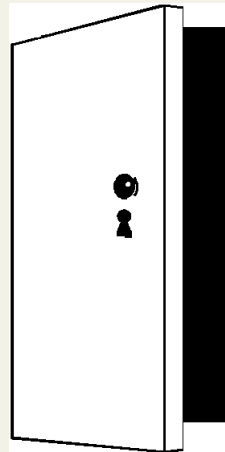
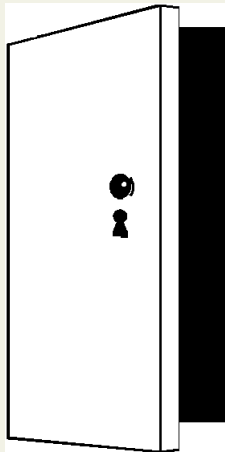
Cantor's hotel has infinitely many rooms.

What is Infinity? Cantor's Hotel



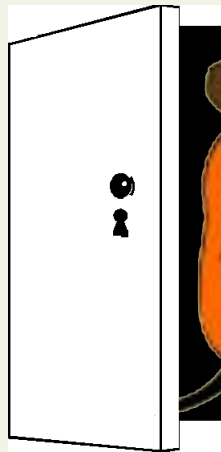
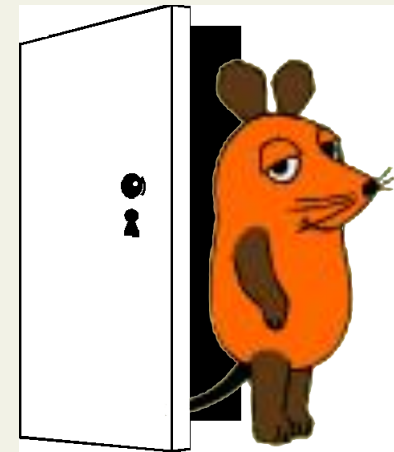
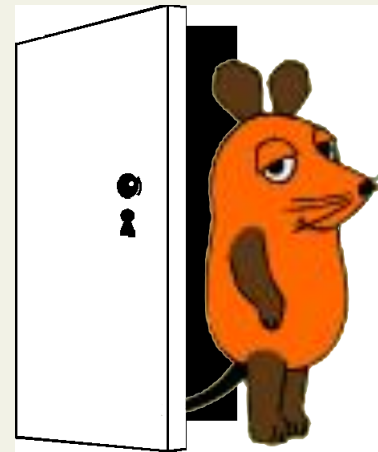
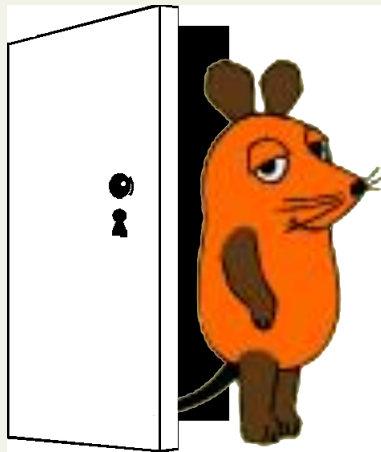
Cantor's hotel has infinitely many rooms. New guest arrives.

What is Infinity? Cantor's Hotel



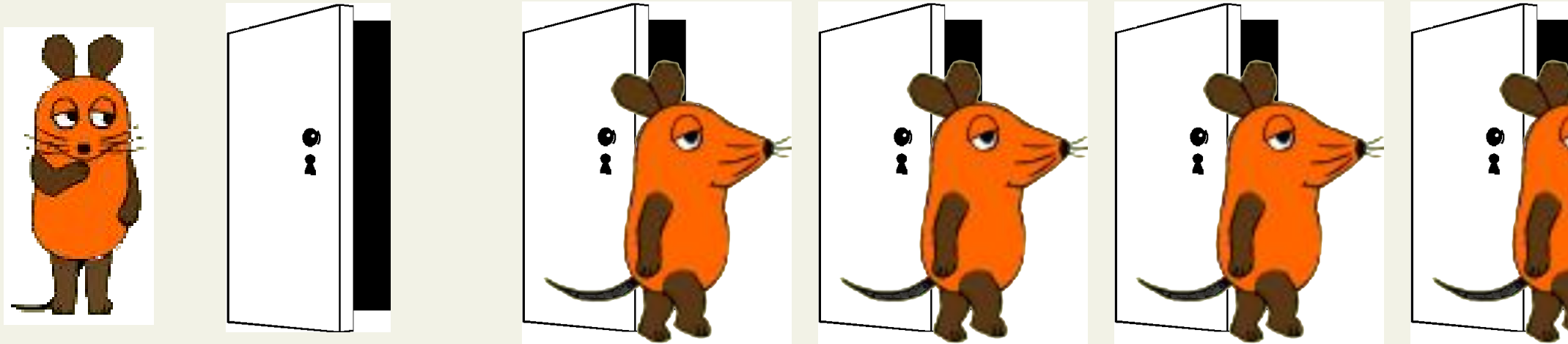
Cantor's hotel has infinitely many rooms. New guest arrives.
The doors open,

What is Infinity? Cantor's Hotel



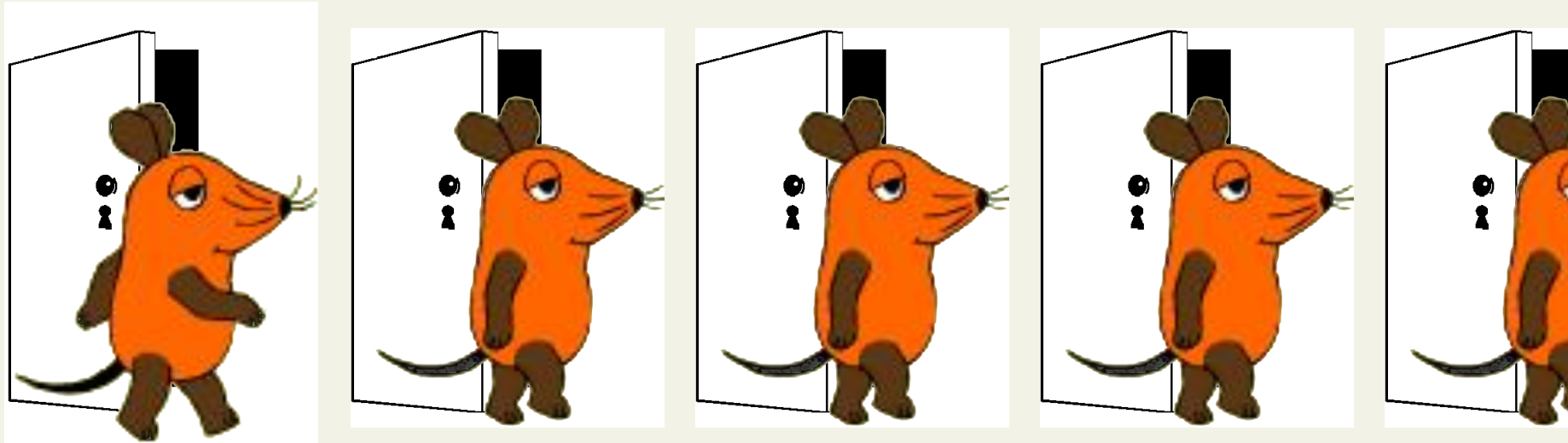
Cantor's hotel has infinitely many rooms. New guest arrives. The doors open, and all guests come out of their rooms.

What is Infinity? Cantor's Hotel



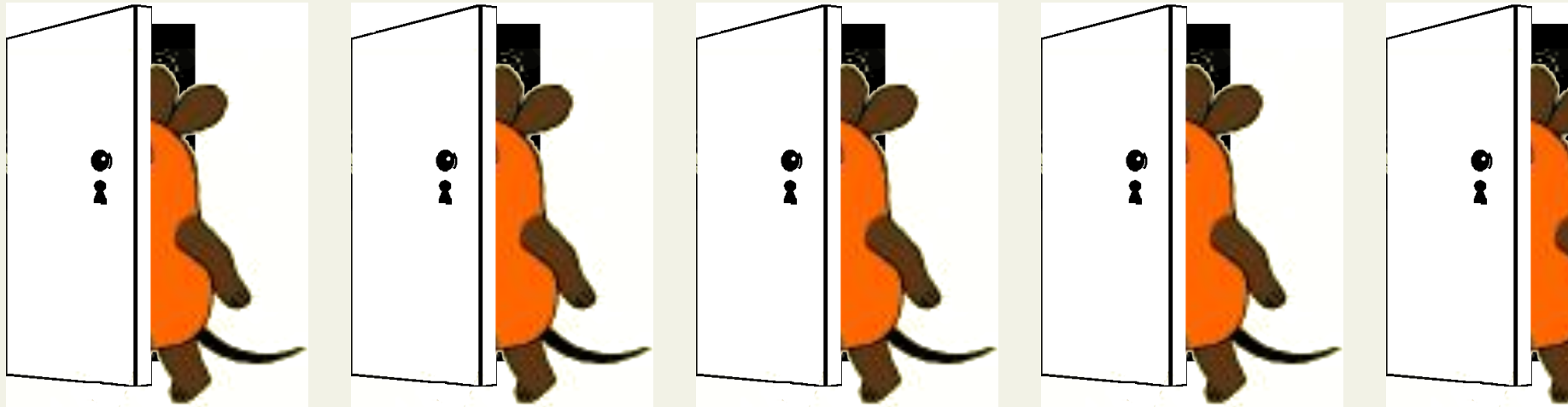
Cantor's hotel has infinitely many rooms. New guest arrives.
The doors open, and all guests come out of their rooms.
They move one room forward,

What is Infinity? Cantor's Hotel



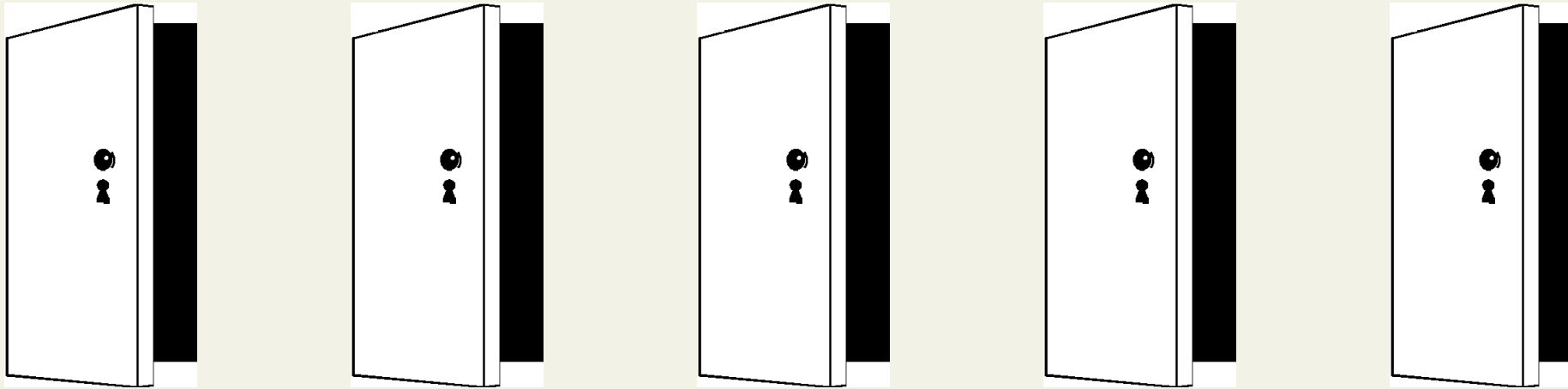
Cantor's hotel has infinitely many rooms. New guest arrives. The doors open, and all guests come out of their rooms. They **move one room forward**, the new guest walks towards the first room,

What is Infinity? Cantor's Hotel



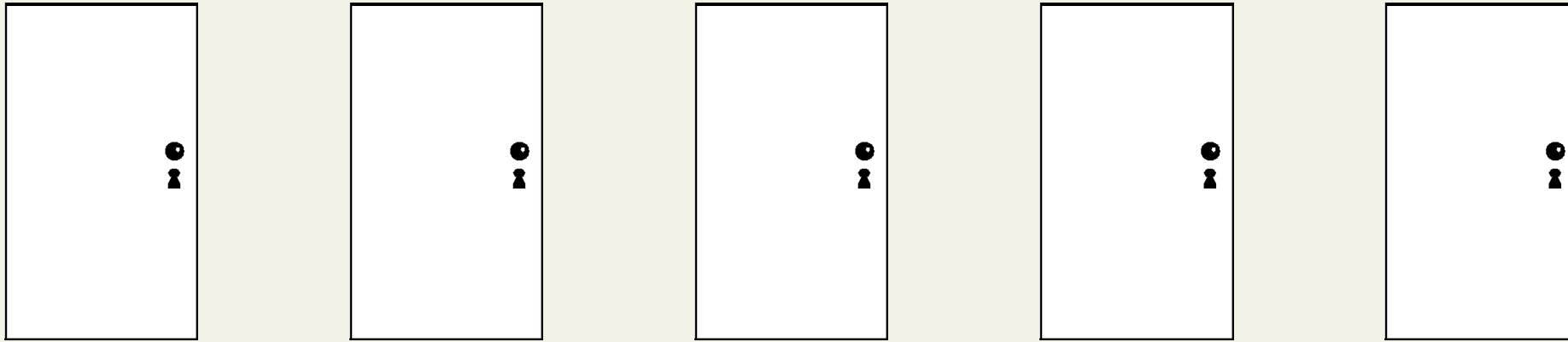
Cantor's hotel has infinitely many rooms. New guest arrives. The doors open, and all guests come out of their rooms. They **move one room forward**, the new guest walks towards the first room, they turn around,

What is Infinity? Cantor's Hotel



Cantor's hotel has infinitely many rooms. New guest arrives. The doors open, and all guests come out of their rooms. They **move one room forward**, the new guest walks towards the first room, they turn around, enter their new rooms.

What is Infinity? Cantor's Hotel



Cantor's hotel has infinitely many rooms. New guest arrives. The doors open, and all guests come out of their rooms. They **move one room forward**, the new guest walks towards the first room, they turn around, enter their new rooms. The doors close, all guests are accomodated.

Axiom of Infinity

The axiomatic core of datatypes (and hence, numbers):

$$\frac{}{\exists f :: (ind \rightarrow ind). \text{injective } f \wedge \text{surjective } f} \text{infty}$$

where

$$\text{injective } f = \forall xy. f\ x = f\ y \rightarrow x = y$$

$$\text{surjective } f = \forall y. \exists x. y = f\ x$$

Forces *ind* to be “infinite type” (called “*I*” in [Chu40]).

We will see soon how this is done in Isabelle.

Type-Closed Conservative Extensions

Why must conservative extensions be type-closed? (the standard example)

Consider $H \equiv \exists f :: \alpha \Rightarrow \alpha. \text{injective } f \wedge \text{surjective } f$

Type-Closed Conservative Extensions

Why must conservative extensions be type-closed? (the standard example)

Consider $H \equiv \exists f :: \alpha \Rightarrow \alpha. \text{injective } f \wedge \text{surjective } f$

Then the type of H is *bool*, but H contains a subterm of type $\alpha \Rightarrow \alpha$ (H is not type-closed).

Then we could reason as follows . . .

Type-Closed Conservative Extensions (2)

$(H \equiv \exists f :: \alpha \Rightarrow \alpha. \text{injective } f \wedge \text{surjective } f)$

Type-Closed Conservative Extensions (2)

$(H \equiv \exists f :: \alpha \Rightarrow \alpha. \text{injective } f \wedge \text{surjective } f)$

$$\begin{aligned} H &= H && \text{holds by } \textit{refl} \\ \Rightarrow \exists f :: \textit{bool} \Rightarrow \textit{bool}. \textit{inj } f \wedge \textit{sur } f &= \\ &\quad \exists f :: \textit{ind} \Rightarrow \textit{ind}. \textit{inj } f \wedge \textit{sur } f \\ \Rightarrow \textit{False} &= \textit{True} \\ \Rightarrow \textit{False} \end{aligned}$$

(unfolding H using two different type instantiations, and then using **axiom of infinity** and the fact that there are only finitely many functions on \textit{bool}).

Types Affect the Semantics

Type instantiations may change semantic values, and hence cause **inconsistency**!

This example was somewhat more concrete than our previous simpler example.

Natural Numbers: Nat.thy

consts

Zero_Rep :: ind

Suc_Rep :: "ind => ind"

axioms

inj_Suc_Rep: "inj Suc_Rep"

Suc_Rep_not_Zero_Rep: "Suc_Rep x ~= Zero_Rep"

So the *axiom of infinity* is formulated by defining a constant *Suc_Rep* having the two required properties.

inj and *sur* are defined in *Fun.thy*.

Think of Zero_Rep, Suc_Rep as **provisional** 0, successor.

Defining the Set Nat

Want to define **new type** nat . How?

Defining the Set Nat

Want to define **new type** nat . How?

Must define a set **isomorphic** to the natural numbers. How?

Defining the Set *Nat*

Want to define **new type** *nat*. How?

Must define a set **isomorphic** to the natural numbers. How?

By induction using the **inductive syntax**:

```
inductive Nat
```

```
intros
```

```
  Zero_RepI: "Zero_Rep : Nat"
```

```
  Suc_RepI: "i : Nat ==> Suc_Rep i : Nat"
```

Translated by Isabelle to:

$$Nat = lfp (\lambda X. \{Zero_Rep\} \cup (Suc_Rep \circ X))$$

Defining the Type *nat*

Now we have the **set** *Nat*. What next?

Defining the Type *nat*

Now we have the **set** *Nat*. What next?

Define the **type** *nat*, isomorphic to *Nat*, using the **typedef** syntax:

```
typedef (open Nat)  
  nat = "Nat" by (rule exI, rule Nat.Zero_RepI)
```

After **these two steps** we have the type *nat*.

Constants in *nat*

Moreover, **define**:

consts

Suc :: "nat => nat"

pred_nat :: "(nat * nat) set"

defs

Zero_nat_def: "0 == Abs_Nat Zero_Rep"

Suc_def: "Suc ==
(%n. Abs_Nat (Suc_Rep (Rep_Nat n)))"

pred_nat_def: "pred_nat == {(m, n). n = Suc m}"

Some Theorems in Nat

`nat_induct` $\llbracket P\ 0; \bigwedge n. P\ n \implies P\ (Suc\ n) \rrbracket \implies P\ n$
 $\llbracket \bigwedge x. P\ x\ 0; \bigwedge y. P\ 0\ (Suc\ y);$
`diff_induct` $\bigwedge xy. P\ x\ y \implies P\ (Suc\ x)\ (Suc\ y) \rrbracket$
 $\implies P\ m\ n$

We can now exploit that *nat* is defined based on a **set** defined using **least fixpoints**. In particular, `nat_induct` follows (**but not “automatically”!**) from the **induct** theorem of **Lfp**.

Nat and Well-Founded Orders

Examples of theorems involving well-founded orders:

`wf_pred_nat` *wf pred_nat*

`less_linear` $m < n \vee m = n \vee n < m$

`Suc_less_SucD` $Suc\ m < Suc\ n \implies m < n$

Using Primitive Recursion

`Nat.thy` defines rich theory on *nat*. Uses `primrec` syntax for defining recursive functions, and `case` construct.

```
primrec
```

```
  add_0      "0 + n = n"
```

```
  add_Suc    "Suc m + n = Suc(m + n)"
```

```
primrec
```

```
  diff_0     "m - 0 = m"
```

```
  diff_Suc   "m - Suc n =
```

```
    (case m - n of 0 => 0 | Suc k => k)"
```

```
primrec
```

```
  mult_0     "0 * n = 0"
```

```
  mult_Suc   "Suc m * n = n + (m * n)"
```

Some Theorems in Nat

add_0_right $m + 0 = m$

add_ac $m + n + k = m + (n + k)$

$$m + n = n + m$$

$$x + (y + z) = y + (x + z)$$

mult_ac $m * n * k = m * (n * k)$

$$m * n = n * m$$

$$x * (y * z) = y * (x * z)$$

Note **third part** of `add_ac`, `mult_ac`, respectively.

Technically, `add_ac` and `mult_ac` are lists of **thm**'s.

