

Computer-Supported Modeling and Reasoning

Jan-Georg Smaus

WS06/07

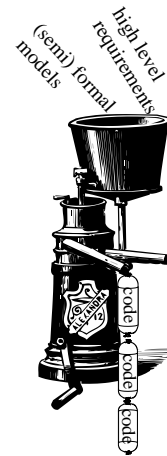
How to Use Lecture Notes

These lecture notes are generated from sources that were originally intended for hypermedia, as lecture slides or online course. Instead of hyperlinks you have footnotes and pointers to page numbers, indicated by ➡. The online versions of this material make heavy use of overlays. In this printout version, overlays are usually handled by putting the items in question side by side, separated by ⇨.

1 General Introduction

What this Course is about

- Mechanizing and using logic
 - program verification¹:
 - input**: theories, programs, properties
 - output**: correctness guarantees
 - Hilbert’s program²:
 - input**: arithmetic
 - output**: theorems
- Technically: mechanization and application³
- Generally: making logic come to life and useful as a general tool.



¹Verification 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”.

²In 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 can be proved or disproved?
2. Is mathematics consistent in the sense that no statement can be proved both true and false?
3. Is mathematics decidable in the sense that there exists 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

Overview: Four Parts

1. Logics⁴ (propositional, first-order, higher-order): appr. 6 units
2. Metalogics⁵ (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 theory⁶ (functional or imperative programming, or the specification language Z): appr. 2 units

Presentation roughly follows this structure.

³We will learn to make logic run on a computer by using the Isabelle system.

⁴The 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.

⁵A metalogic is a logic that allows us to express properties of another logic.

⁶Intuitively, 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

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, ...

that portion of the “world”.

Theories will be considered in more detail later (➔ p.115).

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 (➔ p.623)?

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 logic⁷ 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 that it makes no sense to follow just a little bit. It is hard in the beginning but the rewards are large.

⁷We 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].

What's Happening in Freiburg?

Harald Hiss, Stefan Maus, and Stefan Wölfl work with Isabelle here at Freiburg:

- There is a trend to use XML (a generalisation of HTML) for database applications. However, this gives rise to possible inconsistencies. Harald uses Isabelle to prove formally that such inconsistencies cannot occur.
- There are various formal theories that allow to reason about the relationship of objects in space and time. Stefan uses Isabelle for proving consequences of such theories, dependencies between theories etc.

Also, David Basin occasionally seeks PhD students.

2 Propositional Logic

2.1 Propositional Logic: Overview

- System for formalizing certain valid patterns of reasoning
- Expressions built by combining “atomic propositions” using **not**, **if...then...**, **and**, **or**, etc.
- Validity⁸ 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**. \Rightarrow
$$\frac{A \rightarrow B \quad A}{B}$$

- What about⁹

⁸**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.

⁹

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”.

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

More Examples (Which are Valid?)¹⁰

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.

10

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

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).

Further reading: [vD80], [Tho91, chapter 1].

More Formal Examples

Formalization allows us to “turn the crank”¹¹.

Phrases like “from . . . it follows” or “therefore” are formalized¹² 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 system¹³ in the style of natural deduction.

¹¹By 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.

¹²At 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 (→ p.296).

¹³A 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 (→ p.45) rules.

2.2 Formalizing Propositional Logic

- We must formalize
 1. Language¹⁴ and semantics (\rightarrow p.22)
 2. Deductive system
- Here we will focus on formalizing the deductive machinery and say little about metatheorems¹⁵ (soundness and completeness¹⁶).
- For labs we will carry out proofs using the Isabelle System. Isabelle supports a Natural Deduction deductive system.

¹⁴By 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 (\rightarrow p.6).

¹⁵A 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.

¹⁶A proof system is sound if only valid (\rightarrow p.12) propositions can be derived in it.

A proof system is complete if all valid (\rightarrow p.12) propositions can be derived in it.

2.3 Propositional Logic: Language and Semantics

Propositions are built from a collection of (propositional) variables¹⁷ and closed under disjunction, conjunction, implication, ...

¹⁷In 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 (➔ p.64).

Propositional Logic: Language (2)

More formally: Let a set V of variables be given. L_P , the¹⁸ language of propositional logic, is the smallest set¹⁹ where:

- X in L_P if X in V .

¹⁸Strictly 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 (\rightarrow p.69), we will be more fussy about this point.

¹⁹The 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

- \perp ²⁰ in L_P .
- $(A \wedge$ ²¹ $B)$ 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 .

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 ... and ψ_n are in L_P , then some formula built from ψ_1 and ... 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].

²⁰

The symbol \perp stands for “false”.

²¹The connectives are called conjunction (\wedge), disjunction (\vee), implication (\rightarrow) and negation (\neg).

The connectives $\wedge, \vee, \rightarrow$ are binary since they connect two

- $((\neg A)^{22}$ in L_P if A in L_P .)

The elements of L_P are called (propositional) formulas²⁴.

We omit unnecessary brackets²⁵.

formulas, the connective \neg is unary (most of the time, one only uses the word connective for binary connective).

²²“Officially”, negation does not exist in our language and proof system. Negation is only used as a shorthand, or syntactic sugar²³, 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.

²⁴In 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 (\rightarrow p.69) learn about another syntactic category, that of terms.

²⁵To save brackets, we use standard associativity and precedences. All binary connectives (\rightarrow p.20) are right-

Propositional Logic: Semantics

An assignment is a function $\mathcal{A} : V \rightarrow \{0, 1\}$. We say that \mathcal{A} assigns a truth value to each propositional variable. We identify 1 with *True* and 0 with *False*.

\mathcal{A} is lifted (=extended) to formulas in L_P as follows ...

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

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

Propositional Logic: Semantics (2)

$$\begin{aligned}\mathcal{A}(\perp) &= 0 \\ \mathcal{A}(\neg\phi) &= \begin{cases} 1 & \text{if } \mathcal{A}(\phi) = 0 \\ 0 & \text{otherwise} \end{cases} \\ \mathcal{A}(\phi \wedge \psi) &= \begin{cases} 1 & \text{if } \mathcal{A}(\phi) = 1 \text{ and } \mathcal{A}(\psi) = 1 \\ 0 & \text{otherwise} \end{cases} \\ \mathcal{A}(\phi \vee \psi) &= \begin{cases} 1 & \text{if } \mathcal{A}(\phi) = 1 \text{ or } \mathcal{A}(\psi) = 1 \\ 0 & \text{otherwise} \end{cases} \\ \mathcal{A}(\phi \rightarrow \psi) &= \begin{cases} 1 & \text{if } \mathcal{A}(\phi) = 0 \text{ or}^{26} \mathcal{A}(\psi) = 1 \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

Propositional Logic: Semantics (3)

If $\mathcal{A}(\phi) = 1$, we write $\mathcal{A} \models \phi$.

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

The semantics will be generalised later (\rightarrow p.73).

2.4 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

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

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. $A \rightarrow (B \rightarrow C), A, B \vdash C$ ²⁷.

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

²⁷For 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 (➔ p.48).

Natural Deduction: an Abstract Example²⁸

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

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

How do you read these rules?²⁹

How about this one?³⁰

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

²⁸Natural deduction is not just about propositional logic! We explain here the general principles (\rightarrow p.16) 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.

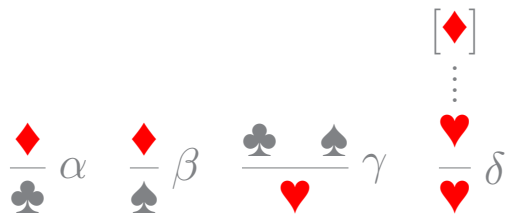
²⁹The first rule reads: if at some root of a tree in the forest you have constructed so far, there is a \diamondsuit , then you are allowed to draw a line underneath that \diamondsuit and write \clubsuit underneath that line.

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

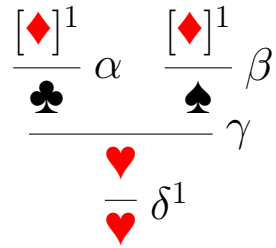
³⁰The last rule reads: if at some root of a tree in the forest

Proof of ♥

The rules:



The proof:



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.

We make³¹ an assumption. The assumption is now open³².

We apply α .

Similarly with β .

We apply γ .

We apply δ , discharging two occurrences of \blacklozenge . 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 (\rightarrow p.29) (all discharged).

³¹In 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.

³²For example, all assumptions in

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

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

2.5 Deductive System: Rules of Propositional Logic

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

Some rules introduce³³, others eliminate connectives.

³³It is typical that the basic (\rightarrow p.16) 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 (\rightarrow p.16) rules, we will later see that there are also derived rules.

Rules of Propositional Logic (\rightarrow p.16): Conjunction

- Rules of two kinds: introduce (\rightarrow p.30) and eliminate (\rightarrow p.30) 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 schematic³⁴.
- Why valid³⁵? If all assumptions are true, then so is conclusion

$$\mathcal{A} \models A \wedge B \ (\rightarrow \text{p.24}) \text{ iff } \mathcal{A} \models A \text{ and } \mathcal{A} \models B$$

³⁴The 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 (\rightarrow p.51).

³⁵A rule is valid if for any assignment (\rightarrow p.22) under which the assumptions of the formula are true, the conclusion is true as well.

This is consistent with the earlier intuitive explanation (\rightarrow p.12) 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

Example Derivation with Conjunction

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

Can we prove anything with just these three rules?³⁶

later (\rightarrow p.76), 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.

³⁶All 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 (\rightarrow p.26) that a proof is a derivation with no open assumptions.

Consequently, the answer is no.

Rules of Propositional Logic: Implication

- Rules

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

- $\rightarrow\text{-}E$ is also called modus ponens.
- $\rightarrow\text{-}I$ 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 $P \rightarrow P$.

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

Do you find this strange?³⁷

³⁷When we make the assumption P , we obtain a forest (\rightarrow p.27) 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 (\rightarrow p.27).

Examples with Conjunction and Implication

1. $A \rightarrow B \rightarrow A$ ³⁸
2. $A \wedge (B \wedge C) \rightarrow A \wedge C$ ³⁹

The rule(s):	The proof:
³⁸ $\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B} \rightarrow\text{-}I$	$\frac{\frac{[A]^1}{B \rightarrow A} \rightarrow\text{-}I}{A \rightarrow B \rightarrow A} \rightarrow\text{-}I^1$
<p>The rules:</p> $\frac{A \quad B}{A \wedge B} \wedge\text{-}I$ $\frac{A \wedge B}{A} \wedge\text{-}EL$ ³⁹ $\frac{A \wedge B}{B} \wedge\text{-}ER$ $\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B} \rightarrow\text{-}I$	<p>The proof:</p> $\frac{\frac{[A \wedge (B \wedge C)]^2}{B \wedge C} \wedge\text{-}ER}{\frac{A}{A \wedge C} \wedge\text{-}I} \wedge\text{-}ER$ $\frac{\frac{[A \wedge (B \wedge C)]^2}{A} \wedge\text{-}EL \quad \frac{B \wedge C}{C} \wedge\text{-}ER}{(A \wedge (B \wedge C)) \rightarrow (A \wedge C)} \rightarrow\text{-}I^2$

3. $(A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$ ⁴⁰

Are these object or metavariables here?⁴¹

The rules:	The proof:
$ \begin{array}{c} [A] \\ \vdots \\ B \\ \hline A \rightarrow B \rightarrow -I \\ \hline A \rightarrow B \quad A \rightarrow -E \\ \hline B \end{array} $	$ \begin{array}{c} [(A \rightarrow B \rightarrow C)]^3 \quad [A]^5 \rightarrow -E \quad [(A \rightarrow B)]^4 \quad [A]^5 \rightarrow -E \\ \hline B \rightarrow C \quad B \rightarrow -E \\ \hline C \rightarrow -E \\ \hline A \rightarrow C \rightarrow -I^5 \\ \hline (A \rightarrow B) \rightarrow A \rightarrow C \rightarrow -I^4 \\ \hline (A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C \rightarrow -I^3 \end{array} $

⁴¹In 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 .

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

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

- 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

$$\frac{\perp \quad (\rightarrow \text{ p.20})}{A} \perp\text{-}E$$

No introduction rule!⁴²

- Negation: define (\rightarrow p.21) $\neg A$ as $A \rightarrow \perp$. Rules for \neg just special cases⁴³ of rules for \rightarrow . Convenient to have

⁴²The 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.

⁴³The rule

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

is simply an instance of $\rightarrow\text{-}E$ (\rightarrow p.33) (since $\neg A$ is shorthand for $A \rightarrow \perp$).

Likewise, the rule

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

$$\frac{\neg A \quad A}{B} \neg\text{-}E^{44} \quad \text{derived by } (\rightarrow \text{ p.40}) \quad \frac{\frac{\neg A \quad A}{\perp} \rightarrow\text{-}E}{B} \perp\text{-}E$$

is simply an instance of $\rightarrow\text{-}I$ (\rightarrow p.33). Therefore, we will not introduce these as special rules. But there is a special rule $\neg\text{-}E$ (\rightarrow p.40).

⁴⁴For 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 (\rightarrow p.45).

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

Intuitionistic versus Classical Logic

- Peirce's Law: $((A \rightarrow B) \rightarrow A) \rightarrow A$.
Is this valid⁴⁵? Provable⁴⁶?

⁴⁵Yes, 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>

⁴⁶In the proof system given so far (\rightarrow p.42), 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.

- It is provable in classical logic⁴⁷, obtained by adding

$$A \vee \neg A^{48} \text{ or } \frac{\begin{array}{c} [\neg A] \\ \vdots \\ \perp \end{array}}{A} RAA_{49} \text{ or } \frac{\begin{array}{c} [\neg A] \\ \vdots \\ A \end{array}}{A} \textit{classical}_{50}.$$

⁴⁷The 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] [Tho91, chapter 3]. 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

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

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

We still don't know how to choose a and b so that a^b is rational. Hence the proof is non-constructive (\rightarrow p.42).

in intuitionistic logic.

⁴⁸ $A \vee \neg A$ is called axiom of the excluded middle.

⁴⁹The rule

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

is called reduction ad absurdum.

⁵⁰The rule

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

corresponds to the formulation is Isabelle.

Overview of Rules

$$\begin{array}{c}
 \frac{A \quad B}{A \wedge B} \wedge\text{-}I \qquad \frac{A \wedge B}{A} \wedge\text{-}EL \qquad \frac{A \wedge B}{B} \wedge\text{-}ER \\
 \\
 \frac{A}{A \vee B} \vee\text{-}IL \qquad \frac{B}{A \vee B} \vee\text{-}IR \qquad \frac{A \vee B \quad \begin{array}{c} [A] \\ \vdots \\ C \end{array} \quad \begin{array}{c} [B] \\ \vdots \\ C \end{array}}{C} \vee\text{-}E \\
 \\
 \frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B} \rightarrow\text{-}I \qquad \frac{A \rightarrow B \quad A}{B} \rightarrow\text{-}E \qquad \frac{\perp \quad (\rightarrow \text{ p.20})}{A} \perp\text{-}E
 \end{array}$$

2.6 Deductive System: Derived Rules

Using the basic (\rightarrow p.16) rules, we can derive new rules.

Example: Resolution rule.

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

It looks like this.

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

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

So now S .

$\neg S$ and S allow us to apply \rightarrow - E (\rightarrow p.21).

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

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

A Variation of Natural Deduction: Boxes

We have seen just one deductive system.

One variation of natural deduction is the following: A derivation is not a tree, but a sequence of numbered lines. Instead of subtrees relying on open assumptions, a subderivation relying on an assumption is enclosed in a box.

You find this explained in [HR04].

2.7 Alternative Deductive System Using Sequent Notation

One can base the deductive system around the derivability judgement⁵¹, i.e., reason about $\Gamma \vdash A$ where $\Gamma \equiv A_1, \dots, A_n$ instead of individual formulae.

⁵¹An object like $A \rightarrow (B \rightarrow C), A, B \vdash C$ is called a derivability judgement. We explained it earlier (\rightarrow p.26) 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 (\rightarrow p.267)). 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

Sequent Rules (for \rightarrow / \wedge Fragment)

Rules for assumptions⁵² and weakening⁵³:

$$\Gamma \vdash A^{54} \quad (\text{where } A \in \Gamma) \quad \frac{\Gamma \vdash B}{A, \Gamma \vdash B} \text{weaken}$$

Rules for \wedge and \rightarrow :

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

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

call natural deduction using a sequent notation.

⁵²The special rule for assumptions takes the role in this sequent style (\rightarrow p.48) notation that the process of making and discharging assumptions had in natural deduction based on trees (\rightarrow p.25).

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.

⁵³The 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 (\rightarrow p.263).

⁵⁴An axiom is a rule without premises. We call a rule with premises proper.

More rules can be derived⁵⁵.

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 (\rightarrow p.44) for propositional logic contain no axioms. In the sequent style (\rightarrow p.48) 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.

⁵⁵ As an example, consider

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

Example: Refinement Style with Metavariables

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

This rule can be derived as follows:

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

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.

The topmost connective of the formula is \rightarrow , so the best rule⁵⁶ to choose is \rightarrow -*I*.

The topmost connective of the formula is \wedge , so the best rule (\rightarrow p.52) to choose is \wedge -*I*.

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

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

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

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

⁵⁶In 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 (\rightarrow p.260).

Comments about Refinement

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

- Refinement style means we work from goals to axioms⁵⁷
- metavariables used to delay commitments

Isabelle allows other refinements⁵⁸/alternatives too (see labs).

⁵⁷As you saw in our animation, we worked from the root of the tree to the leaves.

⁵⁸One 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 (\rightarrow p.236).

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!

3 Natural Deduction: Review

Overview

- Short review: ND Systems and proofs (➔ p.55)
- First-Order Logic (➔ p.62)
 - Overview (➔ p.62)
 - Syntax (➔ p.69)
 - Semantics (➔ p.73)
 - Deduction (➔ p.80), some derived rules, and examples

How Are ND Proofs Built?

ND proofs⁵⁹ build derivations under (possibly temporary) assumptions.

⁵⁹ND stands for Natural Deduction. It was explained in the previous lecture (➔ p.25).

ND: Example for \rightarrow / \wedge Fragment

Rules:

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

$$\frac{A \wedge B}{B} \wedge\text{-}ER \quad \frac{[A] \quad \vdots \quad B}{A \rightarrow B} \rightarrow\text{-}I$$

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

Proof:

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

$$\frac{B \wedge A}{A \wedge B \rightarrow B \wedge A} \rightarrow\text{-}I^1$$

Alternative Formalization Using Sequents⁶⁰

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

$$\begin{array}{c} \Gamma \vdash A \quad (\text{where } A \in \Gamma) \\ \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge\text{-}I \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge\text{-}EL \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge\text{-}ER \\ \frac{A, \Gamma \vdash B}{\Gamma \vdash A \rightarrow B} \rightarrow\text{-}I \quad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \rightarrow\text{-}E \end{array}$$

Two representations (\rightarrow p.58) equivalent. Sequent notation seems simpler in practice⁶¹.

⁶⁰The judgement $(\Gamma \vdash \phi)$ means that we can derive ϕ from the assumptions in Γ using certain rules. As, explained in the previous lecture (\rightarrow p.48), one can make such judgements the central objects of the deductive system.

⁶¹In particular, the sequent style notation is more amenable to automation, and thus it is closer to what happens in Isabelle.

Example: Refinement Style with Metavariables

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

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 (\rightarrow p.51).

Comments about Refinement

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

- Refinement style means we work from goals to axioms (➡ p.53)
- Metavariables used to delay commitments

Isabelle allows other refinements (➡ p.53)/alternatives too (see labs).

4 First-Order Logic

4.1 First-Order Logic: Overview

In propositional logic, formulae are Boolean⁶² combinations of propositions. This will remain important for modeling simple patterns of reasoning (\rightarrow p.12).

An atomic (\rightarrow p.12) proposition is just a letter (variable (\rightarrow p.18)). All one can say about it is that it is true or false. E.g. it is meaningless to say “ A and B state something similar”. Also, infinity plays no role.

⁶²The 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*.

First-Order Logic: the Essence

In first-order logic, an atom(ic proposition) says that “things” have certain “properties”⁶³. Infinitely many “things” can be denoted, hence infinitely many atoms generated and distinguished. Comparisons of atoms become meaningful: “Tim is a boy” and “Carl is a boy” state something similar.

Example reasoning: “Tim is a boy”; “boys don’t cry”; hence “Tim doesn’t cry”.

Further reading: [vD80], [Tho91, chapter 1].

⁶³In 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 (\rightarrow p.24) and for a set containing n variables, there are only finitely many (to be precise 2^{2^n}) different propositional formulae.

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 (\rightarrow p.18) where variables stand for propositions.

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

Propositional connectives (\rightarrow p.20) 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 ”.

But the sense in which $p(x)$, $d(y, x)$, $d(z, x)$ state something similar is lost. What it means to be divisible or to be a prime cannot be expressed.

Functions: Intuition

- A constant stands for a “fixed thing”⁶⁴ in a domain⁶⁵.
- More generally, a function of arity (\rightarrow p.69) n expresses an n -ary operation over some domain, e.g.

Function	arity	expresses ...
0	nullary	number “0”
s	unary	successor in \mathbb{N} ⁶⁶
$+$	binary	function plus in \mathbb{N}

Note special notations⁶⁷: infix, prefix, etc.

⁶⁴As opposed to a variable which also stands for a “thing”. This distinction will become clear soon (\rightarrow p.68).

⁶⁵For 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.

⁶⁶ \mathbb{N} denotes the natural numbers.

⁶⁷So 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.

Quantifiers: Intuition

- A variable stands for “some⁶⁸ 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?⁶⁹

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

$$x < y \rightarrow \exists z. x < z \wedge z < y \quad \text{true for any dense (→ p.119) order}$$

$$\exists x. x \neq 0 \quad \text{true for domains with more than one element}$$

$$(\forall x. p(x, x)) \rightarrow p(a, a) \quad \text{valid}$$

⁶⁸Just 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 $x \dots$ ” or “there exists x such that \dots ”.

⁶⁹Intuitively, satisfiable means “can be made true” and valid means “always true”.

More formally, this will be defined later (→ p.76).