Proof of add_0_right

$$m + 0 = m$$

add_0_right

Proof of add_0_right

$$n + 0 = n$$

$$\frac{\frac{\text{add_0}}{0 + 0 = 0} \quad \text{Suc } n + 0 = \text{Suc } n}{m + 0 = m} \text{nat_induct}$$

Proof of add_0_right

$$\begin{array}{c}
 \text{add_0} \\
 \hline
 0 + 0 = 0
 \end{array}
 \quad
 \begin{array}{c}
 \text{add_Suc} \\
 \hline
 \text{Suc } n + 0 = \text{Suc}(n + 0) \\
 \hline
 \text{Suc}(n + 0) = \text{Suc } n + 0
 \end{array}
 \quad
 \begin{array}{c}
 \text{sym} \\
 \hline
 \text{Suc}(n + 0) = \text{Suc } n + 0
 \end{array}
 \quad
 \begin{array}{c}
 n + 0 = n \\
 \hline
 \text{Suc}(n + 0) = \text{Suc } n
 \end{array}
 \quad
 \begin{array}{c}
 \text{arg_cong} \\
 \hline
 \text{Suc}(n + 0) = \text{Suc } n
 \end{array}
 \quad
 \begin{array}{c}
 \text{subst} \\
 \hline
 \text{Suc } n + 0 = \text{Suc } n
 \end{array}
 \quad
 \begin{array}{c}
 \text{nat_induct} \\
 \hline
 m + 0 = m
 \end{array}$$

Note that $\text{Suc } n + 0 = \text{Suc}(n + 0)$ is an instance of $\text{Suc } m + n = \text{Suc}(m + n)$.

Proof of add_0_right

$$\begin{array}{c}
 \text{add_0} \\
 \hline
 0 + 0 = 0
 \end{array}
 \quad
 \begin{array}{c}
 \text{add_Suc} \\
 \hline
 \text{Suc } n + 0 = \text{Suc}(n + 0)
 \end{array}
 \quad
 \text{sym}
 \quad
 \begin{array}{c}
 [n + 0 = n]^1 \\
 \hline
 \text{Suc}(n + 0) = \text{Suc } n
 \end{array}
 \quad
 \text{arg_cong}
 \quad
 \text{subst}
 \quad
 \begin{array}{c}
 \text{Suc } n + 0 = \text{Suc } n \\
 \hline
 m + 0 = m
 \end{array}
 \quad
 \text{nat_induct}^1$$

Note that $\text{Suc } n + 0 = \text{Suc}(n + 0)$ is an instance of $\text{Suc } m + n = \text{Suc}(m + n)$.

Integers

The integers are implemented as equivalence classes over *nat nat*.

```
IntDef = Equiv + NatArith +  
constdefs
```

```
  intrel :: "(nat * nat) * (nat * nat) set"  
  "intrel == {p. EX x1 y1 x2 y2.  
    p=((x1::nat,y1),(x2,y2)) & x1+y2 = x2+y1}"
```

```
typedef (Integ)  
  int = "UNIV//intrel"   (quotient_def)
```

Some Theorems in IntArith

$$\text{zminus_zadd_distrib} \quad -(z + w) = -z + -w$$

$$\text{zminus_zminus} \quad -(-z) = z$$

$$\text{zadd_ac} \quad z1 + z2 + z3 = z1 + (z2 + z3)$$

$$z + w = w + z$$

$$x + (y + z) = y + (x + z)$$

$$\text{zmult_ac} \quad z1 * z2 * z3 = z1 * (z2 * z3)$$

$$z * w = w * z$$

$$z1 * (z2 * z3) = z2 * (z1 * z3)$$

Compare to *nat* theorems.

Further Number Theories

- Binary Integers (`Integ/Bin.thy`, for fast computation)
- Rational Numbers (`Real/PRat.thy`)
- **Reals** (`Real/PReal.thy`: based on Dedekind-cuts of rationals [?])
- **Hyperreals** (`Real/RealDef.thy` for non-standard analysis)
- Machine numbers (floats); see work for Intel's PentiumIV; built in HOL-light [[Har98](#), [Har00](#)]

Conclusion on Arithmetic

Using conservative extensions in HOL, we can build

- the naturals (as type definition based on *ind*), and
- higher number theories (via equivalence construction).

Conclusion on Arithmetic

Using conservative extensions in HOL, we can build

- the naturals (as type definition based on *ind*), and
- higher number theories (via equivalence construction).

Potential for

- analysis of processor arithmetic units, and
- function analysis in HOL (combination with computer algebra systems such as Mathematica).

Future: analysis of hybrid systems.

The methodological overhead can be tackled by powerful mechanical support, since many proof-tasks are routine. ►

More Detailed Explanations

Well-Founded Recursion and Sets

What's the role of sets in defining well-founded recursion? It's the fact that well-founded recursion is based on well-founded orders, which are **relations**, which are sets.

Enforcing Infinity

Our intuition/knowledge about arithmetics clearly requires that there are infinite sets, e.g., the set of infinite numbers. Technically, the HOL model of the set of natural numbers must be an infinite set, otherwise we would not be willing to say that have “modeled” arithmetic.

The Peano Axioms

The Peano axioms include

- $0 \in \text{nat}$
- $\forall x. x \in \text{nat} \rightarrow \text{Suc}(x) \in \text{nat}$
- $\forall x. \text{Suc}(x) \neq 0$
- $\forall P. (P(0) \wedge \forall n. (P(n) \rightarrow P(\text{Suc}(n)))) \rightarrow \forall n. P(n)$

Successors on Rooms

This means, there must be a successor function on rooms. To each room, it assigns the “next” room.

injective **and** *surjective*

These constants (actually called *inj* and *sur*) are defined in `Fun.thy`.

Is it Necessary to Add an Axiom?

Note that theoretically, it is not needed to add the infinity axiom (or some [equivalent formulation](#)) to HOL. Instead one could add the infinity axiom as premise to each arithmetic theorem that one wants to prove. However this would not be a viable approach since the resulting formulas would be very, very complicated.

Numbers as Datatype

The natural numbers can be built as an algebraic datatype by having a constant 0 and a term constructor Suc (for **successor**).

inj and *sur*

We use *inj* and *sur* as abbreviations for *injective* and *surjective*.

A Type Definition Based on an Inductive Set

Note the two ingredients for defining the type `nat`:

- An **inductively defined set** `Nat`, i.e., a set defined as fixpoint of a monotone function. In Isabelle (`Nat.thy`), the **inductive** syntax is used for this purpose. This **automatically** generates an induction rule for the set.
- A **type definition** based on this set, defined using the **typedef** syntax. Recall that this process automatically generates the two constants `Abs_Nat` and `Rep_Nat`.

But note: the induction theorem is not inherited automatically. More precisely, the `typedef` syntax does not cause the type `nat` to inherit the inductive theorem of the set `Nat`. The theorem `nat_induct` is explicitly proven in `Nat.thy`.

Constructor Abstraction

Based on the generic constants `Abs_Nat` and `Rep_Nat`, we define all the constants that we need to work conveniently with `nat`, most importantly, `0` and `Suc`.

Nat.thy

This file should be contained in your Isabelle distribution. In Freiburg it is

`/usr/local/isabelle/Isabelle/src/HOL/Nat.thy`

The case Statement for nat

The case statement for nat is a function of type $\text{nat} \Rightarrow (\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{nat} \Rightarrow \text{nat}$. `case z f n` is defined as follows (using a common mathematical notation):

$$\text{case } z \ f \ n = \begin{cases} z & \text{if } n = 0 \\ f \ k & \text{if } n = \text{Suc } k \end{cases}$$

The syntax

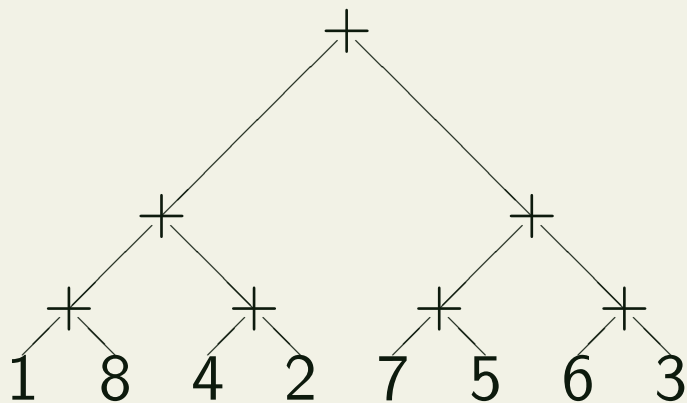
```
diff_Suc "m - Suc n = (case m - n of 0 => 0 | Suc k => k)"
```

used on the slide is a paraphrasing (“concrete syntax”) of the original (“abstract”) syntax. In the original syntax it would read `case 0 (λx.x) (n - m)`.

Left Commutation

The theorems $x + (y + z) = y + (x + z)$ and $x * (y * z) = y * (x * z)$ are called **left-commutation laws** and are crucial for (ordered) rewriting.

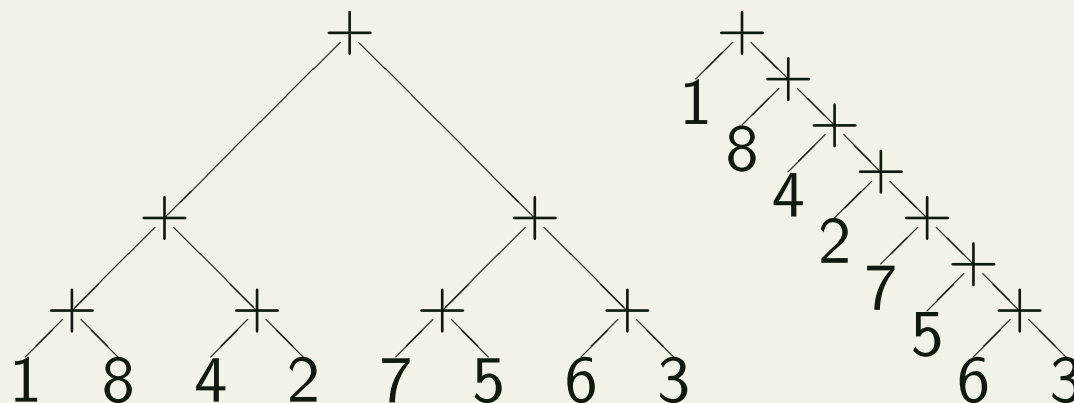
Suppose we have the term shown below.



Left Commutation

The theorems $x + (y + z) = y + (x + z)$ and $x * (y * z) = y * (x * z)$ are called **left-commutation laws** and are crucial for (ordered) rewriting.

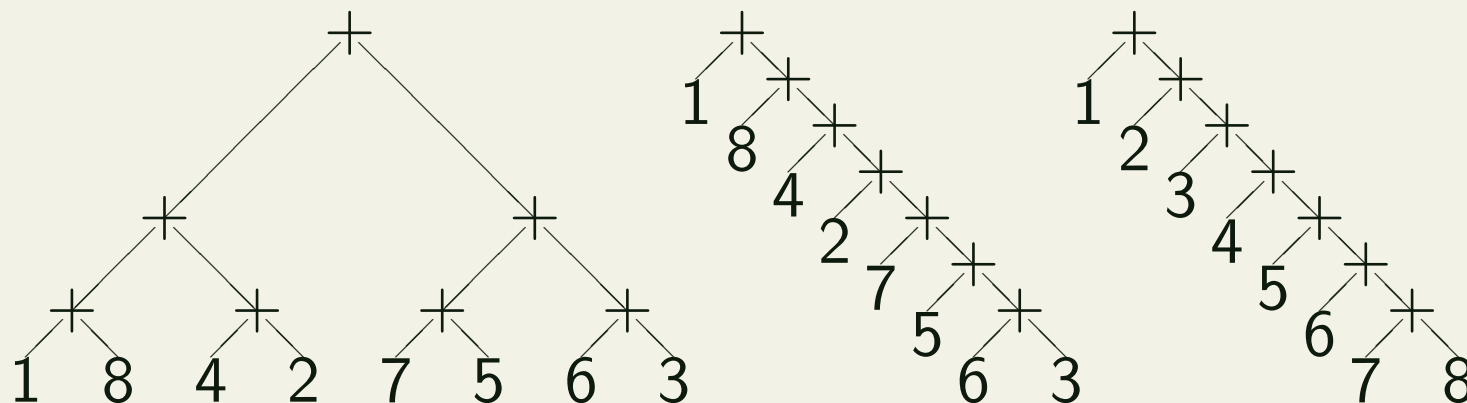
Suppose we have the term shown below. Using associativity ($m + n + k = m + (n + k)$) this will be rewritten to the second term.



Left Commutation

The theorems $x + (y + z) = y + (x + z)$ and $x * (y * z) = y * (x * z)$ are called **left-commutation laws** and are crucial for (ordered) rewriting.

Suppose we have the term shown below. Using associativity ($m + n + k = m + (n + k)$) this will be rewritten to the second term. Using left-commutation, this will be rewritten to the third term. This is a so-called **AC-normal form**, for an appropriately chosen **term ordering**.



IntDef.thy

The file should be contained in your Isabelle distribution. In Freiburg it is

`/usr/local/isabelle/Isabelle/src/HOL/Integ/IntDef.thy`

Note that `IntDef.thy` is in a special subdirectory.

Equivalence Classes

Recall the general concept of an **equivalence relation**. Generally, for a set S and an equivalence relation R defined on the set, one can define $S//R$, the **quotient of S w.r.t. R** .

$$S//R = \{A \mid A \subseteq S \wedge \forall x, y \in A. (x, y) \in R\}$$

That is, one partitions the set S into subsets such that each subset collects equivalent elements. This is a standard mathematical concept. We do not go into the Isabelle details here, but we explain how this works for the integers. One can view a pair (n, m) of natural numbers as representation of the integer $n - m$. But then (n, m) and (n', m') represent the same integer if and only if $n - m = n' - m'$, or equivalently, $n + m' = n' + m$. In this case (n, m) and (n', m') are said to be **equivalent**. The construction of the integer type is based on this

equivalence relation, called `intrel`. More precisely, the definition of the integers will be **based on** the set of **all** pairs of naturals (which corresponds to the `UNIV` constant on the type `nat nat`) **modulo** the equivalence `intrel`. In other words, it will be based on the quotient of the set of pairs of naturals w.r.t. `intrel`.

Reals According to Dedekind

The reals have been axiomatized by Dedekind by stating that a set R is partitioned into two sets A and B such that $R = A \cup B$ and for all $a \in A$ and $b \in B$, we have $a < b$. Now there is a number s such that $a \leq s \leq b$ for all $a \in A$ and $b \in B$. The irrational numbers are characterised by the fact that there exists exactly one such s . This axiomatization has been used as a basis for formalizing real numbers in Isabelle/HOL.

Hyperreals

In non-standard analysis, one works with sequences that are not necessarily converging. This is a relatively new field in mathematics and Isabelle/HOL has been successfully applied in it [?]. We just mention this here to say that Isabelle/HOL is used for “cutting-edge” mathematics and not just toy examples.

Hybrid Systems

Hybrid systems is a field in software engineering concerned with using finite automata for controlling physical systems such as ABS in cars etc.

Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Datatypes: Motivation

Last lecture: Construction of natural numbers.

How can we build “**datatypes**” (as in ML [Pau96])?

Can we generalize the natural number construction to arbitrary datatypes?

Datatypes: Motivation

Last lecture: Construction of natural numbers.

How can we build “**datatypes**” (as in ML [Pau96])?

Can we generalize the natural number construction to arbitrary datatypes?

- Well, yes — by **Gödelization**. But that wouldn't be automatable easily . . .

Datatypes: Motivation

Last lecture: Construction of natural numbers.

How can we build “datatypes” (as in ML [Pau96])?

Can we generalize the natural number construction to arbitrary datatypes?

- Well, yes — by Gödelization. But that wouldn't be automatable easily . . .
- Idea: Can we use *S-expressions* instead of *nat*?

Datatypes: Motivation

Last lecture: Construction of natural numbers.

How can we build “*datatypes*” (as in ML [Pau96])?

Can we generalize the natural number construction to arbitrary datatypes?

- Well, yes — by *Gödelization*. But that wouldn't be automatable easily . . .
- Idea: Can we use *S-expressions* instead of *nat*? Yes.

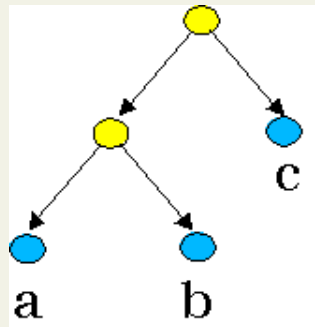
Caveat: We will only *sketch* the construction and we will *simplify*, meaning that the technical details will not be strictly correct! See *Datatype_Universe.thy* and [Wen99].

S-Expressions as Basis

In the end we want to have datatypes such as lists and trees. It turns out that LISP-like S-expressions are a datatype that is so rich that other datatypes can nicely be embedded in it. Since we have not got the concept of datatype yet, we must first represent S-expressions using constructs we already have.

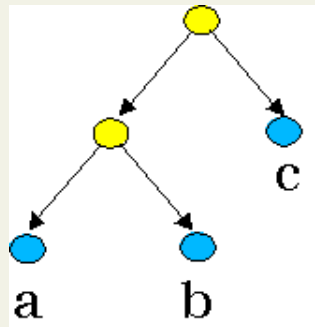
S-Expressions

LISP-like *S-expressions* are a kind of of binary trees. We call the type α *dtree*. This uses $\alpha + \text{nat}$.



S-Expressions

LISP-like *S-expressions* are a kind of binary trees. We call the type $\alpha \text{ dtree}$. This uses $\alpha + \text{nat}$.



This is encoded as a *set of “nodes”* (defined by their path from the root and a value in the leaves), e.g.:

$$\{(\langle 0, 0 \rangle, a), (\langle 0, 1 \rangle, b), (\langle 1 \rangle, c)\}$$

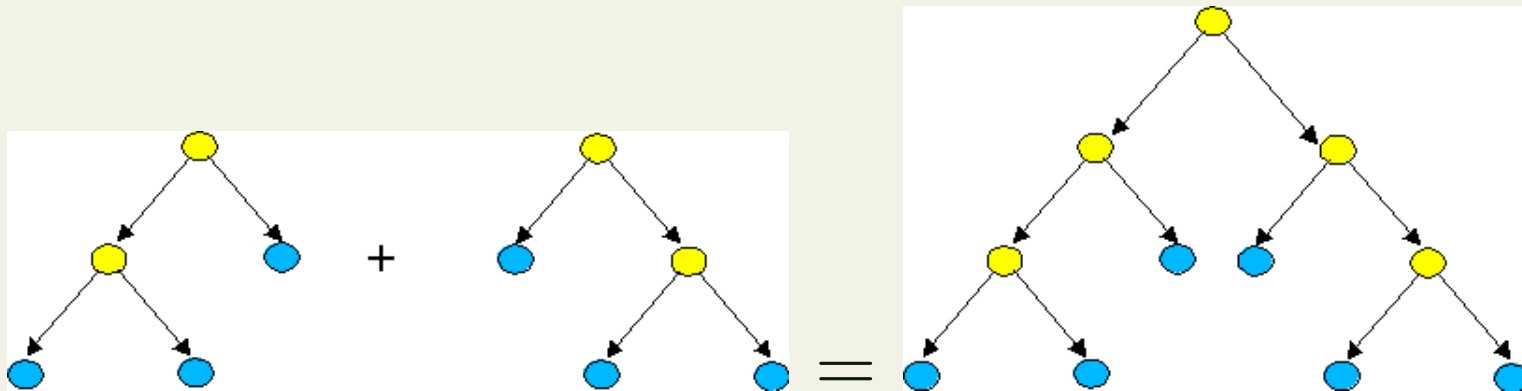
The *type definition* of $\alpha \text{ dtree}$ uses such an encoding.

Building Trees

- $Atom(n)$

●
n

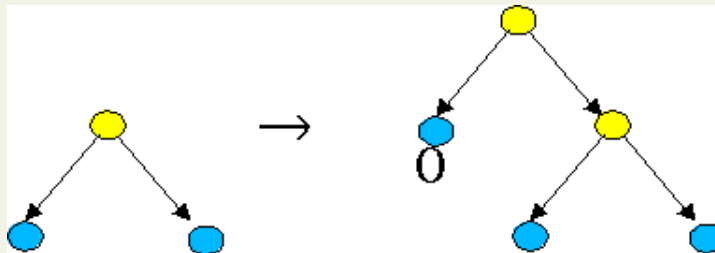
- $Scons\ X\ Y$



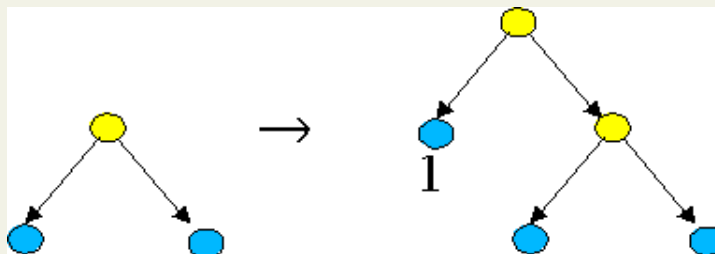
Tagging Trees

We want to **tag** an S-expression by either 0 or 1. This can be done by “*Scons*”-ing it with an S-expression consisting of an **administration label**. By convention, the tag is to the left.