4.2 First-Order Logic: Syntax

- Two syntactic categories: terms⁷⁰ and formulae
- A first-order language⁷¹ is characterized by giving a finite collection of function symbols \mathcal{F} and predicate symbols \mathcal{P} as well as a set Var of variables.
- Sometimes write f^i (or p^i) to indicate that function symbol f (or predicate symbol p) has arity $i \in \mathbb{N}$ (\rightarrow p.67).
- One often calls the pair $\langle \mathcal{F}, \mathcal{P} \rangle$ a signature.

⁷⁰We have already learned about the syntactic category of formulae last lecture (\rightarrow p.21).

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!

⁷¹There 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.

Terms in First-Order Logic

$Term$ ⁷², the set of terms, is the smallest (\rightarrow p.19) set where

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

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.

⁷² $Term$ and $Form$ together make up a first-order language. Note that strictly speaking, $Term$ and $Form$ depend on the signature (\rightarrow p.69), but we always assume that the signature is clear from the context.

Formulae in First-Order Logic

Form (\rightarrow p.70), the set of formulae, is the smallest (\rightarrow p.19) 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. $((\rightarrow \text{p.21})\phi \circ \psi) (\rightarrow \text{p.21}) \in Form$ 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 scoping⁷³.

⁷³We 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 (\rightarrow p.21) and associativity in propositional logic.

Variable Occurrences

- All occurrences of a variable in a formula⁷⁴ 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:

$$\begin{aligned} & (q(x) \vee \exists x. \forall y. p(f(x), z) \wedge q(y)) \vee \forall x. r(x, z, g(x)) \mapsto (q(x) \vee \\ & \exists x. \forall y. p(f(\textcolor{red}{x}), z) \wedge q(\textcolor{red}{y})) \vee \forall x. r(\textcolor{red}{x}, z, g(\textcolor{red}{x})) \mapsto (q(\textcolor{green}{x}) \vee \\ & \exists x. \forall y. p(f(\textcolor{red}{x}), \textcolor{green}{z}) \wedge q(\textcolor{red}{y})) \vee \forall x. r(\textcolor{red}{x}, \textcolor{green}{z}, g(\textcolor{red}{x})) \mapsto (q(\textcolor{green}{x}) \vee \\ & \exists \textcolor{blue}{x}. \forall \textcolor{blue}{y}. p(f(\textcolor{red}{x}), \textcolor{green}{z}) \wedge q(\textcolor{red}{y})) \vee \forall \textcolor{blue}{x}. r(\textcolor{red}{x}, \textcolor{green}{z}, g(\textcolor{red}{x})) \mapsto \end{aligned}$$

Which are **bound**? Which are **free**? Which are **binding**?

There will be an exercise.

⁷⁴All 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 (\rightarrow p.70) and formulae (\rightarrow p.71).

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 ($\circ \in \{\wedge, \vee, \rightarrow\}$);

4.3 First-Order Logic: Semantics

A structure⁷⁵ 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.

-
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

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

⁷⁵As 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 (\rightarrow p.67).

As you saw, a structure gives a meaning to functions,

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. $\mathcal{A}(f(t_1, \dots, t_n)) = f^{\mathcal{A}}(\mathcal{A}(t_1), \dots, \mathcal{A}(t_n))$.

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 (\rightarrow p.70), we assume that the signature (\rightarrow p.69) 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].

The Value of Formulae

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

$$\begin{aligned}\mathcal{A}(p(t_1, \dots, t_n)) &= \begin{cases} 1 & \text{if } (\mathcal{A}(t_1), \dots, \mathcal{A}(t_n)) \in p^{\mathcal{A}} \\ 0 & \text{otherwise} \end{cases} \\ \mathcal{A}(\forall x. \phi) &= \begin{cases} 1 & \text{if for all } u \in U_{\mathcal{A}}, \mathcal{A}_{[x/u]}^{76}(\phi) = 1 \\ 0 & \text{otherwise} \end{cases} \\ \mathcal{A}(\exists x. \phi) &= \begin{cases} 1 & \text{if for some } u \in U_{\mathcal{A}}, \mathcal{A}_{[x/u]}(\phi) = 1 \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

Rest as for propositional logic (\rightarrow p.23).

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 structure⁷⁷ 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.⁷⁸

⁷⁷A structure (\rightarrow p.73) is suitable for ϕ if it defines meanings for the signature (\rightarrow p.69) 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 (\rightarrow p.74), we always assume that structures are suitable.

⁷⁸If 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 (\rightarrow p.73), 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 (\rightarrow p.74) or formula (\rightarrow p.75). In the alternative terminology, we write $\mathcal{I}(A)(\cdot)$ instead. This makes sense since

An Example

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

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 an \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.

A model⁷⁹:

Not a model⁸¹:

$$\begin{array}{ll} U_{\mathcal{A}} = \mathbb{N} \ (\rightarrow \text{p.67}) & U_{\mathcal{A}} = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\} \\ p^{\mathcal{A}} = \{(m, n) \mid m < {}^{80}n\} & p^{\mathcal{A}} = \{(\mathbf{a}, \mathbf{b}), (\mathbf{a}, \mathbf{c})\} \\ s^{\mathcal{A}}(x) = x + 1 \ (\rightarrow \text{p.78}) & s^{\mathcal{A}} = \text{“the identity function”} \end{array}$$

⁷⁹It is true that for all numbers n , n is less than $n + 1$.

⁸⁰In logic, we insist on the distinction between syntax and semantics. In particular, we set up the formalism so that the syntax is fixed first (\rightarrow p.69) and then the semantics (\rightarrow p.73), 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 $\mathbf{0}$ 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

4.4 Towards a Deductive System

In natural language, quantifiers are often implicit⁸²: all males don't cry.

Some phrases in natural language proofs have the flavor of introduction rules (\rightarrow p.30).

Take “boys are males” and “males don't cry” implies “boys don't cry”: assume an arbitrary boy x ; then x is a male; hence x doesn't cry; hence “ x is a boy” implies “ x doesn't cry” (\rightarrow -I); since x was arbitrary, we can say this for all x . (\forall -I). See later (\rightarrow p.87).

Existential statements are proven by giving a witness.

we write $m < n$, we assume that $m, n \in \mathbb{N}$ (\rightarrow p.76), in this context.

⁸¹The identity function maps every object to itself.

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

⁸²In the statement

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

the \forall -quantifier is implicit. It should be

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

4.5 First-Order Logic: Deductive System

First-order logic is a generalization of propositional logic. All the rules of propositional logic (\rightarrow p.30) are “inherited”⁸³. But we must introduce rules for the quantifiers.

⁸³First-order logic inherits all the rules of propositional logic (\rightarrow p.30). Note however that the metavariables (\rightarrow p.31) in the rules now range over first-order formulae.

Universal Quantification (\forall): Rules

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

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

Note that rules are schematic⁸⁴.

⁸⁴Similarly as in the previous lecture (\rightarrow p.31), 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 $\forall\text{-}E$, $P(t)$ stands for a formula where all occurrences of x are replaced by t (\rightarrow p.84).

Universal Quantification: Side Condition

What does arbitrary mean? Consider the following “proof”

$$\begin{array}{c}
 \dfrac{[x = 0]^1}{\forall x. x = 0} \text{ } \color{red}{\forall\text{-}I} \\
 \dfrac{\dfrac{x = 0 \rightarrow \forall x. x = 0}{\forall x. (x = 0 \rightarrow \forall x. x = 0)} \rightarrow\text{-}I^1}{\forall x. (x = 0 \rightarrow \forall x. x = 0)} \forall\text{-}I \\
 \dfrac{\dfrac{0 = 0 \rightarrow \forall x. x = 0}{\forall x. x = 0} \forall\text{-}E \quad \dfrac{}{0 = 0} \text{ } \text{refl}^{85}}{\forall x. x = 0} \rightarrow\text{-}E
 \end{array}$$

Formal meaning of side condition (\rightarrow p.81): x not free in any open assumption on which $P(x)$ depends. Violated!⁸⁶

⁸⁵When one has a predicate symbol $=$, it is usual to have a rule that says that $=$ is reflexive (\rightarrow p.107).

Don't worry about it at this stage, just take it that we have such a rule. We will look at this later (\rightarrow p.103).

⁸⁶The side condition is violated in the proof since in the first $\forall\text{-}I$ step, x does occur free in $x = 0$.

Another Proof? (1)

Is the following a proof? Is the conclusion valid?

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

Conclusion is not valid.

The formula is false when $U_{\mathcal{A}}$ has at least 2 elements.⁸⁷

Proof is incorrect.

Reason: Substitution⁸⁸ must avoid capturing⁸⁹ variables.

Replacing x with y in \forall -E is illegal because y is bound (\rightarrow p.72) in $\neg\forall y. y = y$. This detail concerns substitution (and re-naming of bound (\rightarrow p.72) variables), not \forall -E. Exercise

⁸⁷Here we assume that the predicate symbol $=$ is interpreted by \mathcal{A} (\rightarrow p.73) 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 $\mathcal{A}_{[x/u]}$ (\rightarrow p.75)($\neg\forall y. x = y$) = 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, $\mathcal{A}((\forall x. \neg\forall y. x = y) \rightarrow \neg\forall y. y = y) = 0$.

⁸⁸The 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

⁸⁹A substitution (\rightarrow p.84) (replacement of a variable by a term) must not replace bound (\rightarrow p.72) occurrences of variables, and if we replace x with t in an expression ϕ , then this replacement should not turn free (\rightarrow p.72) occurrences of variables in t into bound (\rightarrow p.72) occurrences in ϕ . It is possible to avoid this by renaming variables.

This is part of the standard definition of a substitution (\rightarrow p.84). 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 (\rightarrow p.159), in a different context.

Another Proof? (2)

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

Yes (check side conditions⁹⁰ of $\forall\text{-}I$).

⁹⁰In both cases, x does not occur free (\rightarrow p.72) in $\forall x. A(x) \wedge B(x)$, which is the open assumption (\rightarrow p.82) on which $A(x)$, respectively $B(x)$, depends.

Boys Don't Cry

Let $\phi \equiv (\forall x. b(x) \rightarrow m(x)) \wedge (\forall x. m(x) \rightarrow \neg c(x))$.

$$\begin{array}{c}
 \frac{[\phi]^1}{\forall x. m(x) \rightarrow \neg c(x)} \wedge\text{-}ER \quad \frac{\frac{[\phi]^1}{\forall x. b(x) \rightarrow m(x)} \wedge\text{-}EL}{b(x) \rightarrow m(x)} \forall\text{-}E \quad [b(x)]^2}{\frac{m(x) \rightarrow \neg c(x)}{\forall\text{-}E} \quad \frac{b(x) \rightarrow m(x)}{\forall\text{-}E} \quad \frac{[b(x)]^2}{\rightarrow\text{-}E}} \rightarrow\text{-}E \\
 \frac{\neg c(x)}{b(x) \rightarrow \neg c(x)} \rightarrow\text{-}I^2 \\
 \frac{b(x) \rightarrow \neg c(x)}{\forall x. b(x) \rightarrow \neg c(x)} \forall\text{-}I \\
 \frac{\forall x. b(x) \rightarrow \neg c(x)}{\phi \rightarrow (\forall x. b(x) \rightarrow \neg c(x))} \rightarrow\text{-}I^1
 \end{array}$$

Aside: $A \leftrightarrow B$

Define⁹¹ $A \leftrightarrow B$ as $A \rightarrow B \wedge B \rightarrow A$.

The following rule can be derived (\rightarrow p.45) (in propositional logic, actually):

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

You could do this as an exercise!

⁹¹By 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 (\rightarrow p.21).

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 (\rightarrow p.72) in A .

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

Side Conditions and Proof Boxes

We mentioned previously (\rightarrow p.47) a style of writing derivations where subderivations based on temporary assumptions are enclosed in boxes.

These boxes are also handy for doing derivations in first-order logic, since one can use the very clear formulation: a variable occurs inside or outside of a box. See [HR04].

Existential Quantification

- We could define⁹² $\exists x. A$ as $\neg \forall x. \neg A$.
- Equivalence follows from our definition of semantics (\rightarrow p.75).

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

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

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

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

⁹²By 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 (\rightarrow p.21).

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 can⁹³ use definition $\exists x. A \equiv \neg \forall x. \neg A$ and the given rules for \forall to derive (\rightarrow p.45) ND (\rightarrow p.57) proof rules.
- Alternatively, we can give rules as part of the deduction system and prove equivalence as a lemma, instead of by definition.

We will do the first here. The Isabelle formalisation follows the second approach.

93

- We can use definition $\exists x. A \equiv \neg \forall x. \neg A$ and the given rules for \forall to derive (\rightarrow p.45) ND (\rightarrow p.57) proof rules. In this case, the soundness (\rightarrow p.17) 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 (\rightarrow p.17) must be proven by hand (however, proving rules sound is an aspect we neglect (\rightarrow p.10) 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 (\rightarrow p.17).

\exists -I as a Derived Rule

$$\begin{array}{c|l}
 \text{The rule:} & \\
 \frac{A(t)}{\exists x. A(x)} \exists\text{-I} & \frac{\frac{\frac{[\forall x. \neg A(x)]^1}{\neg A(t)} \forall\text{-E} \quad A(t)}{\perp} \rightarrow\text{-E}}{\exists x. A(x) \neg \forall x. \neg A(x)} \rightarrow\text{-I}^1
 \end{array}$$

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

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

We aim for applying \rightarrow -I in the last step (recall \neg -definition (\rightarrow p.21)).

We apply \forall -E.

Making assumption $A(t)$ allows us to use \rightarrow -E (recall \neg -definition (\rightarrow p.21)).

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

\exists -E as a Derived Rule

The rule:

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

$$\begin{array}{c}
 \begin{array}{c}
 [A(x)]^2 \\
 \vdots \\
 B \\
 \hline
 [\neg B]^1
 \end{array}
 \end{array}
 \begin{array}{c}
 \vdots \\
 B \\
 \hline
 \rightarrow-E
 \end{array}$$

$$\begin{array}{c}
 \vdots \\
 B \\
 \hline
 \perp
 \end{array}
 \begin{array}{c}
 \vdots \\
 B \\
 \hline
 \rightarrow-I^2
 \end{array}$$

$$\begin{array}{c}
 \vdots \\
 B \\
 \hline
 \neg A(x)
 \end{array}
 \begin{array}{c}
 \vdots \\
 B \\
 \hline
 \forall-I
 \end{array}$$

$$\begin{array}{c}
 \vdots \\
 B \\
 \hline
 \exists x. A(x) \neg \forall x. \neg A(x)
 \end{array}
 \begin{array}{c}
 \vdots \\
 B \\
 \hline
 \forall x. \neg A(x)
 \end{array}
 \begin{array}{c}
 \vdots \\
 B \\
 \hline
 \rightarrow-E
 \end{array}$$

$$\begin{array}{c}
 \vdots \\
 B \\
 \hline
 \perp
 \end{array}
 \begin{array}{c}
 \vdots \\
 B \\
 \hline
 RAA^1
 \end{array}$$

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

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

We assume a hypothetical derivation⁹⁴.

We make an additional assumption and apply \rightarrow -E (recall \neg -definition (\rightarrow p.21))

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

At this step, the side condition from \forall -I applies. \exists -E will inherit it!⁹⁵

We apply \rightarrow -E.

We are done. Note that this proof uses classical⁹⁶ reasoning.

⁹⁴We are constructing here a “schematic fragment” of a derivation tree. Within this construction, we 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 (\rightarrow p.27) involving discharging.

⁹⁵ \exists -E will inherit the side condition from \forall -I. Hence, the

⁹⁶Defining (\rightarrow p.91) $\exists x. A$ as $\neg\forall x. \neg A$ is only sensible in classical reasoning (\rightarrow p.42), since the derivation of the rule \exists - E requires the RAA (\rightarrow p.42) rule.

Example Derivation Using \exists -E

Assumption: x does not occur free in B (\rightarrow p.89)

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

4.6 Conclusion on FOL

- Propositional logic is good for modeling simple patterns of reasoning (\rightarrow p.12) like “if ... then ... else”.
- In first-order logic, one has “things” and relations on / properties of “things”. Quantify over “things”. Powerful⁹⁷!
- Some people advocate intuitionistic, relevant, and other

⁹⁷In 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 (\rightarrow p.102), 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 ἀνήρ (anér = human male), is used.

The example is a so-called syllogism 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 fre-

“deviant” logics⁹⁸.

quently 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.

⁹⁸There are still controversies about what the best logic is for reasoning about “things” and properties/relations, and scope (quantification). Some argue for intuitionistic (\rightarrow p.42), 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

- Limitation: cannot quantify over predicates⁹⁹.
- “A” world or “the” world is modeled in first-order logic using so-called first-order theories. This will be studied next lecture (➔ p.115).

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 .

⁹⁹The 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:

5 First-Order Logic with Equality

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 .

Overview

Last lecture: first-order logic (\rightarrow p.62).

This lecture:

- first-order logic with equality (\rightarrow p.103) and first-order theories (\rightarrow p.115);
- set-theoretic reasoning (\rightarrow p.127).

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 one¹⁰⁰.

Speak of first-order logic with equality rather than adding equality as “just another predicate”.

100

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

Recall (\rightarrow p.69) 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 (\rightarrow p.69), and whose meaning must be defined “by the user” by giving a structure (\rightarrow p.73).

Above we say “mathematical” instead of “non-logical” because we assume that mathematics is our domain of discourse, so that the signature (\rightarrow p.69) 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

Syntax and Semantics

Syntax: $=$ is a binary infix predicate.

$t_1 = t_2 \in Form$ (\rightarrow p.71) if $t_1, t_2 \in Term$ (\rightarrow p.70).

Semantics : recall a structure (\rightarrow p.73) 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 “=”¹⁰¹ here!

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.

¹⁰¹

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

The first \equiv is a predicate symbol.

The second \equiv 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 \equiv is semantic equality, i.e., the identity relation on the domain (\rightarrow p.73).

Rules¹⁰²

- Equality is an equivalence relation¹⁰³

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

- Equality is also a congruence¹⁰⁴ on terms and all rela-

¹⁰²Since $=$ is a logical symbol in the formalism of first-order logic with equality, there should be derivation rules (\rightarrow p.80) for $=$ to derive which formulas $a = b$ are true.

¹⁰³In 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 $a \equiv b \text{ mod } 6$.

¹⁰⁴In 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.

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

tions¹⁰⁵

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

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

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 mod 6 and 15 is divisible by 3, hence 9 is divisible by 3.

14 \equiv 8 mod 6 and 14 is not divisible by 3, hence 8 is not divisible by 3.

¹⁰⁵Why did we use letters t and A here?

Recall the rules for building terms (\rightarrow p.70) and atoms (\rightarrow p.71).

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 (\rightarrow p.31). $t(x_1, \dots, x_n)$ stands for any term in which x_1, \dots, x_n occur, and

Soundness of Rules

Equality in $I_{\mathcal{A}}$ is an equivalence relation¹⁰⁶ and functions/predicates/logical

$A(x_1, \dots, x_n)$ 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 (\rightarrow p.225).

¹⁰⁶On the semantic level, two things are equal if they are identical. Semantic equality is an equivalence relation (\rightarrow p.107). This semantic fact is so fundamental that we cannot explain it any further.

operators are “truth-functional”¹⁰⁷.

Adding further rules gives us an equational theory (\rightarrow p.122), e.g. groups (\rightarrow p.121).

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.

¹⁰⁷If $t(x)$ 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.

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.¹⁰⁸

¹⁰⁸The 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 (\rightarrow p.84).

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).

Examples

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

$$A = x^2 + y^2 > 12 \cdot x, \quad A = z^2 + y^2 > 12 \cdot x, \quad {}^{109}$$

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

We show two ways:

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

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

¹⁰⁹The 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 simply¹¹⁰ (using a tree syntax)

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

or literally

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

¹¹⁰The Isabelle FOL rule is:

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

In this rule, P is an Isabelle metavariable (\rightarrow p.31).

Why doesn't the Isabelle rule contain a z to mark (\rightarrow p.111) which occurrences should be replaced?

We cannot understand this yet (\rightarrow p.225), 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 .

Proving $\exists x. t = x$

$$\frac{}{t = t} \text{refl} (\rightarrow \text{p.107})$$

$$\frac{}{\exists x. t = x} \exists\text{-I} (\rightarrow \text{p.93})$$

In the rule $\frac{A(t)}{\exists x. A(x)} \exists\text{-I} (\rightarrow \text{p.93})$, “ $A(x)$ ” is metanotation (\rightarrow p.31).
 In the example, $A(x) = (t = x)$.

Notational confusion avoided by a precise metalanguage (\rightarrow p.204).

6 First-Order Theories

What Is a Theory?

Recall our intuitive explanation of theories (\rightarrow p.6).

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 (\rightarrow p.16).

A structure (\rightarrow p.73) in which these rules are true is then called a model (\rightarrow p.76) of the theory.

6.1 Example 1: Partial Orders

- The language of the theory of partial orders¹¹¹: \leq ¹¹²

¹¹¹A partial order is a binary relation that is reflexive, transitive (\rightarrow p.107), and anti-symmetric: $a \leq b$ and $b \leq a$ implies $a = b$.

¹¹² \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.

- Axioms (\rightarrow p.49)

$$\forall x, y, z. x \leq y \wedge y \leq z \rightarrow x \leq z^{113}$$

$$\forall x, y. x \leq y \wedge y \leq x \leftrightarrow x = y^{114}$$

- Alternative to axioms is to use 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.¹¹⁵

¹¹³The axiom $\forall x, y, z. x \leq y \wedge y \leq z \rightarrow x \leq z$ encodes transitivity.

¹¹⁴Note that $\forall x, y. x \leq y \wedge y \leq x \leftrightarrow x = y$ encodes both antisymmetry (\rightarrow) and reflexivity (\leftarrow). Recall (\rightarrow p.88) that $A \leftrightarrow B$ as shorthand for $A \rightarrow B \wedge B \rightarrow A$.

¹¹⁵One can see that using \rightarrow -I and \rightarrow -E (\rightarrow p.33), one can always convert a proof using the axioms to one using the proper (\rightarrow p.49) 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!