- In0_def $In0(X) \equiv Scons\ Atom(Inr(0))(X)$



- In1_def $In1(X) \equiv Scons\ Atom(Inr(1))(X)$



Products and Sums on Sets of S-Expressions

Product of two sets A and B of S-expressions: All *Scons*-trees where left subtree from A , right subtree from B .

$$\text{uprod_def} \quad \text{uprod } A \ B \equiv \bigcup_{x \in A} \bigcup_{y \in B} \{(Scons \ x \ y)\}$$

Products and Sums on Sets of S-Expressions

Product of two sets A and B of S-expressions: All *Scons*-trees where left subtree from A , right subtree from B .

$$\text{uprod_def} \quad \text{uprod } A \ B \equiv \bigcup_{x \in A} \bigcup_{y \in B} \{(Scons \ x \ y)\}$$

Sum of two sets A and B of S-expressions: union of A and B after S-expressions in A have been tagged 0 and S-expressions in B have been tagged 1, so that one can tell where they come from.

$$\text{usum_def} \quad \text{usum } A \ B \equiv \text{In0} \ ' \ A \cup \text{In1} \ ' \ B$$

Some Properties of Trees and Tree Sets

- $Atom$, $In0$, $In1$, $Scons$ are injective.
- $Atom$ and $Scons$ are pairwise distinct. $In0$ are $In1$ pairwise distinct.

Some Properties of Trees and Tree Sets

- $Atom$, $In0$, $In1$, $Scons$ are injective.
- $Atom$ and $Scons$ are pairwise distinct. $In0$ are $In1$ pairwise distinct.
- Tree sets represent a universe that is closed under products and sums: $usum$, $uprod$ have type $[(\alpha \text{ dtree}) \text{ set}, (\alpha \text{ dtree}) \text{ set}] \Rightarrow (\alpha \text{ dtree}) \text{ set}$.
- $uprod$ and $usum$ are monotone.

Some Properties of Trees and Tree Sets

- $Atom$, $In0$, $In1$, $Scons$ are injective.
- $Atom$ and $Scons$ are pairwise distinct. $In0$ are $In1$ pairwise distinct.
- Tree sets represent a universe that is closed under products and sums: $usum$, $uprod$ have type $[(\alpha\ dtree)\ set, (\alpha\ dtree)\ set] \Rightarrow (\alpha\ dtree)\ set$.
- $uprod$ and $usum$ are monotone.
- Tree sets represent a universe that is closed under products and sums combined with arbitrary applications of lfp .

Reminder: we simplified!

Lists in Isabelle

Similar to the construction of *nat*, we first construct a **set** of S-expressions having the “structure of lists”. We start by defining “provisional” list constructors:

```
constdefs
```

```
  NIL :: 'a dtree
```

```
  "NIL == In0(Atom(Inr(0)))"
```

```
  CONS :: ['a dtree, 'a dtree] => 'a dtree
```

```
  "CONS M N == In1(Scons M N)"
```

What type **do you expect** *Cons* to have, and how does *CONS* compare?

Lists in Isabelle

Similar to the construction of *nat*, we first construct a **set** of S-expressions having the “structure of lists”. We start by defining “**provisional**” list constructors:

```
constdefs
```

```
  NIL :: 'a dtree
```

```
  "NIL == In0(Atom(Inr(0)))"
```

```
  CONS :: ['a dtree, 'a dtree] => 'a dtree
```

```
  "CONS M N == In1(Scons M N)"
```

What type **do you expect** *Cons* to have, and how does *CONS* compare? Must wrap **list elements** by $Atom \circ Inl$.

Lists as S-Expressions: Intuition

Examples of how lists would be represented as S-expressions:

Nil

[]

Cons(7, *Nil*)

[7]

Cons(5, *Cons*(7, *Nil*))

[5, 7]

Lists as S-Expressions: Intuition

Examples of how lists would be represented as S-expressions:

Nil []

In0(Atom(Inr 0))

Cons(7, Nil) [7]

Cons(5, Cons(7, Nil)) [5, 7]

Lists as S-Expressions: Intuition

Examples of how lists would be represented as S-expressions:

<i>Nil</i>	$[]$
$In0(Atom(Inr\ 0))$	
$Cons(7, Nil)$	$[7]$
$CONS\ (Atom(Inl\ 7))\ In0(Atom(Inr\ 0))$	
$Cons(5, Cons(7, Nil))$	$[5, 7]$

Lists as S-Expressions: Intuition

Examples of how lists would be represented as S-expressions:

<i>Nil</i>	<code>[]</code>
$\frac{In0(Atom(Inr\ 0))}{Cons(7, Nil)}$	<code>[7]</code>
$\frac{CONS\ (Atom(Inl\ 7))\ In0(Atom(Inr\ 0))}{Cons(5, Cons(7, Nil))}$	<code>[5, 7]</code>
$CONS\ (Atom(Inl\ 5))$ $(CONS\ (Atom(Inl\ 7))\ In0(Atom(Inr\ 0)))$	

Now let's construct the S-expressions having this form.

Lists as S-Expressions: Inductive Construction

Idea: let $A :: (\alpha \text{ dtree}) \text{ set}$ be the set of all “wrapped” elements, e.g. for $\alpha = \text{nat}$, the set $\{(Atom\ Inl\ 0), (Atom\ Inl\ 1), \dots\}$. Then define $list(A)$, the set of S-expressions that represent lists of element type α :

```
list      :: "'a dtree set => 'a dtree set"
inductive "list(A)"
  intrs
    NIL_I   "NIL : list(A)"
    CONS_I  "[| a : A; M : list(A) |] ==>
              CONS a M : list(A)"
```

See [SList.thy](#) for how it's really done!

Defining the “Real” List Type

We now apply the `type definition mechanism` using the `typedef` syntax. How do we define A formally?

Defining the “Real” List Type

We now apply the **type definition mechanism** using the **typedef** syntax. How do we define A formally?

```
typedef (List)
  'a list =
    "list(range (Atom o Inl)) :: 'a dtree set"
by ...
```

Choosing A as $\text{range } (\text{Atom} \circ \text{Inl})$ together with the explicit type declaration forces A to be the set containing all $\text{Atom } (\text{Inl } t)$, for each $t :: \alpha$.

Example of a definition of a **polymorphic** type.

List Constructors

We define the real constructor names for lists:

```
Nil_def    "Nil::'a list == Abs_list(NIL)"  
Cons_def   "x#(xs::'a list) ==  
            Abs_list(CONS (Atom(Inl(x))) (Rep_list xs))"
```

We then forget about *NIL* and *CONS*.

Isabelle's Datatype Package

Similar to the `typedef syntax`, Isabelle provides the datatype syntax to support the `construction` of a datatype:

```
datatype 'a list = Nil | Cons 'a ('a list)
```

In particular, this automates the proofs of:

- the induction theorem;
- distinctness;
- injectivity of constructors.

Isabelle's Datatype Package

Similar to the `typedef syntax`, Isabelle provides the datatype syntax to support the `construction` of a datatype:

```
datatype 'a list = Nil | Cons 'a ('a list)
```

In particular, this automates the proofs of:

- the induction theorem;
- distinctness;
- injectivity of constructors.

The package also works for mutually and indirectly recursive datatype definitions.

Question: `Why didn't we use this package to define nat?` ►

More Detailed Explanations

What Is a Datatype?

We have seen types, but what are **data**types?

First of all, a datatype must be of **order 0**, so it must be a non-functional type. Note that if we do not have polymorphism, this means that a datatype must be a in \mathcal{B} . But if we have polymorphism, it just means that the type must not contain \rightarrow . E.g., $\alpha \text{ list}$ could be a datatype.

However, when one describes a datatype, one would usually speak about generic instances such as $\alpha \text{ list}$, and not about, say, nat list .

Secondly, the terms that inhabit a datatype τ must be defined using a finite set of **term constructors** that have τ as result type. At least one term constructor should just have type τ . E.g., $\text{Nil} : \alpha \text{ list}$ and

$\text{Cons} : \alpha \rightarrow (\alpha \text{ list}) \rightarrow \alpha \text{ list}$ are the term constructors that define the list datatype. One also finds a syntax where Nil is written $[]$ and Cons is written $::$. Intuitively, we could say: the terms of a datatype are exactly

the terms that can be constructed by some finite syntactic construction rule.

Whenever we have a term constructor that has τ as argument as well as result, the construction rule is **inductive**. E.g., we have

- Nil is a list;
- if t is a list h is of type α , then $Cons(h, t)$ is a list.

This is an inductive construction of lists. Usually, when one speaks about datatypes, one has inductively defined ones in mind. Examples are lists, **natural numbers**, trees. One could say that e.g. *bool* is also a datatype defined by the constants *True* and *False*, but it is not particularly interesting in this context.

At the same time, each term constructed by such a syntactic rule is **unique**. So if we say: lists are defined by the above inductive construction, then we imply that $Cons(1, Nil)$ must **not** be equal to

$Cons(1, Cons(1, Nil))$.

To understand better the distinction of a **datatype** from another type, consider the following **counterexample**: $\alpha \text{ set}$. Sets are not a datatype:

1. While the type $\alpha \text{ set}$ does not contain an \rightarrow , it is **isomorphic** to $\alpha \rightarrow \text{bool}$ which does contain an \rightarrow .
2. The most basic way of defining “what a set is” is: if f is of type $\tau \rightarrow \text{bool}$, then $Abs_{\text{set}} f$ (alternatively: $Collect f$) is a set. This is not an inductive syntactic construction rule.
3. One could define sets similarly to lists by an inductive rule saying: $\{\}$ is a set; if S is a set and h is some term of type α , then $Insert(h, S)$ is a set. But then $Insert(1, \{\})$ would be **different** from $Insert(1, Insert(1, \{\}))$, which is not what we want! Moreover, we could not define **infinite** sets this way.
4. In point 2 we say: the definition of the terms called “sets” is not an

inductive definition. This is not in contradiction to the **inductive definition** of particular sets. These inductive definitions have the form: If *foo* is **in** the set then *bar* is **in** the set, e.g., if n is in the set then $Suc\ n$ is in the set. This is in contrast to what is suggested in point 3, where we say: If *foo* **is a** set then *bar* **is a** set.

Gödelization

In computation theory, Gödelization is the process of encoding words over arbitrary alphabets, or any arbitrary datastructures, for that matter, as natural numbers.

As an example, how could we encode a **list** of natural numbers as a single natural number. Prime factors are one technique for that.

Suppose the number n has a factorization $2^i \cdot 3^j \cdot 5^k \cdot 7^l$. Then it would encode the list $[i, j, k, l]$.

But such an encoding for datatypes would be very cumbersome. In particular, there is no way of realizing **polymorphism**.

Datatype_Universe.thy

This file should be contained in your Isabelle distribution. In Freiburg it is

`/usr/local/isabelle/Isabelle/src/HOL/Datatype_Universe.thy`

S-Expressions Explained

The datastructure we have in mind here consists of **binary trees** where the inner nodes are not labeled, and the leaves are labeled

- **either** with a term of arbitrary type, in which case the leaf would be an actual “piece of content” in the datastructure,
- **or** with a natural number, in which case the leaf serves special purposes for organizing our datastructure, as we will see later.

I.e., such binary trees have a type parametrized by a type variable α , the type of the latter kind of leaves. Let us call the type of such trees α *dtree*.

As always with **parametric polymorphism**, when we consider how the datastructure as such works, we are not interested in what the values in the former kind of leaves are. This is just like the type and values of list elements are irrelevant for **concatenating** two lists. Of course, α could,

by coincidence, be instantiated to type nat .

Think of a label of the first kind as **content label** and a label of the second kind as **administration label**.

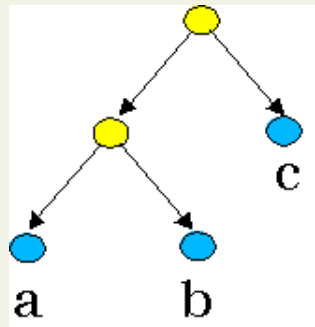
Technically, if something is **either** of this type or of that type, we are talking about a **sum type**. So a **leaf label** has type $\alpha + nat$ (written (α, nat) *sum* before), and it has the form either $Inl(a)$ for some $a :: \alpha$, or $Inr(n)$ for some $n :: nat$.

Path Sets Explained

The set

$$\{(\langle 0, 0 \rangle, a), (\langle 0, 1 \rangle, b), (\langle 1 \rangle, c)\}$$

represents the tree



The path $\langle 0, 0 \rangle$ means: from the root take left subtree, then again left subtree. The path $\langle 1 \rangle$ means: take right subtree.

How can a path $\langle p_0, \dots, p_n \rangle$ be **represented**? One idea is to use the

function $f :: nat \Rightarrow nat$ defined by

$$f\ i = \begin{cases} p_i & \text{if } i \leq n \\ 2 & \text{otherwise} \end{cases}$$

as representation of $\langle p_0, \dots, p_n \rangle$.

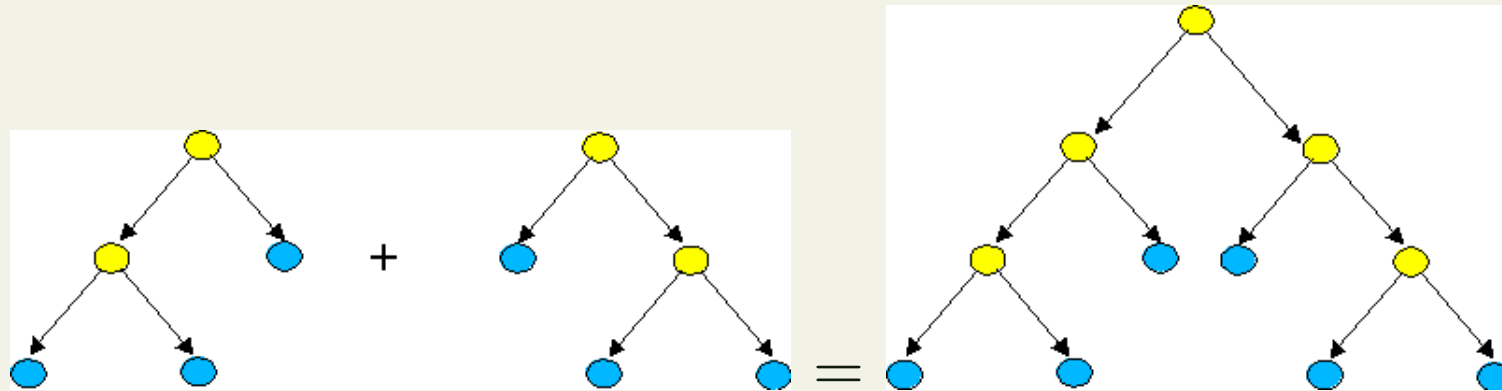
Atom

Atom takes a **leaf label** and turns it into a (simplest possible) **S-expression** (tree).

So it has type $\alpha + \text{nat} \Rightarrow \alpha \text{ dtree}$.

Scons

Scons takes two S-expressions and creates a new S-expression as illustrated below:



So it has type $[\alpha \text{ dtree}, \alpha \text{ dtree}] \Rightarrow \alpha \text{ dtree}$.

$$In0 \text{ ' } \dots, In1 \text{ ' } \dots$$

Recall that ' denotes the **image** of a function applied to a set.

Injective and Pairwise Distinct Functions

This means that any of $Atom$, $In0$, $In1$, $Scons$ applied to **different** S-expressions will return **different** S-expressions.

Moreover, a term with root $Scons$ is definitely different from a term with root $Atom$, and a term with root $In0$ is definitely different from a term with root $In1$.

Why is this important? It is an inherent characteristic of a **datatype**. A datatype consists of terms constructed using **term constructors** and is uniquely defined by what it is syntactically (one also says that terms are generated **freely** using the constructors). **For example**, injectivity of Suc and pairwise-distinctness of 0 and Suc mean for any two numbers m and n , the terms $\underbrace{Suc(\dots Suc(0) \dots)}_{m \text{ times}}$ and $\underbrace{Suc(\dots Suc(0) \dots)}_{n \text{ times}}$ are different.

Computing the Closure

Given a set T of trees (S-expressions), the **closure of T under $Atom$, $In0$, $In1$, $Scons$, $usum$, $uprod$** is the smallest set T' such that $T \subseteq T'$ and given any tree (or two trees, as applicable) from T' , any tree constructable using $Atom$, $In0$, $In1$, $Scons$, $usum$, $uprod$ is also contained in T' .

Remembering the construction of inductively defined sets, the closure is the **least fixpoint** of a monotone function adding trees to a tree set. This function must be constructed using $Atom$, $In0$, $In1$, $Scons$, $usum$, $uprod$. We do not go into the details, but note that it is crucial that $uprod$ and $usum$ are **monotone**, and note as well that slight complications arise from the fact that $usum$ and $uprod$ have type $[(\alpha \text{ dtree}) \text{ set}, (\alpha \text{ dtree}) \text{ set}] \Rightarrow (\alpha \text{ dtree}) \text{ set}$ rather than $(\alpha \text{ dtree}) \text{ set} \Rightarrow (\alpha \text{ dtree}) \text{ set}$.

The Expected Type of *Cons*

Cons should have the polymorphic type $[\alpha, \alpha \text{ list}] \Rightarrow \alpha \text{ list}$. The important point is: the first argument is of different type than the second argument. If the first is of type τ , then the second must be of type $\tau \text{ list}$. In contrast, *CONS* is of type $[(\alpha \text{ dtree}), (\alpha \text{ dtree})] \Rightarrow \alpha \text{ dtree}$. In order to apply *CONS* to a “list” (in fact an S-expression) and a “list element”, we must first wrap the list element by $\text{Atom} \circ \text{Inl}$, so that it becomes an S-expression.

List Syntaxes

Nil, *Cons*(7, *Nil*), *Cons*(5, *Cons*(7, *Nil*)) are lists written according to what some programming languages introduce as the first, “official” syntax for lists.

For convenience, programming languages typically allow for the same lists to be written as [], [7], [5, 7].

SList.thy

This file should be contained in your Isabelle distribution. In Freiburg it is

`/usr/local/isabelle/Isabelle/src/HOL/Induct/SList.thy`

Why Didn't We Use the Datatype Package to Define *nat*?

The datatype syntax is very convenient since the complex construction we have seen today is transparent to the normal user.

In particular, proofs of the induction theorem are automated. This is in contrast to the construction of *nat* where this theorem **was not generated automatically**.

So why didn't we use the datatype syntax to define *nat*, since it is so much more convenient?

The reason is that we **needed** *nat* to define S-expressions, so the type *nat* must exist before there can be a datatype package, and so the datatype package cannot be used to define *nat*.

Summary of HOL Library / Outlook on Modeled Systems

Summary

In the previous weeks, we looked at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#):

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Summary (Cont.)

We conclude: HOL is a logical framework for theoretical computer science. Its features are:

- a clean methodology, which can be supported automatically to a surprising extent;
- a powerful set theory and proof support;
- adequate theories for arithmetics (proof-support: not quite satisfactory so far);
- a package for induction;
- a package for recursion;
- a package for datatypes.

Outline

We will now look at how various formalisms (specification and programming languages) can be **embedded** in HOL:

- Z and data-refinement
- Imperative languages
- Denotational semantics and functional languages
- Object-oriented languages (Java-Light . . .)

IMP

IMP: Introduction

Semantics come in different **flavors**.

Representing imperative languages in HOL:

- **operational** semantics, **denotational** semantics,
- . . . deriving **axiomatic** semantics (Hoare-logic).

IMP: Introduction

Semantics come in different **flavors**.

Representing imperative languages in HOL:

- **operational** semantics, **denotational** semantics,
- . . . deriving **axiomatic** semantics (Hoare-logic).