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 (\rightarrow p.45) as follows:

$$\frac{\frac{x = y}{y = x} \text{sym}}{y \leq x} \leq\text{-refl}$$

More on Orders

- A partial order (\rightarrow p.116) \leq is a linear or total order¹¹⁶ when

$$\forall x, y. x \leq y \vee y \leq x$$

Note: no “pure” rule formulation¹¹⁷ of this disjunction.

- A total order \leq is dense when, in addition

$$\forall x, y. x < {}^{118}y \rightarrow \exists z. (x < z \wedge z < y)$$

What does $<$ mean?

¹¹⁶We define these notions according to usual mathematical terminology.

A partial order (\rightarrow p.116) \leq is a linear or total order if for all a, b , either $a \leq b$ or $b \leq a$.

A partial order (\rightarrow p.116) \leq is dense if for all a, b where $a < b$, there exists a c such that $a < c$ and $c < b$.

¹¹⁷The axiom $\forall x, y. x \leq y \vee y \leq x$ cannot be phrased as a proper (\rightarrow p.49) rule in the style of, for example, the transitivity axiom (\rightarrow p.117).

¹¹⁸We use $s < t$ as shorthand for $s \leq t \wedge \neg s = t$.

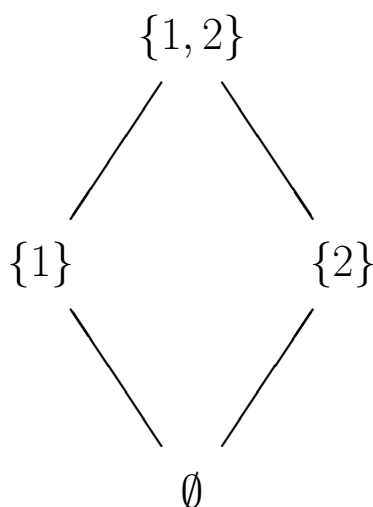
We say that $<$ is the strict part of the partial order (\rightarrow p.116) \leq .

Structures for Orders ...

Give structures (\rightarrow p.73) for orders that are ...

1. not total: \subseteq -relation¹¹⁹;
2. total but not dense: integers with \leq ;
3. dense: reals with \leq .

¹¹⁹The \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 (\rightarrow p.116) by a such a graph is quite common. Here, node a is below node b and connected by an arc if and only if $a < (\rightarrow$ p.119) b and there exists no c with $a < c < b$.

6.2 Example 2: Groups

- Language: Function symbols $_ \cdot _$, $_^{-1}$, e ¹²⁰

In this example, we have the partial order (\rightarrow p.116)

$$\{(\emptyset, \emptyset), (\{1\}, \{1\}), (\{1\}, \{1\}), (\{1, 2\}, \{1, 2\}), \\ (\emptyset, \{1\}), (\emptyset, \{1\}), (\{1\}, \{1, 2\}), (\{1\}, \{1, 2\})\}.$$

¹²⁰ $_ \cdot _$ 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.

\underline{e} is a nullary function symbol (= constant) (\rightarrow p.67).

Note that groups are very common in mathematics, and many different notations, i.e., function names and fixity (infix, prefix. . .) are used for them.

- A group is¹²¹ a model¹²² 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 theory¹²³.

Two theorems: (1) $x^{-1} \cdot x = e$ and (2) $e \cdot x = x$

We will now prove them.

¹²¹In 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 .

¹²²A model (\rightarrow p.76) of the group axioms is a structure (\rightarrow p.73) 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”.

¹²³An equational theory is a set of equations. Each equation is an axiom.

Theorem 1

$$\begin{array}{lll}
\forall x, y, z. (x \cdot y) \cdot z & = & x \cdot (y \cdot z) \quad (\text{assoc}) \\
\forall x. x \cdot e & = & x \quad (\text{r-neutr}) \\
\forall x. x \cdot x^{-1} & = & e \quad (\text{r-inv})
\end{array}$$

$$x^{-1} \cdot x = e \tag{1}$$

$$\begin{aligned}
x^{-1} \cdot x &= x^{-1} \cdot (x \cdot e) = x^{-1} \cdot (x \cdot (x^{-1} \cdot x^{-1-1})) = \\
x^{-1} \cdot ((x \cdot x^{-1}) \cdot x^{-1-1}) &= x^{-1} \cdot (e \cdot x^{-1-1}) = \\
(x^{-1} \cdot e) \cdot x^{-1-1} &= x^{-1} \cdot x^{-1-1} = e.
\end{aligned}$$

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 (\rightarrow p.49) rules where the premises are called conditions [Höl90].

Note also that sometimes, one also considers the basic rules of equality (\rightarrow p.107) 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.

Theorem 2

$$\forall x, y, z. (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad (\text{assoc})$$

$$\forall x. x \cdot e = x \quad (\text{r-neutr})$$

$$\forall x. x \cdot x^{-1} = e \quad (\text{r-inv})$$

$$e \cdot x = x \tag{2}$$

$$e \cdot x = (x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x) = x \quad (\text{Theorem 1})$$

6.3 Lessons Learned from these Examples

Equational proofs are often tricky!

- Equalities used in different directions, “eureka”¹²⁴ terms, etc.
- In some cases (the word problem¹²⁵ is) decidable.

¹²⁴By “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.

¹²⁵The 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 (\rightarrow p.76) of the theory.

Equational versus ND Proofs

- Above proofs were of a particular, equational form¹²⁶.
- In Isabelle this is accomplished by term rewriting.

Term rewriting is a process for replacing equals by equals (see later (\rightarrow p.276)).

- Alternative is natural deduction (\rightarrow p.25):
 - requires explicit proofs using equality rules;
 - tedious in practice. Try it on above examples!¹²⁷

¹²⁶An 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 (\rightarrow p.107). However, it looks very different from the natural deduction style (\rightarrow p.25).

¹²⁷

$$\begin{array}{c}
 \boxed{\text{r-neutr}} \quad \frac{x \cdot e = x}{x \cdot e = x} \quad \frac{\text{Theorem 1 } (\rightarrow \text{ p.121}) \quad \frac{x^{-1} \cdot x = e}{x^{-1} \cdot x = e}}{x^{-1} \cdot x = e} \quad \frac{\boxed{\text{assoc}} \quad \frac{(x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x)}{(x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x)}}{(x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x)} \quad \frac{\boxed{\text{r-inv}} \quad \frac{x \cdot x^{-1} = e}{x \cdot x^{-1} = e} \quad \text{sym } (\rightarrow \text{ p.107}) \quad \frac{e \cdot x = e \cdot x}{e \cdot x = e \cdot x}}{e \cdot x = (x \cdot x^{-1}) \cdot x} \\
 \hline
 e \cdot x = x \cdot e \quad e \cdot x = x \cdot (x^{-1} \cdot x) \\
 \hline
 e \cdot x = x
 \end{array}$$

Most steps use the congruence rule *cong*₂ (\rightarrow p.107).

Each framed box in the derivation tree stands for a sub-tree

7 Naïve Set Theory

7.1 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 (\rightarrow p.359), we will be completely formal.

consisting of a group axiom (\rightarrow p.121) and possibly several applications of \forall - E (\rightarrow p.81).

Sets: Language

Assuming any first-order language with equality, we add:

- set-comprehension $\{x|P(x)\}$ ¹²⁸ and a binary membership predicate \in .
- Term/formula distinction inadequate¹²⁹: need a syntactic category for sets.
- We will be more formal about syntax later (HOL (\rightarrow p.359)).
- Comprehension is a binding operator: x bound in $\{x|P(x)\}$ (\rightarrow p.72).

¹²⁸Set 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.

¹²⁹It is more adequate to regard a set as a term than as a formula. A set is a “thing”, not a statement about “things”. (\rightarrow p.69)

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. (\rightarrow p.71)

However, the syntax used in set comprehensions is not legal syntax for terms (\rightarrow p.70), since $P(x)$ is a formula.

This is why we introduce a special syntactic category for sets.

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?

$$2 \in \{w \mid 6 \notin \{x \mid x \text{ is divisible by } w\}\}$$

Answer: $6 \notin \{x \mid x \text{ divisible by } 2\}$ i.e., 6 not divisible by 2.

Proof Rules for Sets

Introduction, elimination, extensional equality¹³⁰

$$\frac{P(t)}{t \in \{x|P(x)\}} \text{ compr-I} \quad \frac{t \in \{x|P(x)\}}{P(t)} \text{ compr-E}$$

$$\frac{\forall x. x \in A \leftrightarrow x \in B}{A = B} =\text{-I} \quad \frac{A = B}{\forall x. x \in A \leftrightarrow x \in B} =\text{-E}$$

The following equivalence is derivable¹³¹:

$$\forall x. P(x) \leftrightarrow (\rightarrow \text{ p.88}) x \in \{y|P(y)\}$$

¹³⁰Two 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.

¹³¹

$$\frac{\frac{[P(x)]^1}{x \in \{y|P(y)\}} \text{ compr-I} \quad \frac{[x \in \{y|P(y)\}]^2}{P(x)} \text{ compr-E}}{P(x) \rightarrow x \in \{y|P(y)\} \rightarrow\text{-I}^1 \quad x \in \{y|P(y)\} \rightarrow P(x) \rightarrow\text{-I}^2} \wedge\text{-I}$$

$$\frac{P(x) \rightarrow x \in \{y|P(y)\} \wedge x \in \{y|P(y)\} \rightarrow P(x)}{P(x) \leftrightarrow x \in \{y|P(y)\}} \text{ iff } (\rightarrow \text{ p.88})$$

$$\frac{P(x) \leftrightarrow x \in \{y|P(y)\}}{\forall x. P(x) \leftrightarrow x \in \{y|P(y)\}} \forall\text{-I}$$

Rules $\wedge\text{-I}$ (\rightarrow p.31), $\rightarrow\text{-I}$ (\rightarrow p.33), $\forall\text{-I}$ (\rightarrow p.81) were defined in previous lectures. The step marked with iff (\rightarrow p.88) is not a proof step in the technical sense. We only make the expansion of a shorthand notation explicit.

7.2 Digression: Sorted Reasoning

- In mathematical arguments we often (implicitly) assume that variables are restricted to some universe of discourse¹³².

E.g., $x^2 < 9$ (universe either \mathbb{R} , \mathbb{N} , ...)

- To avoid ambiguity¹³³ we can include sort information in formulae:

members x of U where $P(x) \equiv \{x \in U \mid P(x)\}$

Formally

$$\{x \in U \mid P(x)\} \equiv \{x \mid U(x) \wedge P(x)\}.$$

¹³²We already know what a universe (\rightarrow p.73) or domain (\rightarrow p.73) is. To interpret a particular language, we have a structure (\rightarrow p.73) 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 \mathbb{R} (the real numbers) and \mathbb{N} (the natural numbers) to say that \sqrt{x} requires x to be of sort \mathbb{R} and $x!$ requires x to be of sort \mathbb{N} .

¹³³We 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

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 unsorted¹³⁴. We have $y \in \{x \in U \mid P(x)\} \leftrightarrow y \in \{x \mid U(x) \wedge P(x)\} \leftrightarrow U(y) \wedge P(y)$

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 $\{x \in U \mid P(x)\}$ 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

$$\{x \in U \mid P(x)\} \equiv \{x \mid U(x) \wedge P(x)\}.$$

¹³⁴In sorted logic, sorts are part of the syntax. So the signature (\rightarrow p.69) 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.

Sorted Quantification

$$\begin{aligned}\forall x \in U. P(x)^{135} &\equiv \forall x. U(x) \rightarrow P(x) \\ \exists x \in U. P(x) &\equiv \exists x. U(x) \wedge P(x)\end{aligned}$$

In contrast, our logic is unsorted. The special syntax we provide for sorted reasoning is just syntactic sugar (\rightarrow p.21), 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.

¹³⁵So $\forall x \in U. P(x)$ is simply a shorthand or syntactic sugar (\rightarrow p.21) for $\forall x. x \in U \rightarrow P(x)$, and analogously for $\exists x \in U. P(x)$.

7.3 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

$$A \subseteq B \equiv \forall x. x \in A \rightarrow x \in B$$

136

\cap is called intersection.

\cup is called union.

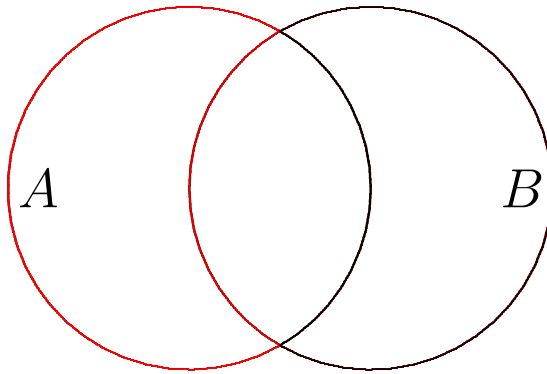
\setminus is called set difference.

\subseteq is called inclusion.

Examples of Operations on Sets

One often depicts sets as circles or bubbles.

What are $A \cap B$, $A \cup B$, $A \setminus B$?



$A \cap B$

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 (\rightarrow p.134) and $\forall x. P(x) \leftrightarrow x \in \{y|P(y)\}$ (\rightarrow p.130).

Example: what is the logical form¹³⁷ of $x \in ((A \cap B) \cup (A \cap C))$? $(x \in A \wedge x \in B) \vee (x \in A \wedge x \in C)$

¹³⁷When 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.

Proof of $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ (1)

Venn diagram (Is this a proof?)¹³⁸

¹³⁸A 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 (\rightarrow p.16).

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.

Proof of $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ (2)

Natural deduction (natural language¹³⁹)

By extensionality (\rightarrow p.130), 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.

¹³⁹We 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.

Same in Isabelle

Last proof carries over to Isabelle: extensionality (\rightarrow p.130), rewriting (\rightarrow p.276), tautology (\rightarrow p.76) checking. Do it!

Prove: for all Sets A and B , $((A \cup B) \setminus B) \subseteq A$

Let's try a similar semi-formal proof:

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 $((A \cup B) \setminus B) \subseteq A$.

Combination¹⁴⁰ of forward reasoning with backward reasoning. This is common in practice and usually easy to unscramble.

140

Let A and B be arbitrary sets.

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 $x \in (A \cup B) \setminus B \rightarrow x \in A$

Therefore $((A \cup B) \setminus B) \subseteq A$ (conclusion)

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.

7.4 Extending Set Comprehensions

Recall set comprehensions (\rightarrow p.128) $\{x|P(x)\}$.

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 (\rightarrow p.130) $\exists x. x > 5 \wedge t = x^2$.

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 $f(x)$.

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.

$$\begin{aligned} x \in \{m_s | s \in S\} &\leftrightarrow (\rightarrow \text{p.141}) \quad x \in \{y | \exists s. s \in S \wedge y = m_s\} \\ &\leftrightarrow (\rightarrow \text{p.130}) \quad \exists s. s \in S \wedge x = m_s \\ &\leftrightarrow (\rightarrow \text{p.133}) \quad \exists s \in S. x = m_s \end{aligned}$$

Uses extended comprehensions (\rightarrow p.141), indexing syntax, and sorted quantification (\rightarrow p.133).

Logical Forms of the New Notation

Question: what is the logical form (\rightarrow p.136) of $\{x_i | i \in I\} \subseteq A$?

$$\begin{aligned} \forall x. x \in \{x_i | i \in I\} &\rightarrow x \in A^{141}, \quad \text{i.e.,} \\ \forall x. (\exists i \in I. x = x_i) &\rightarrow x \in A^{142}. \end{aligned}$$

Intuition¹⁴³ suggests that $\forall i \in I. x_i \in A$ (\rightarrow p.133) 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 exercise¹⁴⁴ on using extended comprehensions (\rightarrow p.141), indexing syntax, and sorted quantification (\rightarrow p.133).

¹⁴¹

$$\{x_i | i \in I\} \subseteq A \equiv \forall x. x \in \{x_i | i \in I\} \rightarrow x \in A$$

follows from the definition of \subseteq (\rightarrow p.136).

¹⁴²

We want to show

$$\forall x. x \in \{x_i | i \in I\} \rightarrow x \in A \equiv \forall x. (\exists i \in I. x = x_i) \rightarrow x \in A$$

$$\begin{aligned} x \in \{x_i | i \in I\} &\equiv (\text{def. of notation}) (\rightarrow \text{p.141}) \\ x \in \{y | \exists i. i \in I \wedge y = x_i\} &\equiv \text{compr-}I (\rightarrow \text{p.130}) \\ \exists i. i \in I \wedge x = x_i &\equiv (\text{Sorted quantification}) (\rightarrow \text{p.133}) \\ \exists i \in I. x = x_i & \end{aligned}$$

¹⁴³It may be helpful to pronounce both forms out loud in natural language to get an intuitive feeling that they are equivalent.

¹⁴⁴Want to prove

$$(\forall x. (\exists i \in I. x = x_i) \rightarrow x \in A) \leftrightarrow (\forall i \in I. x_i \in A)$$

Powersets

$$\wp(A) = \{x \mid x \subseteq A\}.$$

What is the logical form (\rightarrow p.136) 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., $\forall x. x \in A \rightarrow x \in B$.

• “ \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

$$(\forall x. (\exists i \in I. x = x_i) \rightarrow x \in A) \leftrightarrow (\forall i \in I. x_i \in A)$$

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.

7.5 Outlook

Sets can have other sets as elements.

Since we have $\forall x.(\exists i \in I. x = x_i) \rightarrow x \in A$, by rule \forall -E (\rightarrow p.81) 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 $(\exists i \in I. x = x_i) \rightarrow x \in A$,

Implicitly assume that universe of discourse is collection¹⁴⁵ of all sets.

which is what was to be proven.

This proof could be made more formal by drawing a proof tree or using Isabelle.

¹⁴⁵We 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 (→ p.127) 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.

Russell's Paradox

Suppose $U := \{x \mid \top^{146}\}$. Then¹⁴⁷ $U \in U$.

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 form¹⁴⁸ 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. Or even more succinctly...

¹⁴⁶Assume that \top is syntactic sugar (\rightarrow p.21) for a proposition that is always true, say $\top \equiv \perp \rightarrow \perp$. We have not introduced this, but it is convenient.

So semantically (\rightarrow p.75), we have $I_{\mathcal{A}}(\top) = 1$ for all $I_{\mathcal{A}}$.

¹⁴⁷Recall that a set comprehension (\rightarrow p.128) has the form $\{x \mid P(x)\}$, 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 (\rightarrow p.130), this means that $x \in U$ for any x . Thus in particular, $U \in U$.

¹⁴⁸

Recall $R := \{A \mid A \notin A\}$ and recall the notion of logical form (\rightarrow p.136).

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 (\rightarrow p.130), we can

Russell's Paradox (Cont.)

Recall $R := \{A \mid A \notin A\}$.

Assume $R \in R$. By the definition of R , this means $R \in \{A \mid A \notin A\}$. Using *compr-E* (\rightarrow p.130), this implies $R \notin R$.

Now assume $R \notin R$. Using *compr-I* (\rightarrow p.130), this implies $R \in \{A \mid A \notin A\}$. By the definition of R , this means $R \in R$.

What does this tell us about sets?¹⁴⁹

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 (\rightarrow p.88), we have $A \in R \leftrightarrow A \notin A$, and since A was arbitrary, by \forall -I (\rightarrow p.81), we have $\forall A. (A \in R \leftrightarrow A \notin A)$.

¹⁴⁹It 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” (\rightarrow p.127) 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

Where Do We Go from here?

- The λ -calculus (\rightarrow p.150) as basis for a metalanguage (\rightarrow p.204) to avoid notational confusion (\rightarrow p.114)
- Resolution (\rightarrow p.236) and other deduction techniques (\rightarrow p.249): understanding Isabelle better and achieving a higher level of automation
- Higher-order logic (\rightarrow p.359): a formalism for (among other things) non-naïve set theory¹⁵⁰

order”, that is, sets that contain sets of the first kind. This process can be continued ad infinitum.

The formal details will come later (\rightarrow p.359).

¹⁵⁰Higher-order logic (\rightarrow p.359) is a solution to the dilemma posed by Russell’s paradox. (\rightarrow p.148)

It is a surprisingly simple formalism which can be extended (\rightarrow p.434) 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.

8 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!

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!

Further reading: [Tho91, chapter 2], [HS90, chapter 1].

Outline of this Lecture

- The untyped λ -calculus
- The simply typed λ -calculus (\rightarrow p.170) (λ^{\rightarrow})
- An extension of the typed λ -calculus (\rightarrow p.184)
- Higher-order unification (\rightarrow p.200)

8.1 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 8¹⁵¹.

¹⁵¹As 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.

Syntax

$(x \in Var, c \in Const^{152})$

$e ::= x \mid c \mid (ee) \mid (\lambda x. e)^{153}$

¹⁵²Similarly as for first-order logic (\rightarrow p.69), 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 (\rightarrow p.67). 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 (\rightarrow p.153) 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.

¹⁵³A λ -term can either be

The objects generated by this grammar¹⁵⁴ are called λ -terms or simply terms.

- 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 $(\lambda x.e)$).

¹⁵⁴A notation like

$$\begin{aligned}
 e &::= x \mid c \mid (ee) \mid (\lambda x.e) \\
 \tau &::= T \mid \tau \rightarrow \tau \\
 e &::= x \mid c \mid (ee) \mid (\lambda x^\tau.e) \\
 P &::= x \mid \neg P \mid P \wedge P \mid P \rightarrow P \dots
 \end{aligned}$$

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

Conventions: iterated λ & left-associated application¹⁵⁵

$$\begin{aligned}(\lambda x. (\lambda y. (\lambda z. ((xz)(yz))))) &\equiv (\lambda xyz. ((xz)(yz))) \\ &\equiv \lambda xyz. xz(yz)\end{aligned}$$

Is $\lambda x. x + 5$ a λ -term?¹⁵⁶

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](#).

¹⁵⁵We 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 (\rightarrow p.21). There are some good reasons for these conventions.