(In contrast, in MiniSML we would have:

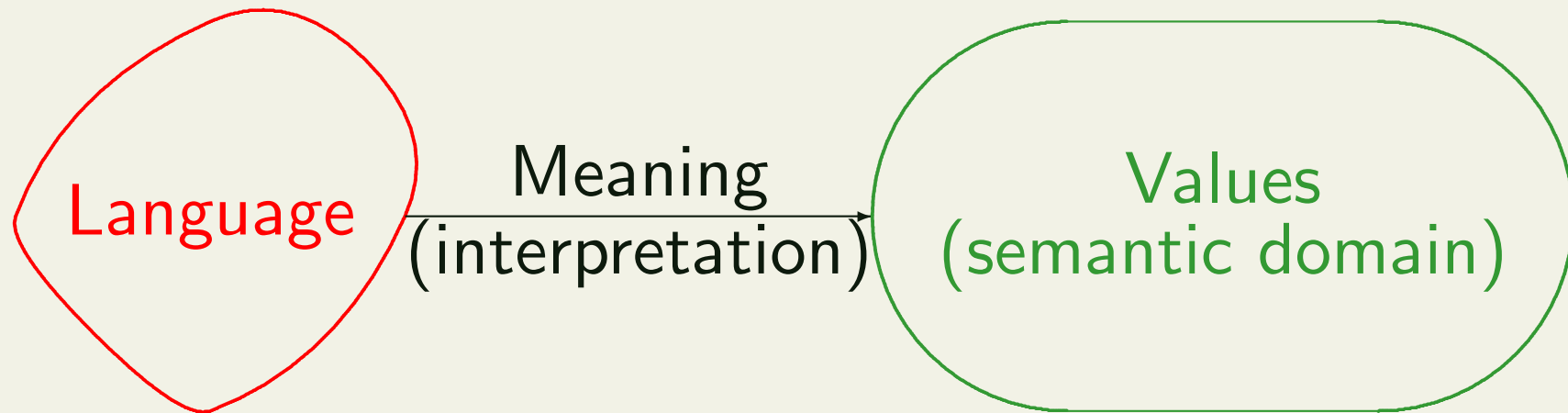
- **denotational** semantics
- . . . deriving **operational** semantics).

Meaning of Imperative Languages

Question: What is the **meaning** of an imperative language?

Meaning of Imperative Languages

Question: What is the **meaning** of an imperative language?



- Syntax: language = set of symbols
- Semantics: set of denotations, the **semantic domain**
- **Meaning** is a function (interpretation) relating these two.

Imperative Languages in the Isabelle/HOL Library

There are several embeddings of imperative languages in Isabelle/HOL [Nip02]:

- Hoare
- IMP
- IMPP
- MicroJava

Imperative Languages in the Isabelle/HOL Library

There are several embeddings of imperative languages in Isabelle/HOL [Nip02]:

- Hoare: shallowish, good examples
- IMP
- IMPP
- MicroJava

Imperative Languages in the Isabelle/HOL Library

There are several embeddings of imperative languages in Isabelle/HOL [Nip02]:

- Hoare: shallowish, good examples
- IMP: deepish, good theory
- IMPP
- MicroJava

Imperative Languages in the Isabelle/HOL Library

There are several embeddings of imperative languages in Isabelle/HOL [Nip02]:

- Hoare: shallowish, good examples
- IMP: deepish, good theory
- IMPP: extends IMP with procedures
- MicroJava

Imperative Languages in the Isabelle/HOL Library

There are several embeddings of imperative languages in Isabelle/HOL [Nip02]:

- **Hoare:** shallowish, good examples
- **IMP:** deepish, good theory
- **IMPP:** extends IMP with procedures
- **MicroJava:** complex, powerful, state-of-the-art

Imperative Languages in the Isabelle/HOL Library

There are several embeddings of imperative languages in Isabelle/HOL [Nip02]:

- **Hoare**: shallowish, good examples
- **IMP**: deepish, good theory
- **IMPP**: extends IMP with procedures
- **MicroJava**: complex, powerful, state-of-the-art

We choose **IMP** to learn a bit about “good ole imperative languages”.

Semantics Provided for IMP

IMP offers:

- operational semantics;
 - natural semantics;
 - transition semantics;
- denotational semantics;
- axiomatic semantics (Hoare logic);

Semantics Provided for IMP

IMP offers:

- operational semantics;
 - natural semantics;
 - transition semantics;
- denotational semantics;
- axiomatic semantics (**Hoare logic**);
- equivalence proofs;
- weakest preconditions and verification condition generator.

It closely follows the standard textbook [Win96].

An Imperative Language Embedding

We will now **define** the syntax and various semantics of IMP, but in fact, we define those as Isabelle theories. We say that we **embed** IMP in Isabelle/HOL.

You will see that such an embedding is more abstract and less detailed as if we were really going to define IMP for use as a programming language, i.e., if we were going to define a compiler for it.

The Command Language (Syntax)

The (abstract) syntax is defined in `Com.thy`.

Com = Main +

types

loc

val = nat (*or whatever*)

state = loc \Rightarrow **val**

aexp = state \Rightarrow **val**

bexp = state \Rightarrow bool

datatype com =

SKIP

| " := " loc aexp (infixl 60)

| Semi com com (" ; " [60, 60] 10)

| Cond bexp com com

(" IF _ THEN _ ELSE _ " 60)

| While bexp com ("WHILE _ DO _ " 60)

The type *loc* stands for `locations`.

Note the `abstractness` of *aexp* and *bexp*.

The Command Language (Syntax)

The (abstract) syntax is defined in `Com.thy`.

Com = Main +

types

loc

val = nat (*or whatever*)

state = loc \Rightarrow **val**

aexp = state \Rightarrow **val**

bexp = state \Rightarrow bool

datatype com =

SKIP

| " := " loc aexp (infixl 60)

| Semi com com (" ; " [60, 60] 10)

| Cond bexp com com

(" IF _ THEN _ ELSE _ " 60)

| While bexp com ("WHILE _ DO _ " 60)

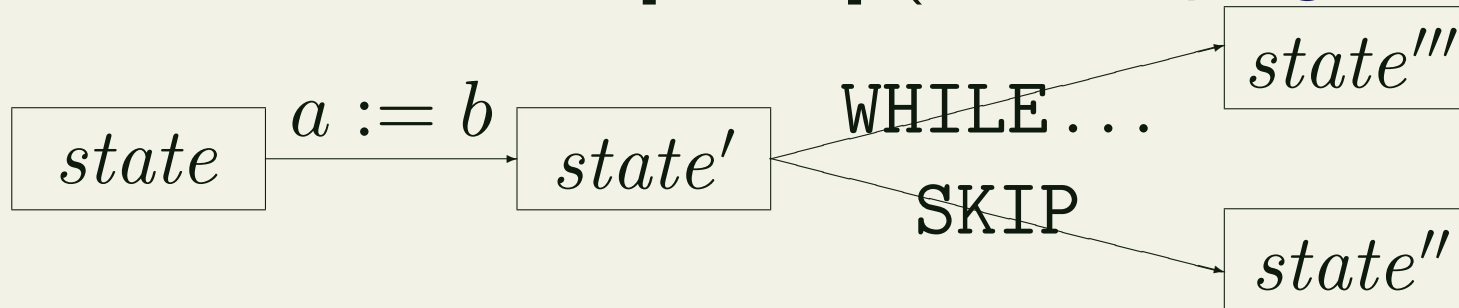
The type *loc* stands for `locations`.

Note the `abstractness` of *aexp* and *bexp*.

The datatype *com* stands for commands (command sequences).

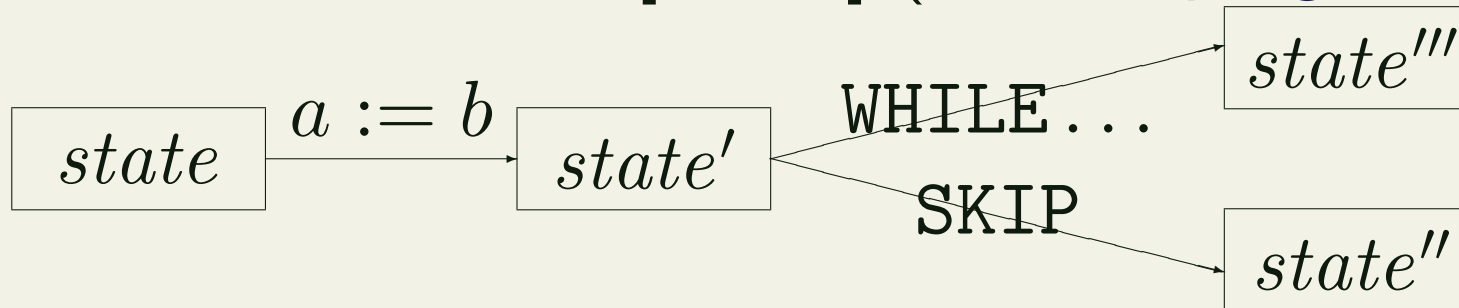
Operational Semantics: Two Kinds

Natural semantics [Plo81] (idea: a program relates states):



Operational Semantics: Two Kinds

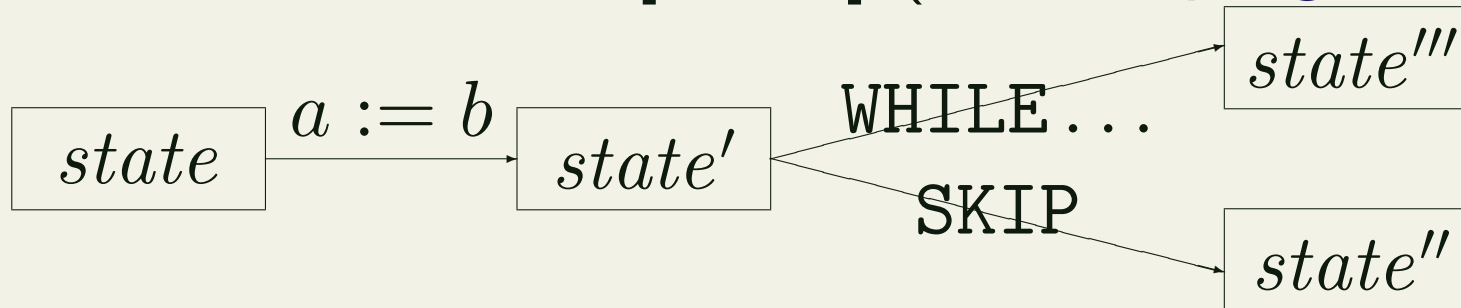
Natural semantics [Plo81] (idea: a program relates states):



$\text{evalc} :: (\text{com} * \text{state} * \text{state}) \text{ set}$

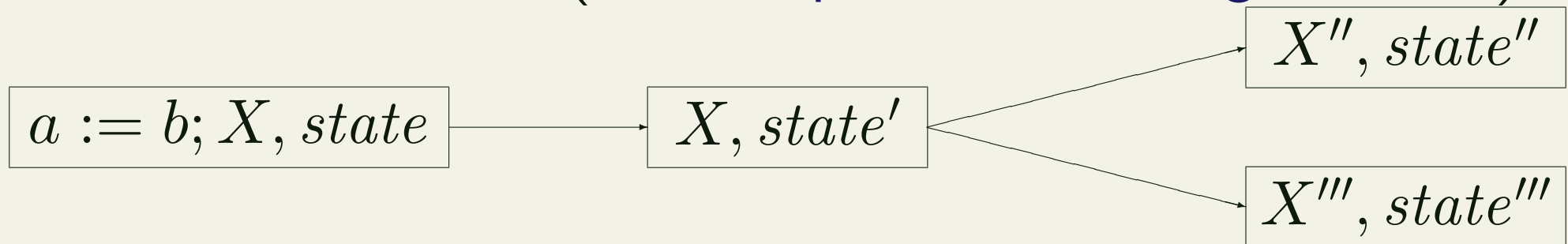
Operational Semantics: Two Kinds

Natural semantics [Plo81] (idea: a program relates states):



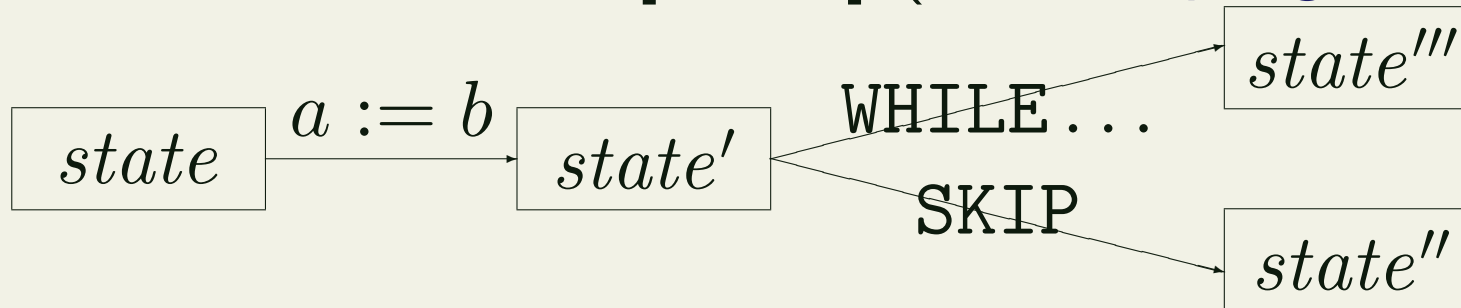
$evalc :: (com * state * state) \text{ set}$

Transition semantics (idea: sequence of “configurations”):



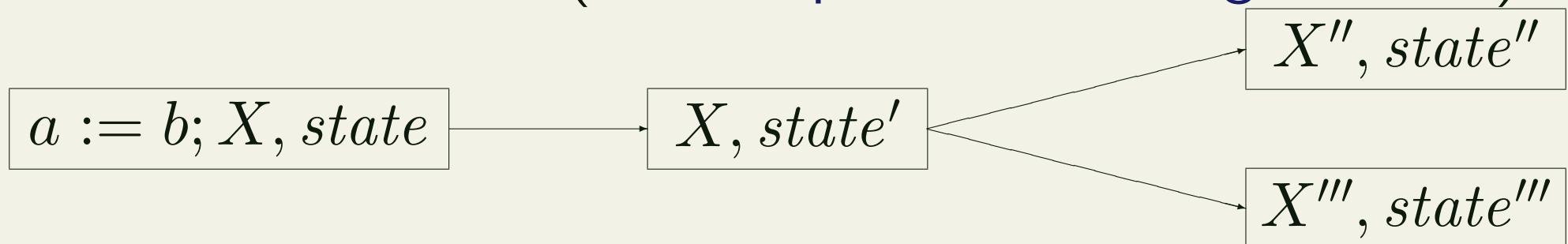
Operational Semantics: Two Kinds

Natural semantics [Plo81] (idea: a program relates states):



$evalc :: (com * state * state) \text{ set}$

Transition semantics (idea: sequence of “configurations”):



$evalc1 :: ((com * state) * (com * state)) \text{ set}$

Embedding of the Natural Semantics

The natural semantics encoding in Isabelle is given by an inductive definition. We first declare its type and define a paraphrasing using an arrow symbol for readability:

Embedding of the Natural Semantics

The natural semantics encoding in Isabelle is given by an inductive definition. We first declare its type and define a paraphrasing using an arrow symbol for readability:

```
consts evalc  :: "(com * state * state) set"
translations "<c,s> -c→ s'" ≡ "(c,s,s') : evalc"
```

Note that $-c\rightarrow$ is one **fixed** arrow symbol.

Embedding of the Natural Semantics

The natural semantics encoding in Isabelle is given by an inductive definition. We first declare its type and define a paraphrasing using an arrow symbol for readability:

```
consts evalc  :: "(com * state * state) set"
translations "<c,s> -c→ s'" ≡ "(c,s,s') : evalc"
```

Note that $-c \rightarrow$ is one **fixed** arrow symbol.

We now start giving the actual inductive definition. It defines the $-c \rightarrow$ transitions (implicit: these are the **only** $-c \rightarrow$ transitions) . . .

The Inductive Definition: Skip and Assignment

inductive evalc

intrs

Skip: $\langle \text{SKIP}, s \rangle \rightarrow s$

Assign: $\langle x ::= a, s \rangle \rightarrow s[x ::= (a \ s)]$

The Inductive Definition: Skip and Assignment

inductive evalc

intrs

Skip: $\langle \text{SKIP}, s \rangle \rightarrow c \rightarrow s$

Assign: $\langle x ::= a, s \rangle \rightarrow c \rightarrow s[x ::= (a \ s)]$

Skip and Assign are just names for the clauses of the inductive definition.

$s[x ::= v]$ is short for *update s x v*, where

$$\text{update } s \ x \ v \equiv \lambda y. \text{ if } y = x \text{ then } v \text{ else } (s \ y)$$

Note that a is of type *aexp* or *bexp*.

The Inductive Definition: Semicolon

$$\begin{aligned} \text{Semi: } & \llbracket \langle c_0, s \rangle \multimap c \rightarrow s_1; \langle c_1, s_1 \rangle \multimap c \rightarrow s_2 \rrbracket \\ & \implies \langle c_0; c_1, s \rangle \multimap c \rightarrow s_2 \end{aligned}$$

The Inductive Definition: Semicolon

$$\begin{aligned} \text{Semi: } & \llbracket \langle c_0, s \rangle \rightarrow c \rightarrow s_1; \langle c_1, s_1 \rangle \rightarrow c \rightarrow s_2 \rrbracket \\ & \implies \langle c_0; c_1, s \rangle \rightarrow c \rightarrow s_2 \end{aligned}$$

The rationale of natural semantics: To figure out the meaning of a program consisting of a “first instruction” c_0 and a “rest” c_1 , starting from state s , you have to show two **subgoals**: c_0 starting from state s goes to some state s_1 , and c_1 starting in state s_1 goes to some state s_2 .

Note that by the definition of **Semi**, c_0 does not have to be “atomic” (whatever this means).

The Inductive Definition: Control

IfTrue : $\llbracket b \ s; \langle c_0, s \rangle \rightarrow c \rightarrow s_1 \rrbracket$

$\implies \langle \text{IF } b \text{ THEN } c_0 \text{ ELSE } c_1, s \rangle \rightarrow c \rightarrow s_1$

IfFalse : $\llbracket \neg b \ s; \langle c_1, s \rangle \rightarrow c \rightarrow s_1 \rrbracket$

$\implies \langle \text{IF } b \text{ THEN } c_0 \text{ ELSE } c_1, s \rangle \rightarrow c \rightarrow s_1$

WhileFalse : $\llbracket \neg b \ s \rrbracket \implies \langle \text{WHILE } b \text{ DO } c, s \rangle \rightarrow c \rightarrow s$

WhileTrue : $\llbracket b \ s; \langle c, s \rangle \rightarrow c \rightarrow s_1; \langle \text{WHILE } b \text{ DO } c, s_1 \rangle \rightarrow c \rightarrow s_2 \rrbracket$
 $\implies \langle \text{WHILE } b \text{ DO } c, s \rangle \rightarrow c \rightarrow s_2$

Note the termination problem in *WhileTrue*! Simplest example: $b \equiv \lambda x. \text{True}$. Then, no proof is possible and no s_2 can effectively be computed.

Embedding of the Transition Semantics

The transition semantics encoding in Isabelle is given by an inductive definition. We first declare its type and define a paraphrasing, *as before*:

Embedding of the Transition Semantics

The transition semantics encoding in Isabelle is given by an inductive definition. We first declare its type and define a paraphrasing, *as before*:

```
consts evalc1  :: "((com * state) * (com * state)) set"  
translations  "cs0  $\text{--}1\text{--}\rightarrow$  cs1"  $\equiv$  "(cs0, cs1) : evalc1"
```

Note that $\text{--}1\text{--}\rightarrow$ is one *fixed* arrow symbol.

Embedding of the Transition Semantics

The transition semantics encoding in Isabelle is given by an inductive definition. We first declare its type and define a paraphrasing, *as before*:

```
consts evalc1  :: "((com * state) * (com * state)) set"  
translations  "cs0  $\text{--}1\text{--}\rightarrow$  cs1"  $\equiv$  "(cs0, cs1) : evalc1"
```

Note that $\text{--}1\text{--}\rightarrow$ is one *fixed* arrow symbol.

We now start giving the actual inductive definition . . .

The Inductive Definition

inductive evalc1

intrs

Assign: " $(x ::= a, s) \rightarrow (SKIP, s[x ::= a \ s])$ "

Semi1: " $(SKIP; c, s) \rightarrow (c, s)$ "

Semi2: " $(c_0, s) \rightarrow (c_2, s_1) \implies (c_0; c_1, s) \rightarrow (c_2; c_1, s_1)$ "

The Inductive Definition

inductive evalc1

intrs

Assign: " $(x ::= a, s) \rightarrow (SKIP, s[x ::= a \ s])$ "

Semi1: " $(SKIP; c, s) \rightarrow (c, s)$ "

Semi2: " $(c_0, s) \rightarrow (c_2, s_1) \implies (c_0; c_1, s) \rightarrow (c_2; c_1, s_1)$ "

So far, we see that the component of *com* type in the configuration corresponds to a program stack (built by ";"), which represents a **program counter**.

The Inductive Definition: Control

IfTrue : " $b \ s \Longrightarrow (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2, s) \rightarrow (c_1, s)$ "

IfFalse : " $\neg b \ s \Longrightarrow (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2, s) \rightarrow (c_2, s)$ "

WhileFalse : " $\neg b \ s \Longrightarrow (\text{WHILE } b \ \text{DO } c, s) \rightarrow (\text{SKIP}, s)$ "

WhileTrue : " $b \ s \Longrightarrow (\text{WHILE } b \ \text{DO } c, s) \rightarrow (c; \text{WHILE } b \ \text{DO } c, s)$ "

Termination problem **as before**, but somehow less disturbing:
we cannot be shocked about the fact that some
computations are infinite, and at least, the transition
semantics assigns a meaning to any **finite prefix** of an infinite
computation.

Generalizations to more than one Step

n -step semantics:

$$"cs_0 \multimap^n cs_1" \equiv "(cs_0, cs_1) : evalc1^n"$$

Unlike $\multimap c$ and $\multimap 1$, $\multimap n$ is not a fixed arrow symbol, but **meta-notation**: for any number n , there is the **paraphrasing** $\multimap n$ defined as above. Here, $evalc1^n$ is defined in `Relation_Power.thy`.

multistep-semantics:

$$"cs_0 \multimap^* (c_1, s_1)" \equiv "(cs_0, c_1, s_1) : evalc1^*"$$

\multimap^* is a fixed arrow symbol.

Equivalence of Semantics

Natural semantics vs. transition semantics.

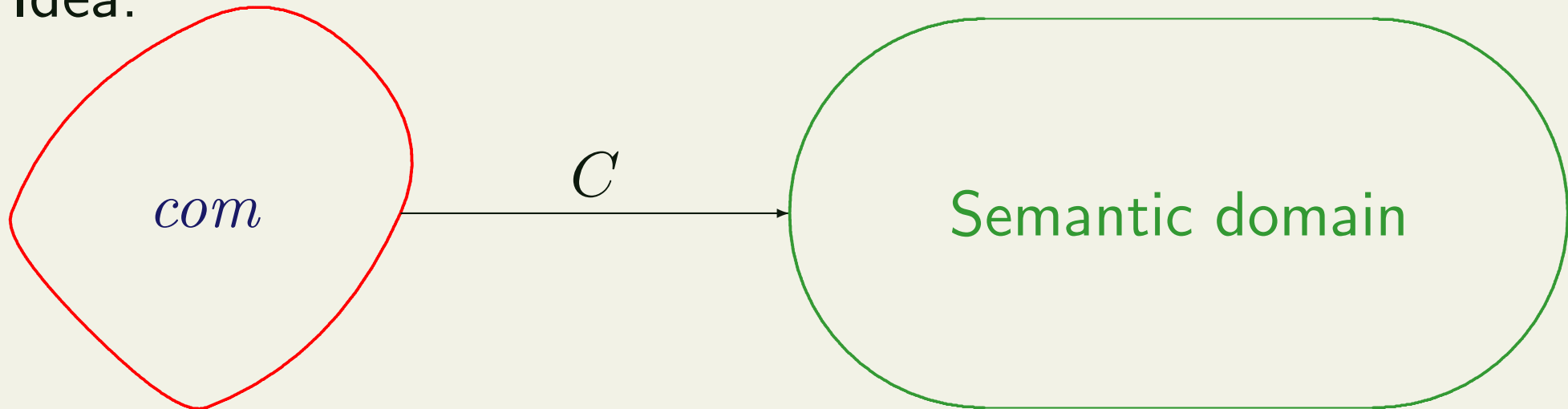
Theorem 6 (`evalc1_eq_evalc`):

$$(c, s) \text{ --* } \rightarrow (SKIP, t) = (\langle c, s \rangle \text{ --} c \rightarrow t)$$

The proof is by induction on the structure of programs.

Embedding of the Denotational Semantics

Idea:



Domain: A semantics relates states (similar to *natural* semantics)

$\text{com_den} = (\text{state} * \text{state}) \text{ set}$

Semantic function: assigns semantics to a program

$\text{consts } C :: \text{com} \Rightarrow \text{com_den}$

Characteristics of Denotational Semantics

A denotational semantics is a **function** (here: C) assigning a meaning to a program. More precisely, the meaning of a program is some “mathematical” function of the meanings of its components.

This is in contrast to the operational view where **computation order** (“first do this, then that. . .”) and logical reasoning using **proof rules** (“**if** (. . .) computes (. . .) **then** (. . .) computes (. . .)”) are focused.

The “mathematics” uses the *lfp* operator.

The Inductive Definition

The semantics C is defined **inductively**:

primrec

C_skip " $C(SKIP) = Id$ "

C_assign " $C(x ::= a) = \{(s, t). t = s[x ::= a(s)]\}$ "

C_comp " $C(c_0 ; c_1) = C(c_1) \circ C(c_0)$ "

C_if " $C(IF\ b\ THEN\ c_1\ ELSE\ c_2) =$
 $\{(s, t). (s, t) : C(c_1) \wedge b(s)\} \cup$
 $\{(s, t). (s, t) : C(c_2) \wedge \neg b(s)\}$ "

C_while " $C(WHILE\ b\ DO\ c) = lfp\ (\Gamma\ b\ (C\ c))$ "

where

" $\Gamma\ b\ cd \equiv (\lambda\phi. \{(s, t). (s, t) : (\phi \circ cd) \wedge b(s)\} \cup$
 $\{(s, t). s=t \wedge \neg b(s)\})$ "

Equivalence of Programs

We have seen an equivalence result relating different semantics.

Equivalence of Programs

We have seen an equivalence result relating **different semantics**.

The following is an equivalence relating program fragments.

Theorem 7 (C_While_If):

$$\begin{aligned} C \text{ (WHILE } b \text{ DO } c) &= \\ C \text{ (IF } b \text{ THEN } c; \text{ WHILE } b \text{ DO } c \text{ ELSE SKIP)} \end{aligned}$$

Such a result is very important because it justifies a **program transformation** (the two fragments have the same semantics and so they are interchangeable).

Equivalence of Semantics

We have already suggested that the natural semantics is a **hybrid** between **operational** and **denotational** semantics. In fact, there is a simple equivalence relationship between the two:

Theorem 8 (denotational is natural):

$$((s, t) : C c) = (\langle c, s \rangle \rightarrow c \rightarrow t)$$

Axiomatic (Hoare) Semantics

Idea: we relate “legal states” before and after a program execution. A set of legal states is modeled as “assertion”:

types $\text{assn} = \text{state} \Rightarrow \text{bool}$

Axiomatic (Hoare) Semantics

Idea: we relate “legal states” before and after a program execution. A set of legal states is modeled as “assertion”:

types $\text{assn} = \text{state} \Rightarrow \text{bool}$

So rather than reasoning about single **states**, we reason about **properties** or **sets** of states. This is what we really need for verification of programs.

Semantics called **axiomatic** for **historic reasons**. It is also called **Hoare** semantics.

Embedding of the Hoare Semantics

The Hoare semantics encoding in Isabelle is given by an inductive definition. We first declare its type and a paraphrasing:

```
consts hoare :: "(assn * com * assn) set"
translations " $\vdash \{P\}c\{Q\}$ "  $\equiv$  "(P,c,Q) : hoare"
```

An object of the form $\{P\}c\{Q\}$ is called a **Hoare-triple**.

We now start giving the actual inductive definition . . .

The Inductive Definition: SKIP

inductive hoare

intrs

skip " $\vdash \{P\} \text{ SKIP } \{P\}$ "

No surprise here.

The Inductive Definition

ass " $\vdash \{ \lambda s. P(s[x::=(a \ s)]) \} \ x:=a \ \{P\}$ "

This may be counter-intuitive, why not the other way round?

The Inductive Definition

ass " $\vdash \{ \lambda s. P(s[x ::= (a \ s)]) \} \ x ::= a \ \{ P \}$ "

This may be counter-intuitive, why not the other way round?

Consider an example: $a \equiv \lambda s. 1$ and $P \equiv \lambda s. s \ x = 1$

$$\begin{aligned}
 & \{ \lambda s. (\lambda s. s \ x = 1)(s[x ::= 1]) \} x ::= \lambda s. 1 \{ \lambda s. s \ x = 1 \} \longrightarrow_{\beta} \\
 & \{ \lambda s. \underline{(s[x ::= 1])} \ x = 1 \} x ::= \lambda s. 1 \{ \lambda s. s \ x = 1 \} \longrightarrow_{\beta} \\
 & \{ \lambda s. (1 = 1) \} ::= \lambda s. 1 \{ \lambda s. s \ x = 1 \} \longrightarrow_{\beta} \\
 & \{ \lambda s. True \} x ::= \lambda s. 1 \{ \lambda s. s \ x = 1 \}
 \end{aligned}$$

What do we see? (You might also [check the types.](#))

The Inductive Definition

ass " $\vdash \{\lambda s. P(s[x ::= (a \ s)])\} \ x ::= a \ \{P\}$ "

This may be counter-intuitive, why not the other way round?

Consider an example: $a \equiv \lambda s. 1$ and $P \equiv \lambda s. s \ x = 1$

$$\begin{aligned}
 & \{\lambda s. (\lambda s. s \ x = 1)(s[x ::= 1])\} x ::= \lambda s. 1 \{\lambda s. s \ x = 1\} \longrightarrow_{\beta} \\
 & \{\lambda s. \underline{(s[x ::= 1])} \ x = 1\} x ::= \lambda s. 1 \{\lambda s. s \ x = 1\} \longrightarrow_{\beta} \\
 & \{\lambda s. (1 = 1)\} ::= \lambda s. 1 \{\lambda s. s \ x = 1\} \longrightarrow_{\beta} \\
 & \{\lambda s. True\} x ::= \lambda s. 1 \{\lambda s. s \ x = 1\}
 \end{aligned}$$

What do we see? (You might also [check the types.](#))

The *ass* rule is such that it relates the pre-state *True* with the post-state $\lambda s. s \ x = 1$, which is [what we expect.](#)

The Inductive Definition: *Semi* and *IF – THEN – ELSE*

semi " $\llbracket \vdash \{P\}c\{Q\}; \vdash \{Q\}d\{R\} \rrbracket \implies \vdash \{P\} c;d \{R\}$ "

If " $\llbracket \vdash \{\lambda s. P s \wedge b s\}c\{Q\}; \vdash \{\lambda s. P s \wedge \neg b s\}d\{Q\} \rrbracket \implies$
 $\vdash \{P\} \text{ IF } b \text{ THEN } c \text{ ELSE } d \{Q\}$ "

Since we are reasoning about sets of states, $b s$ may sometimes be true and sometimes false, and so we have two premises for those two cases. It turns out that if $b s$ is trivially true or trivially false, then one of the premises will be trivial to prove.

The Inductive Definition: *WHILE*

While " $\vdash \{\lambda s. P\ s \wedge b\ s\} \ c \ \{P\} \implies$
 $\vdash \{P\} \ \text{WHILE } b \ \text{DO } c \ \{\lambda s. P\ s \wedge \neg b\ s\}$ "

This has a flavor of **loop invariants**: in the pre-state, $b\ s$ holds, in the post-state, $b\ s$ does not hold, and P holds all the time.

The Inductive Definition: Weakening and Strengthening

$$\text{conseq } "[\forall s. P' s \longrightarrow P s; \vdash \{P\}c\{Q\}; \forall s. Q s \longrightarrow Q' s] \Longrightarrow \vdash \{P'\}c\{Q'\}"$$

One can always **strengthen** the pre-condition or **weaken** the post-condition.

The Rules at a Glance

inductive hoare

intrs

skip " $\vdash \{P\} \text{SKIP} \{P\}$ "

ass " $\vdash \{\lambda s. P(s[x ::= a \ s])\} \ x ::= a \ \{P\}$ "

semi " $\llbracket \vdash \{P\} c \{Q\}; \vdash \{Q\} d \{R\} \rrbracket \implies \vdash \{P\} \ c; d \ \{R\}$ "

If " $\llbracket \vdash \{\lambda s. P \ s \wedge b \ s\} c \{Q\}; \vdash \{\lambda s. P \ s \wedge \neg b \ s\} d \{Q\} \rrbracket \implies$
 $\vdash \{P\} \text{ IF } b \text{ THEN } c \text{ ELSE } d \ \{Q\}$ "

While " $\vdash \{\lambda s. P \ s \wedge b \ s\} \ c \ \{P\} \implies$
 $\vdash \{P\} \text{ WHILE } b \text{ DO } c \ \{\lambda s. P \ s \wedge \neg b \ s\}$ "

conseq " $\llbracket \forall s. P' \ s \longrightarrow P \ s; \vdash \{P\} c \{Q\}; \forall s. Q \ s \longrightarrow Q' \ s \rrbracket \implies$
 $\vdash \{P'\} c \{Q'\}$ "

Validity Relation

We define a **validity relation**:

$$\models \{P\}c\{Q\} \equiv \forall s\ t. (s, t) \in C(c) \rightarrow (P\ s) \rightarrow (Q\ t)$$

Validity Relation

We define a **validity relation**:

$$\models \{P\}c\{Q\} \equiv \forall s\ t. (s, t) \in C(c) \rightarrow (P\ s) \rightarrow (Q\ t)$$

A Hoare triple $\{P\}c\{Q\}$ is **valid** if it relates a set of input states and a set of output states **correctly** w.r.t. the denotational (or **equivalently**, operational) semantics: for any input state s and output state t related by the **denotational semantics**, if P holds for s , then Q must hold for t .

Validity Relation

We define a **validity relation**:

$$\models \{P\}c\{Q\} \equiv \forall s\ t. (s, t) \in C(c) \rightarrow (P\ s) \rightarrow (Q\ t)$$

A Hoare triple $\{P\}c\{Q\}$ is **valid** if it relates a set of input states and a set of output states **correctly** w.r.t. the denotational (or **equivalently**, operational) semantics: for any input state s and output state t related by the **denotational semantics**, if P holds for s , then Q must hold for t .

Why do we raise the issue of a semantics being valid, why don't we just say "it's defined like this, full stop"?

Relating Hoare and Denotational Semantics

Theorem 9 (Hoare soundness):

$$\vdash \{P\} c \{Q\} \implies \models \{P\} c \{Q\}$$

Theorem 10 (Hoare relative completeness):

$$\models \{P\} c \{Q\} \implies \vdash \{P\} c \{Q\}$$

Why *relative*?

So the Hoare relation is in fact compatible with the denotational semantics of IMP.

Example Program

```
tm    ::= λx. 1;  
sum   ::= λx. 1;  
i      ::= λx. 0;  
WHILE λs. (s sum) ≤ (s a) DO  
  ( i      ::= λs. (s i) + 1;  
    tm     ::= λs. (s tm) + 2;  
    sum    ::= λs. (s tm) + (s sum))
```

Example Program

```
tm    ::= λx. 1;
sum   ::= λx. 1;
i      ::= λx. 0;
WHILE λs. (s sum) ≤ (s a) DO
  ( i    ::= λs. (s i) + 1;
    tm   ::= λs. (s tm) + 2;
    sum  ::= λs. (s tm) + (s sum))
```

What does this program do?

Example Program

```
tm    ::= λx. 1;  
sum   ::= λx. 1;  
i      ::= λx. 0;  
WHILE λs. (s sum) ≤ (s a) DO  
  ( i      ::= λs. (s i) + 1;  
    tm     ::= λs. (s tm) + 2;  
    sum    ::= λs. (s tm) + (s sum))
```

What does this program do?

Try $a = 1$, $a = 2$, . . . , and look at i !

Square Root

Answer: The program computes the square root. Informally:

$$Pre \equiv \text{''True''}$$

Square Root

Answer: The program computes the square root. Informally:

$$Pre \equiv \text{"True"}$$

$$Post \equiv \text{"}i^2 \leq a < (i + 1)^2\text{"}$$

Square Root

Answer: The program computes the square root. Informally:

$$\begin{aligned} Pre &\equiv \text{"True"} \\ Post &\equiv \text{"}i^2 \leq a < (i + 1)^2\text{"} \end{aligned}$$

Formally

$$Pre \equiv \lambda s. \text{True}$$

Square Root

Answer: The program computes the square root. Informally:

$$\begin{aligned} Pre &\equiv \text{"True"} \\ Post &\equiv \text{"}i^2 \leq a < (i + 1)^2\text{"} \end{aligned}$$

Formally

$$\begin{aligned} Pre &\equiv \lambda s. \text{True} \\ Post &\equiv \lambda s. (\textcolor{blue}{s } i) * (s i) \leq (s a) \wedge \\ &\quad s a < (s i + 1) * (s i + 1) \end{aligned}$$

Proving $\{Pre\} \dots \{Post\}$

We will now construct a proof tree showing that the program computes the square root.

Generally, the **difficulty** is to know when to apply *conseq*.

We try to illustrate the search for the proof tree by animation. Still you may not understand each choice immediately, but only in **hindsight**!

We use two metavariables: *Inv* for the loop invariant, *PW* for the enter condition of the loop. We instantiate later.

Abbreviation: $ExC \equiv \lambda s. Inv\ s \wedge \neg s\ sum \leq s\ a$ (“exit condition”). We omit \vdash !

Proof

$$\{Pre\} \boxed{tm \dots} \{Post\}$$

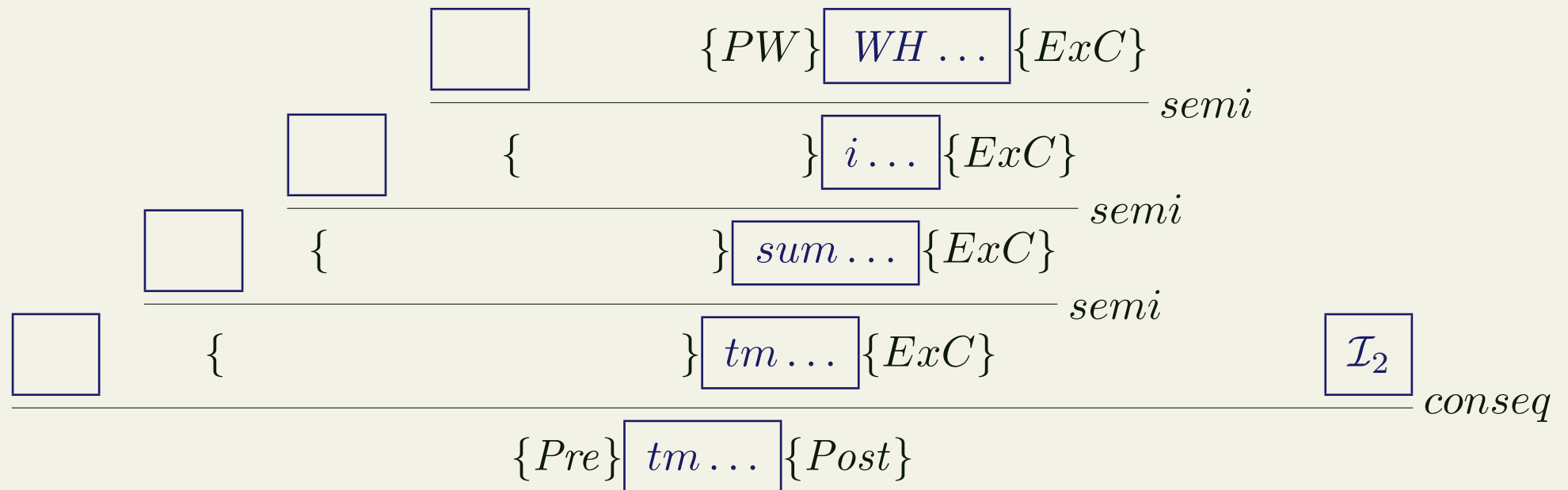
This is what we want to prove.

Proof

$$\frac{\boxed{} \quad \{ \} \quad \boxed{tm \dots} \{ ExC \} \quad \boxed{I_2}}{\{ Pre \} \boxed{tm \dots} \{ Post \}} \text{conseq}$$

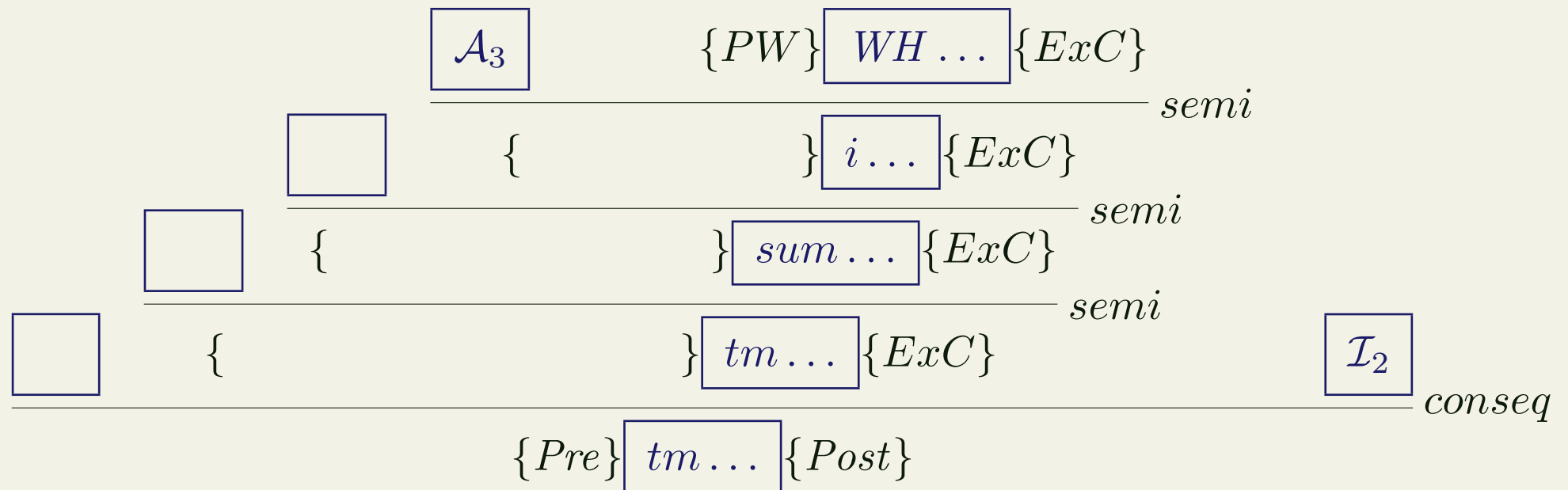
Nothing happens **after** the loop, so **intuition** says that ExC must imply $Post$.