¹⁵⁶Strictly 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

Substitution

- Reduction¹⁵⁷ based on substitutions (\rightarrow p.111)

$$(\lambda x. g x 3)(5) = (g x 3)[x \leftarrow 5]^{158} = g 5 3$$

- Must respect free (\rightarrow p.158) and bound (\rightarrow p.158) variables,

$$(\lambda x. x(\lambda x. xy))(e) = ((x(\lambda x. xy))[x \leftarrow e] = e(\lambda x. xy)$$

- Same problems as with quantifiers (\rightarrow p.84)

$$\frac{\forall x. (P(x) \wedge \exists x. Q(x, y))}{P(e) \wedge \exists x. Q(x, y)} \forall\text{-E} \quad \frac{\forall x. (P(x) \wedge \exists y. Q(x, y))}{P(y) \wedge \exists z. Q(y, z)} \forall\text{-E}$$

λ -terms, allowing for:

- application to several arguments rather than just one;
- infix notation (\rightarrow p.67).

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 Curried (\rightarrow p.164) way).

¹⁵⁷Reduction is the notion of “computing”, or “evaluation”, in the λ -calculus.

¹⁵⁸Here 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.

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: $FV(\textcolor{green}{xy}(\lambda yz. \textcolor{green}{xyz})) = \{x, y\}$

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’¹⁵⁹.

¹⁵⁹Recall the definition (\rightarrow p.84) 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 (\rightarrow p.128). 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 (\rightarrow p.231).

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., $\lambda x. xx$ ¹⁶⁰.

¹⁶⁰If 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 $\lambda x. xx$.

Reduction: Intuition

Reduction is the notion of “computing”, or “evaluation”, in the λ -calculus.

$$f\ x = x + 5 \ (\rightarrow \text{p.152}) \rightsquigarrow f = \lambda x. x + 5$$

$$f\ 3 = 3 + 5 \rightsquigarrow (\lambda x. x + 5)(3) \rightarrow_{\beta} (x + 5)[x \leftarrow 3] = 3 + 5$$

β -reduction replaces (\rightarrow p.153) a parameter by an argument¹⁶¹.

This should propagate into contexts¹⁶², e.g.

$$\lambda x. (\underline{(\lambda x. x + 5)(3)}) \rightarrow_{\beta} \lambda x. (3 + 5).$$

¹⁶¹In 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.

¹⁶²In

$$\lambda x. (\underline{(\lambda x. x + 5)(3)}),$$

the underlined part is a subterm occurring in a context. β -reduction should be applicable to this subterm.

Reduction: Definition

- β -reduction: $(\lambda x.M)N \rightarrow_{\beta} M[x \leftarrow N]$ ¹⁶³
- Rules for contraction (of redices¹⁶⁴) in contexts:

$$\frac{M \rightarrow_{\beta} M'}{NM \rightarrow_{\beta} NM'} \quad \frac{M \rightarrow_{\beta} M'}{MN \rightarrow_{\beta} M'N} \quad \frac{M \rightarrow_{\beta} M'}{\lambda z.M \rightarrow_{\beta} \lambda z.M'}$$
- Reduction is reflexive-transitive (\rightarrow p.107) closure

$$\frac{M \rightarrow_{\beta} N}{M \rightarrow_{\beta}^* N} \quad \frac{}{M \rightarrow_{\beta}^* M} \quad \frac{M \rightarrow_{\beta}^* N \quad N \rightarrow_{\beta}^* P}{M \rightarrow_{\beta}^* P}$$
- A term without redices is in β -normal form.

¹⁶³As you see, β -reduction is defined using rules (two of them being axioms (\rightarrow p.49), the rest proper rules (\rightarrow p.49)) in the same way that we have defined proof systems for logic (\rightarrow p.16) before. Note that we wrote the first axiom (\rightarrow p.49) defining β -reduction without a horizontal bar.

¹⁶⁴In 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.

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 Currying¹⁶⁵

$$\underline{(\lambda x. xx)(\lambda x. xx)} \rightarrow_{\beta} \underline{(\lambda x. xx)(\lambda x. xx)} \rightarrow_{\beta} \dots$$

Shows divergence¹⁶⁶

$$\text{But } \underline{(\lambda xy. y)((\lambda x. xx)(\lambda x. xx))} \rightarrow_{\beta} \lambda y. y$$

¹⁶⁵You 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 (\rightarrow p.156), so $g a b$ is read $(g a) b$.

Currying will become even clearer once we introduce the typed λ -calculus (\rightarrow p.171).

¹⁶⁶We 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

$$(\lambda xy. y)((\lambda x. xx)(\lambda x. xx)) \rightarrow_{\beta} \lambda y. y$$

Conversion

- β -conversion: “symmetric closure” (\rightarrow p.107) 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 implicit¹⁶⁷)

$$\lambda x.M =_{\alpha} \lambda z.M[x \leftarrow z] \quad \text{where } z \notin FV(M)$$

- η -conversion: for normal-form analysis¹⁶⁸

$$M =_{\eta} \lambda x.(Mx) \quad \text{if } x \notin FV(M)$$

but there is also a diverging sequence

$$(\lambda xy.y)((\lambda x.xx)(\lambda x.xx)) \rightarrow_{\beta} (\lambda xy.y)((\lambda x.xx)(\lambda x.xx)) \rightarrow_{\beta} \dots$$

¹⁶⁷ α -conversion is usually applied implicitly, i.e., without making it an explicit step. So for example, one would simply write:

$$\lambda z.z =_{\beta} \lambda x.x$$

¹⁶⁸ 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.

$$gx =_{\eta} \lambda y.gxy \quad \text{reflects} \quad gx b =_{\beta} (\lambda y.gxy)b$$

More specifically: if we did not have the η -conversion rule, then gx and $\lambda y.gxy$ 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.

One also says that the η -conversion expresses the idea of extensionality (\rightarrow p.130) [HS90, chapter 7].

Note that with the help of contraction and transitivity (\rightarrow p.163), η -conversion can be generalized to more than one variable, i.e. $M =_{\beta\eta} \lambda x_1 \dots x_n. M x_1 \dots x_n$. E.g. we can derive $\lambda x y z. M x y z =_{\beta\eta} M$:

$$\begin{array}{c}
 \lambda z. M x y z =_{\eta} M x y \\
 \hline
 \lambda y z. M x y z =_{\beta\eta} \lambda y. M x y \quad \lambda y. M x y =_{\eta} M x \\
 \hline
 \lambda y z. M x y z =_{\beta\eta} M x \\
 \hline
 \lambda x y z. M x y z =_{\beta\eta} \lambda x. M x \qquad \lambda x. M x =_{\eta} M \\
 \hline
 \lambda x y z. M x y z =_{\beta\eta} M
 \end{array}$$

For any n , we call $\lambda x_1 \dots x_n. M x_1 \dots x_n$ an η -expansion of M .

λ -Calculus Meta-Properties¹⁶⁹

Confluence (equivalently¹⁷⁰, 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 there exists a P where $N_1 \rightarrow_{\beta}^* P$ and $N_2 \rightarrow_{\beta}^* P$.

¹⁶⁹By metaproperties, we mean properties about reduction and conversion sequences in general.

¹⁷⁰A 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 $N_1 \leftrightarrow^* N_2$, then there exists a P where $N_1 \rightarrow^* P$ and $N_2 \rightarrow^* P$.

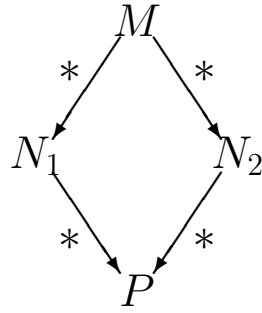
Here, $\leftarrow := (\rightarrow)^{-1}$ is the inverse of \rightarrow , and $\leftrightarrow := \leftarrow \cup \rightarrow$ is the symmetric closure of \rightarrow , and $\leftrightarrow^* := (\leftrightarrow)^*$ is the reflexive transitive symmetric closure of \rightarrow .

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 $M_1 \leftrightarrow^* M_9$.

Confluence is equivalent to the Church-Rosser property [BN98, page 10].



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:

$$\begin{aligned}
 (\lambda xy. y)(\underline{(\lambda x. xx)a}) &\rightarrow_{\beta} \underline{(\lambda xy. y)(aa)} \rightarrow_{\beta} \lambda y. y \\
 \underline{(\lambda xy. y)(\lambda x. xx)a} &\rightarrow_{\beta} \lambda y. y
 \end{aligned}$$

Turing Completeness

The λ -calculus can represent all computable functions.¹⁷¹

¹⁷¹The 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 [HS90, chapter 4]. 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.

8.2 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 objects¹⁷² called terms (\rightarrow p.154).

We now introduce syntactic objects called types¹⁷³.

We will say “a term has a type” or “a term is of a type”.

¹⁷²We also say that we have defined a term language (\rightarrow p.154). A particular language is given by a signature, although for the untyped λ -calculus this is simply the set of constants *Const*.

¹⁷³We 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 (\rightarrow p.20) and \top .

All that matters is that \mathcal{B} is some fixed set “defined by the user”.

Two Syntaxes

- Syntax for types (\mathcal{B} a set of base types (\rightarrow p.170), $T \in \mathcal{B}$)

$$\tau ::= T \mid \tau \rightarrow \tau \quad (\rightarrow \text{ p.155})$$

Examples: \mathbb{N} , $\mathbb{N} \rightarrow \mathbb{N}$, $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$, $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ ¹⁷⁵

- Syntax for (raw¹⁷⁶) terms: λ -calculus (\rightarrow p.154) augmented with types¹⁷⁷

$$e ::= (\rightarrow \text{ p.155}) \ x \mid c \mid (ee) \mid (\lambda x^\tau. e)$$

¹⁷⁴The 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.

¹⁷⁵To 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 (\rightarrow p.156). This may seem confusing at first, but actually, it turns out that the two conventions concerning associativity fit together very neatly.

¹⁷⁶In the context of typed versions of the λ -calculus, raw terms are terms built ignoring any typing conditions (\rightarrow p.175). So raw terms are simply terms as defined for the untyped λ -calculus (\rightarrow p.154), possibly augmented with type superscripts.

¹⁷⁷So far, this is just syntax!

$$(x \in Var, c \in Const^{178})$$

The notation $(\lambda x^\tau. e)$ simply specifies that binding (\rightarrow p.158) 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.

¹⁷⁸ Var and $Const$ are the sets of variables and constants, respectively, as for the untyped λ -calculus (\rightarrow p.154).

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 ($c \in \text{Const}$ (\rightarrow p.172))

$$\Sigma ::= \langle \rangle \mid \Sigma, c : \tau^{179}$$

- 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?¹⁸⁰

¹⁷⁹We 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.

¹⁸⁰For propositional logic (\rightarrow p.19), 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 (\rightarrow p.19).

In first-order logic (\rightarrow p.69), 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 (\rightarrow p.132), the signature must also specify

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

$$\Gamma \vdash_{\Sigma} c \ x : \sigma^{181}$$

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.

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 (\rightarrow p.154).

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 $c \in \text{Const}$, and in fact, that Σ is clear from the context as well. Since Σ contains an expression of the form $c : \tau$ for each $c \in \text{Const}$, it is redundant to give *Const* explicitly. It is sufficient to give Σ .

¹⁸¹The expression

$$\Gamma \vdash_{\Sigma} c \ x : \sigma$$

Type Assignment Calculus: Rules¹⁸²

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 (\rightarrow p.48) before.