Proof



Apply *semi* three times. *PW* (“pre while”) is just a sensible choice of name: we don’t know yet what it is.

Proof



This application of *ass* will allow us to reconstruct the precondition in the line just below.

Proof

$$\begin{array}{c}
 \boxed{A_3} \quad \{PW\} \boxed{WH \dots} \{ExC\} \\
 \hline
 \boxed{A_2} \quad \{\lambda s.PW(s["i"])\} \boxed{i \dots} \{ExC\} \quad \text{semi} \\
 \hline
 \boxed{} \quad \{ \} \boxed{sum \dots} \{ExC\} \quad \text{semi} \\
 \hline
 \boxed{} \quad \{ \} \boxed{tm \dots} \{ExC\} \quad \text{semi} \quad \boxed{I_2} \\
 \hline
 \{Pre\} \boxed{tm \dots} \{Post\} \quad \text{conseq}
 \end{array}$$

And likewise $\boxed{A_2}$.

Proof

$$\begin{array}{c}
 \boxed{A_3} \quad \{PW\} \boxed{WH \dots} \{ExC\} \\
 \hline
 \boxed{A_2} \quad \{\lambda s. PW(s["i"])\} \boxed{i \dots} \{ExC\} \quad \text{semi} \\
 \hline
 \boxed{A_1} \quad \{\lambda s. PW(s["i, sum"])\} \boxed{sum \dots} \{ExC\} \quad \text{semi} \\
 \hline
 \boxed{} \quad \{ \} \boxed{tm \dots} \{ExC\} \quad \text{semi} \quad \boxed{I_2} \\
 \hline
 \{Pre\} \boxed{tm \dots} \{Post\} \quad \text{conseq}
 \end{array}$$

And likewise $\boxed{A_1}$.

Proof

$$\begin{array}{c}
 \boxed{\mathcal{A}_3} \quad \{PW\} \boxed{WH \dots} \{ExC\} \\
 \hline
 \boxed{\mathcal{A}_2} \quad \{\lambda s.PW(s["i"])\} \boxed{i \dots} \{ExC\} \quad \text{semi} \\
 \hline
 \boxed{\mathcal{A}_1} \quad \{\lambda s.PW(s["i, sum"])\} \boxed{sum \dots} \{ExC\} \quad \text{semi} \\
 \hline
 \boxed{\mathcal{I}_1} \quad \{\lambda s.PW(s["i, sum, tm"])\} \boxed{tm \dots} \{ExC\} \quad \text{semi} \quad \boxed{\mathcal{I}_2} \\
 \hline
 \{Pre\} \boxed{tm \dots} \{Post\} \quad \text{conseq}
 \end{array}$$

We now know (by the form of *conseq*) what $\boxed{\mathcal{I}_1}$ is.

Proof

$$\begin{array}{c}
 \boxed{\mathcal{I}_3} \quad \{Inv\} \boxed{WH \dots} \{ExC\} \quad \boxed{\mathcal{I}_4} \\
 \hline
 \boxed{\mathcal{A}_3} \quad \{PW\} \boxed{WH \dots} \{ExC\} \quad \text{conseq} \\
 \hline
 \boxed{\mathcal{A}_2} \quad \{\lambda s.PW(s["i"])\} \boxed{i \dots} \{ExC\} \quad \text{semi} \\
 \hline
 \boxed{\mathcal{A}_1} \quad \{\lambda s.PW(s["i, sum"])\} \boxed{sum \dots} \{ExC\} \quad \text{semi} \\
 \hline
 \boxed{\mathcal{I}_1} \quad \{\lambda s.PW(s["i, sum, tm"])\} \boxed{tm \dots} \{ExC\} \quad \boxed{\mathcal{I}_2} \quad \text{semi} \\
 \hline
 \{Pre\} \boxed{tm \dots} \{Post\} \quad \text{conseq}
 \end{array}$$

Intuition says that PW must imply Inv .
 Of course, we are not ready yet. . .

Completing the Proof

$\boxed{A_1}$, $\boxed{A_2}$ and $\boxed{A_3}$ are complete, and $\boxed{I_4}$ is trivial.

Completing the Proof

\mathcal{A}_1 , \mathcal{A}_2 and \mathcal{A}_3 are complete, and \mathcal{I}_4 is trivial.

\mathcal{I}_1 , \mathcal{I}_2 , \mathcal{I}_3 , and $\{Inv\} WH \dots \{ExC\}$ remain to be shown.

Completing the Proof

\mathcal{A}_1 , \mathcal{A}_2 and \mathcal{A}_3 are complete, and \mathcal{I}_4 is trivial.

\mathcal{I}_1 , \mathcal{I}_2 , \mathcal{I}_3 , and $\{Inv\} WH \dots \{ExC\}$ remain to be shown.

This also involves the question how the metavariables must be instantiated.

What is PW ?

The metavariable PW (“precondition of $WHILE$ ”) must fulfill (to show $\boxed{\mathcal{I}_1}$)

$$\forall s. \textit{Pre } s \rightarrow PW(s[i ::= 0][sum ::= 1][tm ::= 1])$$

where

$$s[i ::= 0][sum ::= 1][tm ::= 1] = \lambda y. \textit{if } y = tm \textit{ then } 1 \textit{ else } \\ (\textit{if } y = sum \textit{ then } 1 \textit{ else } (\textit{if } y = i \textit{ then } 0 \textit{ else } (s \ y)))$$

What is PW ?

The metavariable PW (“precondition of *WHILE*”) must fulfill (to show $\boxed{\mathcal{I}_1}$)

$$\forall s. \textit{Pre } s \rightarrow PW(s[i ::= 0][sum ::= 1][tm ::= 1])$$

where

$$s[i ::= 0][sum ::= 1][tm ::= 1] = \lambda y. \textit{if } y = tm \textit{ then } 1 \textit{ else } \\ (\textit{if } y = sum \textit{ then } 1 \textit{ else } (\textit{if } y = i \textit{ then } 0 \textit{ else } (s \ y)))$$

Solution (recall that $\textit{Pre} \equiv \lambda s. \textit{True}$):

$$PW = \lambda s. s \ i = 0 \wedge s \ sum = 1 \wedge s \ tm = 1$$

What is *Inv*?

Continuing our proof tree construction:

$$\frac{\{\lambda s. Inv\ s \wedge s\ sum \leq s\ a\} \boxed{\text{"body"}} \{Inv\}}{\{Inv\} \boxed{WH \dots} \{Exc\}} \textit{While}$$

What is *Inv*?

Continuing our proof tree construction:

$$\begin{array}{c}
 \{\lambda s. Inv\ s \wedge s\ sum \leq s\ a\} i ::= \lambda s. s\ i + 1 \{P'\} \\
 \{P'\} tm ::= \lambda s. s\ tm + 2 \{P''\} \\
 \{P''\} sum ::= \lambda s. s\ tm + s\ sum \{Inv\} \\
 \hline
 \{\lambda s. Inv\ s \wedge s\ sum \leq s\ a\} \boxed{\text{"body"}} \{Inv\} \quad semi^2 \\
 \hline
 \{Inv\} \boxed{WH \dots} \{Exc\} \quad While
 \end{array}$$

Just blindly applying *semi* twice gives **three formulas** to be proven using *ass*, one for each assignment in the loop.

What is *Inv*?

Continuing our proof tree construction:

$$\begin{array}{c}
 \{\lambda s. Inv\ s \wedge s\ sum \leq s\ a\} i ::= \lambda s. s\ i + 1 \{P'\} \\
 \{P'\} tm ::= \lambda s. s\ tm + 2 \{P''\} \\
 \{P''\} sum ::= \lambda s. s\ tm + s\ sum \{Inv\} \\
 \hline
 \{\lambda s. Inv\ s \wedge s\ sum \leq s\ a\} \boxed{\text{"body"}} \{Inv\} \quad semi^2 \\
 \hline
 \{Inv\} \boxed{WH \dots} \{ExC\} \quad While
 \end{array}$$

Just blindly applying *semi* twice gives **three formulas** to be proven using *ass*, one for each assignment in the loop.

Now what are P' and P'' ? Have a look at rule *ass* first!

Calculating P' and P'' (by Rule *ass*)

$$P'' = \lambda s. Inv(s[sum ::= s\ tm + s\ sum])$$

Calculating P' and P'' (by Rule *ass*)

$$P'' = \lambda s. Inv(s[sum ::= s\ tm + s\ sum])$$

$$P' = \lambda s'. P''(s'[tm ::= s'\ tm + 2]) \quad (\text{rule } ass)$$

Calculating P' and P'' (by Rule *ass*)

$$P'' = \lambda s. Inv(s[sum ::= s\ tm + s\ sum])$$

$$\begin{aligned} P' &= \lambda s'. P''(s'[tm ::= s'\ tm + 2]) \quad (\text{rule } ass) \\ &= \lambda s'. (\lambda s. Inv(s[sum ::= s\ tm + s\ sum])) \\ &\quad (s'[tm ::= s'\ tm + 2]) \end{aligned}$$

Calculating P' and P'' (by Rule *ass*)

$$P'' = \lambda s. Inv(s[sum ::= s\ tm + s\ sum])$$

$$\begin{aligned} P' &= \lambda s'. P''(s'[tm ::= s'\ tm + 2]) \quad (\text{rule } ass) \\ &= \lambda s'. (\lambda s. Inv(s[sum ::= s\ tm + s\ sum])) \\ &\quad (s'[tm ::= s'\ tm + 2]) \\ &= \lambda s'. Inv((s'[tm ::= s'\ tm + 2]) \\ &\quad [sum ::= (s'[tm ::= s'\ tm + 2])\ tm + \\ &\quad (s'[tm ::= s'\ tm + 2])\ sum]) \end{aligned}$$

Calculating P' and P'' (by Rule *ass*)

$$P'' = \lambda s. Inv(s[sum ::= s\ tm + s\ sum])$$

$$\begin{aligned}
 P' &= \lambda s'. P''(s'[tm ::= s'\ tm + 2]) \quad (\text{rule } ass) \\
 &= \lambda s'. (\lambda s. Inv(s[sum ::= s\ tm + s\ sum])) \\
 &\quad (s'[tm ::= s'\ tm + 2]) \\
 &= \lambda s'. Inv((s'[tm ::= s'\ tm + 2]) \\
 &\quad [sum ::= (s'[tm ::= s'\ tm + 2])\ tm + \\
 &\quad \quad (s'[tm ::= s'\ tm + 2])\ sum]) \\
 &= \lambda s'. Inv(s'[tm ::= s'\ tm + 2] \\
 &\quad [sum ::= s'\ tm + 2 + s'\ sum]).
 \end{aligned}$$

Applying *ass* to $i ::= \lambda s. s \ i + 1$

Now treat $i ::= \lambda s. s \ i + 1$ in the same way. Temporarily, let's write P for $\lambda s. Inv \ s \wedge s \ sum \leq s \ a$. Recall $P' =$

$$\lambda s. Inv(s[tm ::= s \ tm + 2][sum ::= s \ tm + 2 + s \ sum]).$$

Applying *ass* to $i ::= \lambda s. s \ i + 1$

Now treat $i ::= \lambda s. s \ i + 1$ in the same way. Temporarily, let's write P for $\lambda s. Inv \ s \wedge s \ sum \leq s \ a$. Recall $P' =$

$$\lambda s. Inv(s[tm ::= s \ tm + 2][sum ::= s \ tm + 2 + s \ sum]).$$

$$P = \lambda s'. P'(s'[i ::= s' \ i + 1]) \quad (\text{by rule } ass)$$

Applying *ass* to $i ::= \lambda s. s\ i + 1$

Now treat $i ::= \lambda s. s\ i + 1$ in the same way. Temporarily, let's write P for $\lambda s. Inv\ s \wedge s\ sum \leq s\ a$. Recall $P' =$

$$\lambda s. Inv(s[tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum]).$$

$$\begin{aligned} P &= \lambda s'. P'(s'[i ::= s'\ i + 1]) \quad (\text{by rule } ass) \\ &= \lambda s'. (\lambda s. Inv(s[tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum])) \\ &\quad (s'[i ::= s'\ i + 1]) \end{aligned}$$

Applying *ass* to $i ::= \lambda s. s \ i + 1$

Now treat $i ::= \lambda s. s \ i + 1$ in the same way. Temporarily, let's write P for $\lambda s. Inv \ s \wedge s \ sum \leq s \ a$. Recall $P' =$

$$\lambda s. Inv(s[tm ::= s \ tm + 2][sum ::= s \ tm + 2 + s \ sum]).$$

$$\begin{aligned} P &= \lambda s'. P'(s'[i ::= s' \ i + 1]) \quad (\text{by rule } ass) \\ &= \lambda s'. (\lambda s. Inv(s[tm ::= s \ tm + 2][sum ::= s \ tm + 2 + s \ sum])) \\ &\quad (s'[i ::= s' \ i + 1]) \\ &= \lambda s'. Inv((s'[i ::= s' \ i + 1]) \\ &\quad [tm ::= (s'[i ::= s' \ i + 1]) \ tm + 2] \\ &\quad [sum ::= (s'[i ::= s' \ i + 1]) \ tm + 2 + (s'[i ::= s' \ i + 1]) \ sum])) \end{aligned}$$

Applying *ass* to $i ::= \lambda s. s \ i + 1$

Now treat $i ::= \lambda s. s \ i + 1$ in the same way. Temporarily, let's write P for $\lambda s. Inv \ s \wedge s \ sum \leq s \ a$. Recall $P' =$

$$\lambda s. Inv(s[tm ::= s \ tm + 2][sum ::= s \ tm + 2 + s \ sum]).$$

$$\begin{aligned} P &= \lambda s'. P'(s'[i ::= s' \ i + 1]) \quad (\text{by rule } ass) \\ &= \lambda s'. (\lambda s. Inv(s[tm ::= s \ tm + 2][sum ::= s \ tm + 2 + s \ sum])) \\ &\quad (s'[i ::= s' \ i + 1]) \\ &= \lambda s'. Inv((s'[i ::= s' \ i + 1]) \\ &\quad [tm ::= (s'[i ::= s' \ i + 1]) \ tm + 2] \\ &\quad [sum ::= (s'[i ::= s' \ i + 1]) \ tm + 2 + (s'[i ::= s' \ i + 1]) \ sum])) \\ &= \lambda s. Inv(s[i ::= s \ i + 1][tm ::= s \ tm + 2][sum ::= s \ tm + 2 + s \ sum]). \end{aligned}$$

Applying *ass* to $i ::= \lambda s. s \ i + 1$

$$\lambda s. Inv \ s \wedge s \ sum \leq s \ a$$

$$= \lambda s. Inv(s[i ::= s \ i + 1][tm ::= s \ tm + 2][sum ::= s \ tm + 2 + s \ sum]).$$

So *Inv* must solve this equation.

Inv Must Fulfill the Equation

Inv must fulfill the equation

$$\begin{aligned} &\lambda s. \textit{Inv } s \wedge s \textit{ sum} \leq s \textit{ a} = \\ &\lambda s. \textit{Inv}(s[i ::= s \textit{ i} + 1][tm ::= s \textit{ tm} + 2] \\ &\quad [sum ::= s \textit{ tm} + 2 + s \textit{ sum}])) \end{aligned}$$

Inv Must Fulfill the Equation

Inv must fulfill the equation

$$\begin{aligned} \forall s. \textit{Inv } s \wedge s \textit{ sum} \leq s \textit{ a} &\leftrightarrow \\ \forall s. \textit{Inv} (s[i ::= s \textit{ i} + 1] [tm ::= s \textit{ tm} + 2] \\ &\quad [sum ::= s \textit{ tm} + 2 + s \textit{ sum}]) \end{aligned}$$

Don't think syntactically! We are in **HOL**: $=$ means \leftrightarrow , and we can replace λ by \forall .

Inv Must Fulfill the Equation

Inv must fulfill the equation

$$\begin{aligned} \forall s. \textit{Inv } s \wedge s \textit{ sum} \leq s \textit{ a} &\leftrightarrow \\ \forall s. \textit{Inv} (s[i ::= s \textit{ i} + 1] [tm ::= s \textit{ tm} + 2] \\ &\quad [sum ::= s \textit{ tm} + 2 + s \textit{ sum}]) \end{aligned}$$

Don't think syntactically! We are in **HOL**: $=$ means \leftrightarrow , and we can replace λ by \forall .

Guessing the right *Inv* is obviously difficult! **Informally**

$$\textit{Inv} \equiv "(i + 1)^2 = \textit{sum} \wedge tm = (2 * i) + 1 \wedge i^2 \leq a"$$

Checking that *Inv* Fulfills equation

$$s \text{ sum} \leq s \text{ a} \quad \wedge \quad (6)$$

$$(s \text{ i} + 1)^2 = (s \text{ sum}) \quad \wedge \quad (7)$$

$$s \text{ tm} = (2 * (s \text{ i})) + 1 \quad \wedge \quad (8)$$

$$(s \text{ i})^2 \leq (s \text{ a}) \quad \wedge \quad (9)$$

$$(\text{recall: } = \text{ means } \leftrightarrow) \quad = \quad (10)$$

$$((s \text{ i} + 1) + 1)^2 = (s \text{ sum}) + (s \text{ tm}) + 2 \quad \wedge \quad (11)$$

$$(s \text{ tm} + 2) = (2 * (s \text{ i} + 1)) + 1 \quad \wedge \quad (12)$$

$$(s \text{ i} + 1)^2 \leq (s \text{ a}) \quad (13)$$

Proof Sketch

First show the “ \rightarrow ”-direction:

(8) \rightarrow (12) and (6) \wedge (7) \rightarrow (13) by simple arithmetic. (11) is shown as follows:

$$\begin{aligned} ((s\ i + 1) + 1)^2 &= (s\ i + 1)^2 + 2 * (s\ i + 1) + 1 \\ &\stackrel{(7)}{=} (s\ sum) + 2(s\ i) + 1 + 2 \\ &\stackrel{(8)}{=} (s\ sum) + (s\ tm) + 2 \end{aligned}$$

Proof Sketch (Cont.)

Now show the “ \leftarrow ”-direction:

(12) \rightarrow (8) and (13) \rightarrow (9) by simple arithmetic. (7) is shown as follows:

$$\begin{aligned}(s\ i + 1)^2 &= ((s\ i + 1) + 1)^2 - 2 * (s\ i + 1) - 1 \\ &\stackrel{(11)}{=} (s\ sum) + (s\ tm) + 2 - 2 * (s\ i + 1) - 1 \\ &\stackrel{(12)}{=} (s\ sum) + 2 * (s\ i + 1) + 1 \\ &\quad - 2 * (s\ i + 1) - 1 \\ &= s\ sum\end{aligned}$$

Finally, (7) \wedge (13) \rightarrow (6).

Proof Sketch (Cont.)

Now show the “ \leftarrow ”-direction:

(12) \rightarrow (8) and (13) \rightarrow (9) by simple arithmetic. (7) is shown as follows:

$$\begin{aligned}(s\ i + 1)^2 &= ((s\ i + 1) + 1)^2 - 2 * (s\ i + 1) - 1 \\ &\stackrel{(11)}{=} (s\ sum) + (s\ tm) + 2 - 2 * (s\ i + 1) - 1 \\ &\stackrel{(12)}{=} (s\ sum) + 2 * (s\ i + 1) + 1 \\ &\quad - 2 * (s\ i + 1) - 1 \\ &= s\ sum\end{aligned}$$

Finally, (7) \wedge (13) \rightarrow (6). So *Inv* is indeed an invariant!

The *WHILE* Loop: Remarks

We have shown

$$(\text{"enter condition"} \wedge \text{"invar. at entry"}) \leftrightarrow \text{"invar. at exit"}$$

The *WHILE* Loop: Remarks

We have shown

$(\text{“enter condition”} \wedge \text{“invar. at entry”}) \leftrightarrow \text{“invar. at exit”}$

One would definitely expect \rightarrow , but \leftarrow is remarkable!

The *WHILE* Loop: Remarks

We have shown

$(\text{“enter condition”} \wedge \text{“invar. at entry”}) \leftrightarrow \text{“invar. at exit”}$

One would definitely expect \rightarrow , but \leftarrow is remarkable!

We can show this because our invariant is so **strong**: for showing \rightarrow , the **weaker** invariant $(7) \wedge (8)$, i.e.

$$”(i + 1)^2 = sum \wedge tm = (2 * i) + 1$$

would do (check it!).

The *WHILE* Loop: Remarks

We have shown

$(\text{"enter condition"} \wedge \text{"invar. at entry"}) \leftrightarrow \text{"invar. at exit"}$

One would definitely expect \rightarrow , but \leftarrow is remarkable!

We can show this because our invariant is so **strong**: for showing \rightarrow , the **weaker** invariant $(7) \wedge (8)$, i.e.

$$"(i + 1)^2 = \text{sum} \wedge \text{tm} = (2 * i) + 1$$

would do (check it!).

But the extra condition $i^2 \leq a$ is needed for showing *Post*, which states what the program actually computes.

Taking Care of $Post$

We have shown $\boxed{\mathcal{I}_1}$ and $\{Inv\} \boxed{WH \dots} \{ExC\}$. Now continue with $\boxed{\mathcal{I}_2}$.

Does $Post\ s$ follow from $Inv\ s \wedge \neg s\ sum \leq s\ a$?

Taking Care of $Post$

We have shown $\boxed{\mathcal{I}_1}$ and $\{Inv\} \boxed{WH \dots} \{ExC\}$. Now continue with $\boxed{\mathcal{I}_2}$.

Does $Post\ s$ follow from $Inv\ s \wedge \neg s\ sum \leq s\ a$?

Yes!

$(s\ i)^2 \leq (s\ a)$ follows from (9)

$(s\ a) < (s\ i + 1)^2$ follows from $\neg s\ sum \leq (s\ a)$ and (7).

The Final Missing Part

\mathcal{I}_3 remains to be shown, i.e.

$$\forall s. PW\ s \rightarrow Inv\ s$$

or, expanding the solutions for PW and Inv

$$\begin{aligned} \forall s. \quad & s\ i = 0 \wedge s\ sum = 1 \wedge s\ tm = 1 \rightarrow \\ & (s\ i + 1)^2 = s\ sum \wedge \\ & s\ tm = (2 * (s\ i)) + 1 \wedge \\ & (s\ i)^2 \leq (s\ a) \end{aligned}$$

This is easy to check.

An Alternative for Tackling the Loop Part

Recall that our loop invariant was “too strong”. An alternative:

$$\frac{\{\lambda s. Inv\ s \wedge s\ sum \leq s\ a\} \boxed{\text{"body"}} \{Inv\}}{\{Inv\} \boxed{WH \dots} \{Exc\}} \textit{While}$$

An Alternative for Tackling the Loop Part

Recall that our loop invariant was “too strong”. An alternative:

$$\begin{array}{c}
 \forall s. (Inv\ s \wedge \\
 s\ sum \leq s\ a) \rightarrow \\
 Inv'\ s
 \end{array}
 \quad
 \frac{\{Inv'\} \boxed{\text{"body"}} \{Inv\}}{conseq}$$

$$\frac{\{\lambda s. Inv\ s \wedge s\ sum \leq s\ a\} \boxed{\text{"body"}} \{Inv\}}{While}$$

$$\{Inv\} \boxed{WH \dots} \{Exc\}$$

An Alternative for Tackling the Loop Part

Recall that our loop invariant was “too strong”. An alternative:

$$\begin{array}{c}
 \{Inv'\} i ::= \lambda s. s \ i + 1 \{P'\} \\
 \{P'\} tm ::= \lambda s. s \ tm + 2 \{P''\} \\
 \{P''\} sum ::= \lambda s. s \ tm + s \ sum \{Inv\} \\
 \hline
 \forall s. (Inv \ s \wedge s \ sum \leq s \ a) \rightarrow Inv' \ s \quad \frac{\{P''\} sum ::= \lambda s. s \ tm + s \ sum \{Inv\}}{\{Inv'\} \boxed{\text{"body"}} \{Inv\}} \text{semi}^2 \\
 \hline
 \{ \lambda s. Inv \ s \wedge s \ sum \leq s \ a \} \boxed{\text{"body"}} \{Inv\} \quad \text{conseq} \\
 \hline
 \{Inv\} \boxed{WH \dots} \{ExC\} \quad \text{While}
 \end{array}$$

Alternative (Cont.)

Applying *ass* as before gives

$$Inv' = \lambda s. Inv(s[i ::= s\ i + 1][tm ::= s\ tm + 2] \\ [sum ::= s\ tm + 2 + s\ sum])$$

We are left with the proof obligation

$$\forall s. (Inv\ s \wedge s\ sum \leq s\ a) \rightarrow Inv(s[i ::= s\ i + 1] \\ [tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum])$$

Just this could be shown setting *weak Inv* $\equiv (7) \wedge (8)$, but for actually showing *Post*, $i^2 \leq a$ is still needed.

Automating Hoare Proofs

In the **example**, we have verified a program computing the square root.

But this was tedious, and parts of the task can be automated.

Weakest Liberal Preconditions

Observation: the Hoare relation is deterministic to a certain extent.

Idea: we use this fact for the generation of (weakest liberal) preconditions.

Weakest liberal preconditions are:

constdefs wp :: com \Rightarrow assn \Rightarrow assn

"wp c Q $\equiv (\lambda s. \forall t. (s, t) : C(c) \longrightarrow Q t)$ "

So $wp\ c\ Q$ returns the set of states containing all states s such that if t is reached from s via c , then the post-condition Q holds for t . Computable?

Weakest Liberal Preconditions

Observation: the Hoare relation is deterministic to a certain extent.

Idea: we use this fact for the generation of (weakest liberal) preconditions.

Weakest liberal preconditions are:

constdefs wp :: com \Rightarrow assn \Rightarrow assn

"wp c Q $\equiv (\lambda s. \forall t. (s, t) : C(c) \longrightarrow Q\ t)$ "

So $wp\ c\ Q$ returns the set of states containing all states s such that if t is reached from s via c , then the post-condition Q holds for t . Computable? Not obvious.

Equivalence Proofs

Main results of the wp-generator are:

- wp_SKIP: $\text{wp SKIP } Q = Q$
- wp_Ass: $\text{wp } (x ::= a) Q = (\lambda s. Q (s[x ::= a \ s]))$
- wp_Semi: $\text{wp } (c; d) Q = \text{wp } c (\text{wp } d Q)$
- wp_If: $\text{wp } (\text{IF } b \text{ THEN } c \text{ ELSE } d) Q =$
 $(\lambda s. (b \ s \longrightarrow \text{wp } c Q \ s) \wedge (\neg b \ s \longrightarrow \text{wp } d Q \ s))$
- wp_While_True: $b \ s \Longrightarrow \text{wp } (\text{WHILE } b \text{ DO } c) Q \ s =$
 $\text{wp } (c; \text{WHILE } b \text{ DO } c) Q \ s$
- wp_While_False: $\neg b \ s \Longrightarrow \text{wp } (\text{WHILE } b \text{ DO } c) Q \ s = Q \ s$
- wp_While_if: $\text{wp } (\text{WHILE } b \text{ DO } c) Q \ s =$
 $(\text{if } b \ s \text{ then } \text{wp}(c; \text{WHILE } b \text{ DO } c) Q \ s \text{ else } Q \ s)$

WP-Semantics

Except for termination problem due to *While*, (weakest liberal) precondition wp can be computed.

This fact can be used for further proof support by verification condition generation.

Verification Condition Generation

First, we must enrich the syntax by loop-invariants:

```
datatype acom =  
  | Skip  
  | Aass loc aexp  
  | Asemi acom acom  
  | Aif bexp acom acom  
  | Awhile bexp assn acom
```

Almost same as *com*, but *While* gets an additional argument for asserting a loop invariant. Asserting this is the difficult, creative step to be done by a human.

Computing a Weakest Liberal Precondition

We define a function that computes a **wp**:

primrec

"awp Askip Q = Q"

"awp (Aass x a) Q = ($\lambda s. Q(s[x ::= a \ s])$)"

"awp (Asemi c d) Q = awp c (awp d Q)"

"awp (Aif b c d) Q = ($\lambda s. (b \ s \longrightarrow \text{awp } c \ Q \ s) \wedge (\neg b \ s \longrightarrow \text{awp } d \ Q \ s)$)"

"awp (Awhile b Inv c) Q = Inv"

Idea: for all statements, the **exact wp** is computed, except for *While*, where the assertion provided by the user is taken as approximation. **Proof obligation**: show that such an assertion is compatible with the program and the desired property . . .

A Verification Condition

Construct a formula $vc\ c\ Q\ s$ with the intuitive reading: as far as the **invariant assertions** are concerned, s is a good pre-state for reaching desired post-property Q using **annotated program** c . Invariants only occur in *While* statements, so $vc\ (Awhile\ b\ Inv\ c)\ Q\ s$ is only interesting case. What should hold?

A Verification Condition

Construct a formula $vc\ c\ Q\ s$ with the intuitive reading: as far as the **invariant assertions** are concerned, s is a good pre-state for reaching desired post-property Q using **annotated program** c . Invariants only occur in *While* statements, so $vc\ (Awhile\ b\ Inv\ c)\ Q\ s$ is only interesting case. What should hold?

- The body ensures the invariant: $vc\ c\ Inv\ s$;
- invariant + exit condition imply Q : $Inv\ s \wedge \neg(b\ s) \rightarrow Q\ s$;
- invariant + loop condition imply **precondition of invariant**: $Inv\ s \wedge (b\ s) \rightarrow awp\ c\ Inv\ s$.

Verification Condition Generation

For all other constructs, vc is quite trivial:

primrec

"vc Askip Q = ($\lambda s. \text{True}$)"

"vc (Aass x a) Q = ($\lambda s. \text{True}$)"

"vc (Asemi c d) Q = ($\lambda s. vc\ c\ (\text{awp}\ d\ Q)\ s \wedge vc\ d\ Q\ s$)"

"vc (Aif b c d) Q = ($\lambda s. vc\ c\ Q\ s \wedge vc\ d\ Q\ s$)"

"vc (Awhile b Inv c) Q = ($\lambda s. (\text{Inv}\ s \wedge \neg b\ s \longrightarrow Q\ s) \wedge$
 $(\text{Inv}\ s \wedge b\ s \longrightarrow \text{awp}\ c\ \text{Inv}\ s) \wedge vc\ c\ \text{Inv}\ s$)"

vc is not about distinguishing good pre-states from bad pre-states, but just about formalising good invariant assertions. So we want $\forall s. vc\ c\ Q\ s$, i.e., $vc\ c\ Q \equiv \lambda s. \text{True}$.

Equivalence Proofs

Main results of the wp-generator are:

$\text{vc_sound}: \forall Q. (\forall s. \text{vc } ac \ Q \ s) \longrightarrow \vdash \{ \text{awp } ac \ Q \} \text{ astrip } ac \ \{ Q \}$

$\text{vc_complete}: \vdash \{ P \} \ c \ \{ Q \}$

$\implies \exists ac. \text{astrip } ac = c \wedge (\forall s. \text{vc } ac \ Q \ s) \wedge (\forall s. P \ s \longrightarrow \text{awp } ac \ Q \ s)$

The function *astrip* turns *acom* programs into *com* programs by removing the loop invariant assertions.

Quintessence: To prove that a program *c* has property *Q* after execution, you must *annotate* it with loop invariants (*ac*) and show $\forall s. \text{vc } ac \ Q \ s$. This implies that a Hoare proof exists, for the computable precondition *awp ac Q*. For good (robust) programs, $\text{awp } ac \ Q = \lambda s. \text{True}$.

Summary

IMP closely follows the standard textbook [Win96].

Isabelle/HOL is a powerful framework for embedding imperative languages.

Isabelle/HOL is also a framework for state-of-the-art languages like JAVA including interfaces, inheritance, dynamic methods.

It works in theory and for non-trivial problems in practice (but of modest size). ►

More Detailed Explanations

Flavors of Semantics

One distinguishes

- operational,
- denotational,
- axiomatic

semantics.

For **operational semantics**, the idea is that our machine is always in some **state**, essentially consisting of the values of the **program variables**. The instructions of a program transform a state into a new state. Operational semantics are useful for compiler construction.

For **denotational semantics**, the idea is that the meaning of a particular program is a **relation** between “input” states and “output” states.

Axiomatic semantics consist of a calculus for constructing proof

obligations.

Imperative Languages in the HOL Library

You should find directories for each mentioned imperative language in your Isabelle distribution. In Freiburg it is in

```
/usr/local/isabelle/Isabelle/src/HOL/
```

Shallow and Deep

The notions shallow and deep refer to the way the imperative language at hand is encoded in Isabelle.

Equivalence Proofs

Summarizing, we have the following equivalence results:

- natural vs. transition semantics
- denotational vs. natural semantics

Com.thy

This file defines the command syntax. An Isabelle term of type *com* is an IMP program.

You should find the files in your Isabelle distribution. In Freiburg they are in

```
/usr/local/isabelle/Isabelle/src/HOL/IMP/
```

Locations

We realize program variables via pointers (locations). The type of pointers is an **abstract datatype**.

We take the type of **values** to be *nat*, just to have something simple.

A **state** is a function taking a location to a value, i.e. intuitively, each program variable has a value in a state.

Abstractness of *aexp* and *bexp*

In a formalization of the syntax of an imperative language, there will usually be some **grammar saying that** 1 , $x + 1$ (provided that x is an arithmetic variable) etc. are **arithmetic expressions** and that $True$, $x == 1$ etc. are **Boolean expressions**. Such expressions can only be evaluated if the **state**, i.e. the value of the program variables, is given. Now, our notion of expressions (as realized by the types *aexp* and *bexp*) is much more abstract than that. An expression is a **function** taking a **state** to a value or Boolean, as applicable.

The fact that IMP has no explicit expression language allows for simple and abstract proofs.

The Intuition of Natural Semantics

The idea of the natural semantics is that a program relates two states, the “input state” and the “output state”.

This may remind you of **denotational** semantics, and in fact, the natural semantics is a kind of hybrid between operational and denotational semantics.

The fact that the natural semantics just relates an “input state” and an “output state” means, so to say, that it does not record what happens in between, i.e. at the single steps of a computation. In that respect, it resembles denotational semantics.

But the way the meaning of a whole program is defined is still operational in nature. Essentially, it is **defined** in terms of the meaning of the first execution step and the meaning of the rest of the program.

The Intuition of Transition Semantics

Unlike the **natural semantics**, the transition semantics records the single steps of the computation. A **configuration** is a pair consisting of a program and a state, and one step reaches a new program and a new state.

Why “reaching a new program”? This realizes a **program counter**. For example, if the first line of the program is an assignment, then the new program is obtained by removing that line from the old program.

The Power of Paraphrasing

As you see, paraphrasing in Isabelle is very powerful. One can think of $-c \rightarrow$ and $-1 \rightarrow$ as **infix symbols**. But $-n \rightarrow$ is by no means one single symbol. In fact the term $cs_0 -n \rightarrow cs_1$ is a paraphrasing of $(cs_0, cs_1) \in evalc1^n$.

Relation_Power.thy

This file should be contained in your Isabelle distribution. In Freiburg it is

`/usr/local/isabelle/Isabelle/src/HOL/Relation_Power.thy`

Define C by Primitive Recursion

Recall that the `primrec` syntax is used for defining functions recursively. Here, the argument type of the function C is the datatype `com`. It is characteristic for the definition of a datatype that its elements are defined by (structural) induction, i.e., its elements are syntactic terms formed from previously generated syntactic forms using a specific set of **term constructors**. For datatypes, it is clear that the **subterm relation** is a well-founded order. Hence it is legitimate to define C using recursion.

Understanding Γ

Let's try to understand

$$\Gamma b \text{ cd} \equiv (\lambda \phi. \{(s, t). (s, t) : (\phi \circ \text{cd}) \wedge b(s)\} \cup \{(s, t). s=t \wedge \neg b(s)\})$$

Note that in the definition of *WHILE*, the second argument to Γ is $(C \text{ c})$, which is the semantics of the **body** of the *WHILE* statement, i.e., a binary relation between input states and output states. So *cd* above is the semantics of the body of the *WHILE* statement.

Let

$$S_0 = \{(s, t) \mid s = t \wedge \neg b(s)\}$$

and for $i > 0$,

$$S_i = \{(s, t) \mid \exists s_0, \dots, s_i. s_0 = s \wedge s_i = t \wedge (s_0, s_1) \in cd \wedge \dots \wedge (s_{i-1}, s_i) \in cd \wedge b(s_0) \wedge \dots \wedge b(s_{i-1})\}$$

When we apply $(\Gamma b \, cd)$ to \emptyset (the empty relation), then since $\emptyset \circ cd = \emptyset$, we get just S_0 . Now it is easy to see that $(\Gamma b \, cd) S_0 = S_0 \cup S_1$, and generally, for all $i > 0$

$$(\Gamma b \, cd)(\bigcup_{j \leq i} S_j) = \bigcup_{j \leq i+1} S_j$$

So the intuition of the functional $(\Gamma b \, cd)$ is as follows: given the semantics of the *WHILE* statement for up to i passes through the loop, return the semantics of the *WHILE* statement for up to $i + 1$ passes

through the loop. The *lfp* of this functional is the actual semantics of the *WHILE* statement.

Is the Axiomatic Semantics Really Axiomatic?

In terms of Isabelle/HOL, the semantics is not defined by **axioms**, but is an **inductive definition**.

The Types in the Hoare Assignment Rule

Things are getting a bit complicated, maybe it helps to recall the types of the terms occurring in

$$\text{ass } "\vdash \{ \lambda s. P(s[x::=(a \ s)]) \} \ x:=a \ \{P\}"$$

P has type $assn$, **which is** $state \Rightarrow bool$. In turn , $state$ **is** $loc \Rightarrow val$.

x has type loc .

a has type $aexp$, **which is** $state \Rightarrow val$.

s has type $state$.

The Assignment Rule

You can also argue a bit more generally. Let Q be an **arbitrary** assertion, and let

$$P \equiv \lambda s. \exists s'. Q\ s' \wedge s = s'[x ::= (a\ s')]$$

Intuitively: P is an assertion allowing any state obtained from a state allowed by Q by updating that state at location x with the expression a . Now consider the rule for assignment:

$$\text{ass } "\vdash \{ \lambda s. P(s[x ::= (a\ s)]) \} \ x ::= a \ \{ P \} "$$

in particular the assertion on the left-hand side. It reduces as follows:

$$\begin{aligned} & \lambda s. \underline{P(s[x ::= (a\ s)])} \longrightarrow_{\beta} \\ & \lambda s. (\exists s'. Q\ s' \wedge s[x ::= (a\ s)] = s'[x ::= (a\ s')]) \longrightarrow_{\beta} \dots \\ & \lambda s. (\exists s'. Q\ s' \wedge s = s') \longrightarrow_{\beta} \dots \lambda s. (Q\ s) \longrightarrow_{\eta} Q \end{aligned}$$

So you see that any pre-state Q will be related to a post-state P as

given above.

By this argument, we have only shown which post-states **are** possible given an arbitrary pre-state, not which post-states **are not**. Such an argument is more complicated.

A Table of Values

a is not modified anywhere. You should think of a as input of the program.

i counts the number of times the loop is entered, i.e. the final value of i is the number of times the loop was entered. This number depends on a . The following table shows that final values of i , tm and sum depending on the value of a :

	i	tm	sum
$0 \leq a < 1$	0	1	1
$1 \leq a < 4$	1	3	4
$4 \leq a < 9$	2	5	9
$9 \leq a < 16$	3	7	16
$16 \leq a < 25$	4	9	25
$25 \leq a < 36$	5	11	36
$36 \leq a < 49$	6	13	49

sum takes the values of all squares successively, computed by the famous binomial formula:

$$(i + 1)^2 = i^2 + 2i + 1$$

Since tm takes the value $2i + 1$ for all i successively, it follows that

$sum + tm$ always gives the next value of sum .

$(s\ i)$, $(s\ a)$ **etc.**

Informally we talk about variables i , x etc. and say “ x has value 5”, for example. But formally, variables are realized via **locations**, and so we get expressions of the form $s\ x$. That is, $s\ x$ is the value of variable x .

Nondeterminacy in the Hoare Calculus

The *conseq* rule can always be applied. If one decides not to apply the *conseq* rule, then the choice of *any other rule* is deterministic.

Why Do We Question Hoare?

You may wonder: Why do we raise the issue of a semantics being valid, why don't we just say "it's defined like this, full stop"? After all, we didn't question the operational and denotational semantics in the same way. So why do we take the denotational semantics as **the real** semantics of a program that another semantics such as the Hoare semantics has to be somehow equivalent to in order to be correct? Couldn't we do it the other way round?

Why Do We Question Hoare?

You may wonder: Why do we raise the issue of a semantics being valid, why don't we just say "it's defined like this, full stop"? After all, we didn't question the operational and denotational semantics in the same way. So why do we take the denotational semantics as **the real** semantics of a program that another semantics such as the Hoare semantics has to be somehow equivalent to in order to be correct? Couldn't we do it the other way round?

First: If you want to accept anything as **the real** semantics of a program, it would be the **transition semantics**, since we believe that by the transition semantics, we have modeled what the compiler of the programming language actually does. The transition semantics records the actual **computation steps**.

Secondly, we have shown that the transition semantics is equivalent to

the **natural semantics**, which in turn is equivalent to the **denotational semantics**.

Together this explains why we call a Hoare triple valid if it is correct w.r.t. the denotational semantics.

Relative Completeness

We will not give any details here, but the completeness result is restricted in the **same way that the completeness of HOL** is restricted to **general** models, as opposed to standard models.

Program “Fragment”

This is the entire program, namely:

$tm ::= \lambda x. 1;$

$sum ::= \lambda x. 1;$

$i ::= \lambda x. 0;$

WHILE $\lambda s. s \text{ sum} \leq s \text{ a}$ DO

$(i ::= \lambda s. s \ i + 1;$

$tm ::= \lambda s. s \ tm + 2;$

$sum ::= \lambda s. s \ tm + s \ sum)$

(return to main proof tree)

Program Fragment

This is the program fragment starting from $sum ::=$, namely:

$sum ::= \lambda x. 1;$

$i ::= \lambda x. 0;$

WHILE $\lambda s. s \text{ sum} \leq s \text{ a}$ DO

$(i ::= \lambda s. s \ i + 1;$

$tm ::= \lambda s. s \ tm + 2;$

$sum ::= \lambda s. s \ tm + s \ sum)$

(return to main proof tree)

Program Fragment

This is the program fragment starting from $i ::=$, namely:

```
i ::=  $\lambda x. 0$ ;  
WHILE  $\lambda s. s \text{ sum} \leq s a$  DO  
  ( $i ::= \lambda s. s i + 1$ ;  
    $tm ::= \lambda s. s tm + 2$ ;  
    $sum ::= \lambda s. s tm + s \text{ sum}$ )  
(return to main proof tree)
```

Program Fragment

This is the program fragment starting from *WHILE*, namely:

```
WHILE  $\lambda s. s \text{ sum} \leq s a$  DO  
  (  $i \quad := \lambda s. s i + 1$ ;  
     $tm \quad := \lambda s. s tm + 2$ ;  
     $sum := \lambda s. s tm + s sum$  )  
(return to main proof tree)
```

Program Fragment

This is the program fragment consisting of the loop body, namely:

```
i      ::= λs. s i + 1;  
tm     ::= λs. s tm + 2;  
sum    ::= λs. s tm + s sum  
(return to main proof tree)
```

A Missing Part

$\boxed{\mathcal{I}_1}$ is the formula

$$\forall s. Pre\ s \rightarrow PW(s[\"i, sum, tm\"])$$

where Pre is defined above and PW is a metavariable (“precondition of *WHILE*”).

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{I}_2}$ is the formula

$$\forall s. \textit{ExC} \ s \rightarrow \textit{Post} \ s,$$

i.e.

$$\forall s. \textit{Inv} \ s \wedge \neg \textit{sum} \ s \leq s \ a \rightarrow \textit{Post} \ s,$$

where \textit{Post} is defined [above](#) and \textit{Inv} is a metavariable (“loop invariant”).
([return to main proof tree](#))

A Missing Part

$\boxed{\mathcal{A}_1}$ is the proof tree

$$\frac{\{\lambda s.PW(s["i, sum, tm"])\}tm ::= \lambda x.1\{\lambda s.PW(s["i, sum"])\}}{ass}$$

where PW is a metavariable (“precondition of *WHILE*”).

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{A}_2}$ is the proof tree

$$\frac{\{\lambda s.PW(s[i ::= 0][sum ::= 1])\}sum ::= \lambda x.1\{\lambda s.PW(s[i ::= 0])\}}{ass}$$

where PW is a metavariable (“precondition of *WHILE*”).

(return to main proof tree)

A Missing Part

$\boxed{A_3}$ is the proof tree

$$\frac{}{\{\lambda s.PW(s[i ::= 0])\}i ::= \lambda x.0\{PW\}}^{ass}$$

where PW is a metavariable (“precondition of *WHILE*”).
(return to main proof tree)

A Missing Part

$\boxed{\mathcal{I}_3}$ is the formula

$$\forall s. PW\ s \rightarrow Inv\ s$$

where PW is a metavariable (“precondition of *WHILE*”) and Inv is a metavariable (“loop invariant”).

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{I}_4}$ is the formula

$$\forall s. \textcolor{blue}{ExC} \ s \rightarrow ExC \ s$$

which is of course trivial to prove.

(return to main proof tree)

An Abbreviation for an Updated State

We use $s["i, sum, tm"]$ as abbreviation for

$$s[i ::= 0][sum ::= 1][tm ::= 1]$$

Note that this is

$$\lambda y. \text{if } y = tm \text{ then } 1 \text{ else} \\ (\text{if } y = sum \text{ then } 1 \text{ else} (\text{if } y = i \text{ then } 0 \text{ else } (s\ y)))$$

(return to main proof tree)

An Abbreviation for an Updated State

We use $s["i, sum"]$ as abbreviation for

$$s[i ::= 0][sum ::= 1]$$

Note that this is

$$\lambda y. \text{if } y = \text{sum} \text{ then } 1 \text{ else } (\text{if } y = i \text{ then } 0 \text{ else } (s\ y))$$

(return to main proof tree)

An Abbreviation for an Updated State

We use $s[\"i\"]$ as abbreviation for

$$s[i ::= 0]$$

Note that this is

$$\lambda y. \text{if } y = i \text{ then } 0 \text{ else } (s \ y)$$

(return to main proof tree)

Displaying Proof Tree

Of course, these three formulas should be side by side in the proof tree, but this cannot be displayed.

What must *Inv* Be?

Recall that we had to prove the three formulas

$$\begin{aligned} \{\lambda s. Inv\ s \wedge s\ sum \leq s\ a\}i &::= \lambda s. s\ i + 1\{P'\} \\ \{P'\}tm &::= \lambda s. s\ tm + 2\{P''\} \\ \{P''\}sum &::= \lambda s. s\ tm + s\ sum\{Inv\} \end{aligned}$$

all by *ass*. Dealing with the second and third formula using *ass*, we found that

$$P' = \lambda s'. Inv(s'[tm ::= s' tm + 2][sum ::= s' tm + 2 + s' sum]).$$

Therefore, to show

$$\{\lambda s. Inv\ s \wedge s\ sum \leq s\ a\}i ::= \lambda s. s\ i + 1\{P'\}$$

as well, Inv must have such a form that the formula becomes an instance of ass .

Why “Precondition of Invariant

Why should invariant + loop condition imply the precondition of the invariant?

To achieve that the invariant will still hold after another unfolding of the *While*-loop).

A Taste of some Isabelle and HOL Applications

Just a few Isabelle or HOL Applications

We briefly introduce two Isabelle/HOL applications, and one application of HOL Light:

- Java bytecode verification;
- floating-point arithmetic;
- red-black trees.

This is just to stimulate you to look for more applications on your own!

Java Bytecode Verification

Typically, Java programs are delivered as **bytecode**, as opposed to **source** code on the one hand and **machine** code on the other hand. Bytecode is **machine-independent**.

A Java runtime system provides the **Java Virtual Machine**, i.e., an interpreter for Java bytecode.

Java is a **typed** language: the type system forbids things like pointer arithmetic, thus preventing **illegal** memory access.

However, bytecode is not type-safe by itself. For various reasons, bytecode could be corrupted. This is obviously critical for security and possibly safety.

Ensuring Type Safety

The loader of a typical JVM has a **bytecode verifier**: A program that checks whether bytecode is type-safe.

Klein and Nipkow have specified a JVM and a bytecode verifier in Isabelle and proved its correctness using Isabelle [KN03, Nip03].

Such applications may have big impact since they are concerned with the correctness of not just some particular program, but rather the programming language (implementation) itself.

JavaCard

JavaCard is a subset of Java employed on **smart cards**.

Aspects in contrast to full Java:

- Memory on smart cards is **limited**.
- Security is vital for smart card applications (banking etc.).

Project Verificard concerned with ensuring reliability of smart card applications.

Verificard @ Munich have applied the work on bytecode verification (using Isabelle) to JavaCard.

End user panel includes Ericsson, France Télécom R&D, and Gemplus.

Floating Point Arithmetic

John Harrison has done much work on verifying arithmetic functions operating on various number types adhering to certain standards [[Har98](#), [Har99](#), [Har00](#)].

He has used HOL Light, not Isabelle. This means: no metalogic, specialized theorem prover for HOL.

He formally proved that the floating point operations of an Intel processor behave according to the IEEE standard 754 [[IEEE85](#)]. First machine-checked proof of this kind.

We briefly review his work [[Har99](#)] using an Isabelle-like syntax where helpful.

What Are Floats?

Conventionally: floats have the form $\pm 2^e \cdot k$.

e is called **exponent**, $E_{min} \leq e \leq E_{max}$.

k is called **mantissa**, can be represented with p bits.

Floats in HOL

For formalization in HOL, equivalent representation

$$(-1)^s \cdot 2^{e-N} \cdot k$$

with $k < 2^p$ and $0 \leq e < E$.

Thus a particular float **format** is characterized by maximal exponent E , precision p , and exponent offset (“ulpscale”) N . The set of **real** numbers representable by a triple is:

$$\text{format } (E, p, N) = \{x \mid \exists s \, e \, k. s < 2 \wedge e < E \wedge k < 2^p \wedge x = (-1)^s \cdot 2^e \cdot k / 2^N\}$$

Rounding

Rounding takes a real to a **representable** real nearby.

E.g. rounding up:

$$\begin{aligned} \text{round } \text{fmt } x = \epsilon a. & \ a \in \text{format } \text{fmt} \wedge a \leq x \wedge \\ & \forall b \in \text{format } \text{fmt}. b \leq x \rightarrow b \leq a \end{aligned}$$

Formalization of the Standard [IEEE85].

Useful lemmas such as:

$$\begin{aligned} x \leq y & \implies \text{round } \text{fmt } x \leq \text{round } \text{fmt } y \\ a \in \text{format } \text{fmt} \wedge b \in \text{format } \text{fmt} \wedge 0.5 \leq \frac{a}{b} \leq 2 & \implies \\ (b - a) & \in \text{format } \text{fmt} \end{aligned}$$

Operations

For operations such as addition, multiplication etc., it is proven in HOL that they behave as if they computed the exact result and rounded afterwards.

However, there are some debatable questions related to the **sign of zeros**.

Red-Black Trees

Red-black trees are trees that can be used for implementing sets/dictionaries, just like AVL trees. To formulate “balanced-ness” invariants, nodes are **colored**:

1. Every red node has a black parent.
2. Each path from the root to a leaf has the same number of black nodes.

Together these invariants ensure that maximal paths can differ in length by at most factor 2.

These invariants must be maintained by insertion and deletion operations.

Red-Black Trees in SML

Red-black trees provided in New Jersey SML library [Pau96].
Angelika Kimmig tried to verify the insertion operation of red-black trees using Isabelle. Findings?

Red-Black Trees in SML

Red-black trees provided in New Jersey SML library [Pau96]. Angelika Kimmig tried to verify the insertion operation of red-black trees using Isabelle. Findings?

- There is a mistake in the implementation of red-black trees in New Jersey SML! Insertion may lead to a violation of the first invariant, since the root may become red.
- As long as one just inserts, this is just a slight constant deterioration.
- Angelika has suggested a fix and proven the correctness of red-black tree insertion using Isabelle.

Node Deletion

- **Deletion** is also wrongly implemented!
- With deletion, not just the root can become red, but the tree coloring can become completely wrong.
- Angelika has an idea for fixing deletion as well, but no proof yet.

Read the Studienarbeit for more details [[Kim03](#)]!

More Detailed Explanations

Illegal Memory Access

By “illegal memory access”, we mean access to regions not assigned to the program.

Limited Memory on Smart Cards

The memory on smart cards is limited. A full-fledged bytecode verifier would be too large/slow. One approach to tackling this problem is to work with bytecode programs with **type annotations**. Checking if a bytecode program is consistent with its type annotations is a much simpler task than computing these type annotations, which is what a bytecode verifier is supposed to do. The task can therefore be performed on a smart card more easily than full bytecode verification.

Angelika Kimmig

Angelika Kimmig is a student who took this course in Wintersemester 02/03 in Freiburg. She then continued working with Isabelle in a Studienarbeit (a project required by computer science students in Freiburg).

References

- [Acz77] Peter Aczel. *Handbook of Mathematical Logic*, chapter An Introduction to Inductive Definitions, pages 739–782. North-Holland, 1977.
- [AHMP92] Arnon Avron, Furio Honsell, Ian A. Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, 1992.
- [And02] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through*

Proofs. Kluwer Academic Publishers, 2002. Second Edition.

[Apt97] Krzysztof R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.

[Ari] Aristotle. *Analytica priora I*, chapter 4.

[Ber19] Paul Bernays. ?? ??, 19??

[BM00] David A. Basin and Seàn Matthews. Structuring metatheory on inductive definitions. *Information and Computation*, 162(1-2):80–95, 2000. Download.

- [BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- [Can18] Georg Cantor. ?? ??, 18??
- [Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
- [dB80] Nicolaas G. de Bruijn. A survey of the project AUTOMATH. In *Essays in Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

- [Des16] Rene Descartes. ?? ??, 16??
- [Dev93] Keith Devlin. *The Joy of Sets. Fundamentals of Contemporary Set Theory*. Undergraduate Texts in Mathematics. Springer-Verlag, 1993.
- [Ebb94] Heinz-Dieter Ebbinghaus. *Einführung in die Mengenlehre*. BI-Wissenschaftsverlag, 1994.
- [Fit96] M. Fitting. *First-order Logic and Automated Theorem Proving*. Springer-Verlag, 1996.
- [Frä19] Adolf Fränkel. ?? ??, 19??

- [Fre93] Gottlob Frege. *Grundgesetze der Arithmetik*, volume I. Verlag Hermann Pohle, 1893. Translated in part in [Fur64].
- [Fre03] Gottlob Frege. *Grundgesetze der Arithmetik*, volume II. Verlag Hermann Pohle, 1903. Translated in part in [Fur64].
- [Fur64] Montgomery Furth. *The Basic Laws of Arithmetic*. Berkeley: University of California Press, 1964. Translation of [Fre03].
- [Gen35] Gerhard Gentzen. Untersuchungen über das lo-

gische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in [Sza69].

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[GM93] Michael J. C. Gordon and Tom F. Melham, editors. *Introduction to HOL*. Cambridge University Press, 1993.

[Göd31] Kurt Gödel. Über formal unentscheidbare sätze

der principia mathematica und verwandter systeme. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

- [Har98] John Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.
- [Har99] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *Proceedings of the 12th TPHOLs*, volume 1690 of *LNCS*, pages 113–130. Springer-Verlag, 1999.

- [Har00] John Harrison. Formal verification of the ia/64 division algorithms. In Mark Aagaard and John Harrison, editors, *Proceedings of the 13th TPHOLs*, volume 1869 of *LNCS*, pages 233–251. Springer-Verlag, 2000.
- [HC68] George E. Hughes and Maxwell John Cresswell. *An Introduction to Modal Logic*. Muthuen and Co. Ltd, London, 1968.
- [Hen50] Henkin. ?? ??, 1950.
- [HHP93] Robert Harper, Furio Honsell, and Gordon D.

Plotkin. A framework for defining logics. *JACM*, 40(1):143–184, 1993.

[HHPW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philipp Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[Höl90] Steffen Hölldobler. Conditional equational theories and complete sets of transformations. *Theoretical Computer Science*, 75(1&2):85–110, 1990.

[HP93] G. Huet and G. Plotkin, editors. *Logical Environ-*

ments. Cambridge University Press, 1993.

- [HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ -Calculus*. Cambridge University Press, 1986.
- [Hu  ] Gerard Hu  t. ?? ??, ??
- [IEE85] The Institute of Electrical and Electronic Engineers, Inc. *IEEE. Standard for binary floating point arithmetic. ANSI/IEEE Standard 754-1985*, 1985.
- [Kim03] Angelika Kimmig. Red-black trees of slmnj.

Studienarbeit at Universität Freiburg, Download, 2003.

[Klo93] Jan Willem Klop. *Handbook of Logic in Computer Science*, chapter "Term Rewriting Systems". Oxford: Clarendon Press, 1993.

[KN03] Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 3(298):583–626, 2003.

[LP81] Harry R. Lewis and Christos H. Papadimitriou. *El-*

ements of the Theory of Computation. Prentice-Hall, 1981.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Min00] Grigori Mints. *A Short Introduction to Intuitionistic Logic*. Kluwer Academic/Plenum Publishers, 2000.

[Nip93] Tobias Nipkow. *Order-Sorted Polymorphism in*

Isabelle, pages 164–188. Cambridge University Press, 1993. In [HP93].

[Nip98] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics. *Formal Aspects of Computing*, 10(2):171–186, 1998.

[Nip02] Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.

[Nip03] Tobias Nipkow. Java bytecode verification. *Jour-*

nal of Automated Reasoning, 30(3-4):233–233, 2003.

[NN99] Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm \mathcal{W} in isabelle/hol. *Journal of Automated Reasoning*, 23(3-4):299–318, 1999.

[NP93] Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proceedings of the 20th ACM Symposium Principles of Programming Languages*, pages 409–418. ACM Press, 1993.

- [Pau89] Lawrence C Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.
- [Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.
- [Pau96] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.
- [Pau97] Lawrence C. Paulson. Generic automatic proof tools. In Robert Veroff, editor, *Automated Rea-*

soning and its Applications: Essays in Honor of Larry Wos, chapter 3. MIT Press, 1997.

[Pau04] Lawrence C. Paulson. *The Isabelle Reference Manual*. Computer Laboratory, University of Cambridge, March 2004.

[Pea18] Guiseppe Peano. ?? ??, 18??

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

- [PM68] Dag Prawitz and Per-Erik Malmnäs. A survey of some connections between classical, intuitionistic and minimal logic. In A. Schmidt and H. Schütte, editors, *Contributions to Mathematical Logic*, pages 215–229. North-Holland, 1968.
- [Pra65] Dag Prawitz. *Natural Deduction: A proof theoretical study*. Almqvist and Wiksell, 1965.
- [Pra71] Dag Prawitz. Ideas and results in proof theory. In Jens Erik Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 235–308. North-Holland, 1971.

- [SH84] Peter Schroeder-Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49(4):1284–1300, 1984.
- [Sza69] M. E. Szabo. *The Collected Papers of Gerhard Gentzen*. North-Holland, 1969.
- [Tho95a] Della Thompson, editor. *The Concise Oxford Dictionary*. Clarendon Press, 1995.
- [Tho95b] Simon Thompson. *Miranda: The Craft of Functional Programming*. Addison-Wesley, 1995.
- [Tho99] Simon Thompson. *Haskell: The Craft of Func-*

tional Programming. Addison-Wesley, 1999. Second Edition.

[vD80] Dirk van Dalen. *Logic and Structure*. Springer-Verlag, 1980. An introductory textbook on logic.

[Vel94] Daniel J. Velleman. *How to Prove It*. Cambridge University Press, 1994.

[vH67] Jean van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-193*. Harvard University Press, 1967. Contains translations of original works by David Hilbert.

- [vL16] Gottfried Wilhelm von Leibniz. ?? ??, 16??
- [WB89] Phillip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.
- [Wen99] Markus Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In Yves Bertot, Gilles Dowek, André Hirschowitz, and and Laurent Théry C. Paulin, editors, *Proceedings of TPHOLs*, volume 1690 of *LNCS*, pages 19–36. Springer-Verlag, 1999.

- [Win96] Glynn Winskel. *The Formal Semantics of Programming Languages – An Introduction*. MIT Press, 1996. 3rd ed.
- [WR25] Alfred N. Whitehead and Bertrand Russell. *Principia Mathematica*, volume 1. Cambridge University Press, 1925. 2nd edition.
- [Zer19] Ernst Zermelo. ?? ??, 19??