¹⁸²Type assignment is defined as a system of rules for deriving type judgements (\rightarrow p.174), in the same way that we have defined derivability judgements (\rightarrow p.48) for logics (\rightarrow p.16), and β -reduction (\rightarrow p.163) for the untyped λ -calculus.

$$\begin{array}{c}
\frac{c : \tau \in {}^{183}\Sigma}{\Gamma \vdash c : \tau} \text{assum} \qquad \Gamma, x : \tau, \Delta \vdash x : \tau \quad \text{hyp}^{184} \\
\\
\frac{\Gamma \vdash e : \sigma \rightarrow \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash ee' : \tau} \text{app} \qquad \frac{\Gamma, x : \sigma^{185} \vdash e : \tau}{\Gamma \vdash \lambda x^\sigma. e : \sigma \rightarrow \tau} \text{abs}
\end{array}$$

¹⁸³Recall that Σ is a sequence (\rightarrow p.176). 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 $c : \tau \in \Sigma'$.

¹⁸⁴One 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.

¹⁸⁵A 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

Note analogy to minimal logic over \rightarrow ¹⁸⁶.

β -reduction defined as before (\rightarrow p.163), has subject re-

$\langle o, o_1, \dots, o_n \rangle$.

An empty sequence is denoted by $\langle \rangle$.

¹⁸⁶Recall the sequent rules (\rightarrow p.49) 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

$$\Gamma, x : \tau, \Delta \vdash x : \tau \quad \text{hyp}$$

of λ^\rightarrow and throw away the terms (so you keep only the types), you obtain essentially the rule for assumptions

$$\Gamma \vdash A \quad (\text{where } A \in \Gamma) \quad (\rightarrow \text{ p.49})$$

of propositional logic.

Likewise, if you do the same with the rule

$$\frac{\Gamma \vdash e : \sigma \rightarrow \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash ee' : \tau} \quad \text{app}$$

of λ^\rightarrow , you obtain essentially the rule

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

(\rightarrow p.49)of propositional logic.

Finally, if you do the same with the rule

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x^\sigma. e : \sigma \rightarrow \tau} \text{abs}$$

of λ^\rightarrow , you obtain essentially the rule

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

(\rightarrow p.49)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 (\rightarrow p.177).

duction property¹⁸⁷ and is strongly normalizing¹⁸⁸.

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 [Tho91]. 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 .

¹⁸⁷Subject reduction is the following property: reduction (\rightarrow p.163) does not change the type of a term, so if $\vdash_\Sigma M : \tau$ and $M \rightarrow_\beta N$, then $\vdash_\Sigma N : \tau$.

¹⁸⁸The simply-typed λ -calculus, unlike the untyped λ -calculus (\rightarrow p.152), 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.

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 types¹⁸⁹!

For simplicity, applications of *hyp* (\rightarrow p.175) are usually not explicitly marked in proof.

¹⁸⁹In 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 (\rightarrow p.35).

Note also that Σ (\rightarrow p.173) is irrelevant for the example and hence arbitrary.

Example 2

$$\Gamma = f : \sigma \rightarrow \sigma \rightarrow \tau, x : \sigma$$

$$\frac{\Gamma \vdash f : \sigma \rightarrow \sigma \rightarrow \tau \quad \Gamma \vdash x : \sigma}{\Gamma \vdash f x : \sigma \rightarrow \tau} \text{app} \quad \frac{\Gamma \vdash f x : \sigma \rightarrow \tau \quad \Gamma \vdash x : \sigma}{\Gamma \vdash f x x : \tau} \text{app}$$

$$\frac{\Gamma \vdash f x x : \tau}{f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^\sigma. f x x : \sigma \rightarrow \tau} \text{abs}$$

$$\frac{f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^\sigma. f x x : \sigma \rightarrow \tau}{\vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}. \lambda x^\sigma. f x x : (\sigma \rightarrow \sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau} \text{abs}$$

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 constant¹⁹⁰.

We will often suppress applications of *assum* (\rightarrow p.175).

¹⁹⁰In Example 3, we have $f : \sigma \rightarrow \sigma \rightarrow \tau \in \Sigma$, and so f is a constant (\rightarrow p.173).

In Example 2, we have $f : \sigma \rightarrow \sigma \rightarrow \tau \in \Gamma$, and so f is a variable (\rightarrow p.173).

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

Type Assignment and $\alpha\beta\eta$ -Conversion

Type construction:

- Type construction¹⁹¹ is decidable.
- There is a practically useful implementation for type-construction (Hindley-Milner algorithm \mathcal{W} [Mil78, NN99]).

Term congruence¹⁹² ($e =_{\alpha\beta\eta} e'$? (\rightarrow p.165)) is decidable.

1, 2, ... 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 $\mathbb{N} \rightarrow \mathbb{N}$.

¹⁹¹Type construction is the problem of given a Σ , Γ and e , finding a τ such that $\Sigma, \Gamma \vdash e : \tau$.

Sometimes one also considers the problem where Γ is unknown and must also be constructed.

¹⁹² $\alpha\beta\eta$ -conversion is defined as for λ^\rightarrow (\rightarrow p.165). Given two (extended) λ -terms e and e' , it is decidable whether $e =_{\alpha\beta\eta} e'$.

8.3 Polymorphism and Type Classes

We will now look at the typed λ -calculus extended by polymorphism (\rightarrow p.185) and type classes (\rightarrow p.189).

As we will see later (\rightarrow p.204), 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 polymorphic¹⁹³.

Type language (\rightarrow p.170) 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, $\text{int list} \rightarrow \text{int list}$.

¹⁹³In 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 (\rightarrow p.190).

In a type system with polymorphism, the notion of base type (\rightarrow p.171) (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 (\rightarrow p.67). However, other conven-

Polymorphism: Two Syntaxes

- Syntax for polymorphic types (\mathcal{B} a set of type constructors¹⁹⁴ including \rightarrow), $T \in \mathcal{B}$, α is a type variable)

$$\tau ::= \alpha \mid (\tau, \dots, \tau) T \quad (\rightarrow \text{ p.155})$$

Examples: $\mathbb{N}, \mathbb{N} \rightarrow (\rightarrow \text{ p.171})\mathbb{N}, \alpha \text{ list}, \mathbb{N} \text{ list}, (\mathbb{N}, \text{bool}) \text{ pair}$.

- Syntax for (raw (\rightarrow p.171)) terms as before (\rightarrow p.171):

$$e ::= (\rightarrow \text{ p.155}) x \mid c \mid (ee) \mid (\lambda x^\tau (\rightarrow \text{ p.171}). e) \\ (x \in \text{Var}, c \in \text{Const} (\rightarrow \text{ p.172}))$$

tions exist, even within Isabelle (\rightarrow p.196).

See [Pau96, Tho95b, Tho99] for details on the polymorphic type systems of functional programming languages.

¹⁹⁴As before (\rightarrow p.170), 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 (\rightarrow p.69). The intention is that a type constructor may be applied to types.

Following the conventions of ML [Pau96], we write types in postfix notation (\rightarrow p.67), 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 (\rightarrow p.196), however written infix rather than postfix.

So the \mathcal{B} is some fixed set “defined by the user”, but it

Polymorphic Type Assignment Calculus

Type substitutions (denoted Θ) defined in analogy to substitutions in FOL¹⁹⁵. Apart from application of Θ in rule *assum*, type assignment is as for λ^\rightarrow (\rightarrow p.175):

$$\frac{c : \tau \in (\rightarrow \text{p.176})\Sigma}{\Gamma \vdash c : \tau\Theta} \text{assum}^* \quad \Gamma, x : \tau, \Delta \vdash x : \tau \quad \text{hyp} (\rightarrow \text{p.176})$$

$$\frac{\Gamma \vdash e : \sigma \rightarrow \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash ee' : \tau} \text{app} \quad \frac{\Gamma, x : \sigma (\rightarrow \text{p.176}) \vdash e : \tau}{\Gamma \vdash \lambda x^\sigma. e : \sigma \rightarrow \tau} \text{abs}$$

*: Θ is any type substitution.

should definitely always include \rightarrow .

¹⁹⁵A type substitution replaces a type variable by a type, just like in first-order logic (\rightarrow p.84), a substitution replaces a variable by a term.

Type Classes: Intuition

Type classes¹⁹⁶ are a way of “making ad-hoc polymorphism¹⁹⁷

¹⁹⁶Type classes are a way of “making ad-hoc polymorphism (\rightarrow p.190) 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 (\rightarrow p.67)) may exist and for some it may 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 (\rightarrow p.185),

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.

but also somewhat different from ad-hoc polymorphism (\rightarrow p.190): The types of the symbols must not be declared separately. E.g., one has to declare only once that \leq is of type $(a :: \text{ord} \rightarrow \alpha)$.

¹⁹⁷Ad-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 (\rightarrow p.185), but also somewhat different from type classes (\rightarrow p.189).

Type classes (\rightarrow p.189) are a way of “making ad-hoc polymorphism less ad-hoc” [HHPW96, WB89].

Type Classes in Isabelle

- Syntactic classes¹⁹⁸ (similarly as in Haskell): E.g., declare that there exists a class *ord* which is a subclass of class *term*, and that for any $\tau :: \textit{ord}$, the constant \leq is defined and has type $\tau \rightarrow \tau \rightarrow \textit{bool}$. Isabelle has syntax for this.

¹⁹⁸A 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 <= : ['a::ord, 'a] => 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” $\textit{ord} \prec \textit{term}$. The class *term* must be defined elsewhere.

The second line declares a symbol \leq . Such a declaration is preceded by the keyword **const**. The notation $\alpha :: \textit{ord}$ stands for a type variable constrained to be in class *ord*. So \leq is declared to be of type $[\alpha :: \textit{ord}, \alpha] \Rightarrow \textit{bool}$, 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 (\rightarrow p.171) in Isabelle. Note that the

- Axiomatic classes¹⁹⁹: 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 (\rightarrow p.193).

second occurrence of α is written without $:: \text{ord}$. This is because it is enough to state the class constraint once.

Note also that $[\alpha :: \text{ord}, \alpha] \Rightarrow \text{bool}$ is in fact just another way of writing $\alpha :: \text{ord} \Rightarrow \alpha \Rightarrow \text{bool}$, similarly as for goals (\rightarrow p.245).

Haskell [HHPW96] has type classes but ML [Pau96] hasn't.

¹⁹⁹In 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

To use a class, we can declare members²⁰⁰ of it, e.g., \mathbb{N} is a member of *ord*.

\implies is the implication symbol in Isabelle (that is to say, the metalevel implication).

Whenever an Isabelle theory declares (\rightarrow p.194) 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.

²⁰⁰One 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 $\alpha[\alpha \leftarrow (\mathbb{N} \rightarrow \mathbb{N})] = \mathbb{N} \rightarrow \mathbb{N}$ (\rightarrow p.187). 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
```

Syntax: Classes, Types, and Terms

Based on

- a set of type classes²⁰¹, say $\mathcal{K} = \{ord, order, lattice, \dots\}$,

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 (\rightarrow p.192) 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” $\tau :: \kappa$.

²⁰¹The 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 (\rightarrow p.192).

In grammars and elsewhere, κ is the letter we use for “type class”.

- a set of type constructors²⁰², say $\mathcal{B} = \{bool, _ \rightarrow _^{203}, ind, _ list, _ set\}$.

²⁰²As before, the set \mathcal{B} we gave is incomplete (there are "...") and just exemplary. We might call \mathcal{B} a type signature (\rightarrow p.170).

Note also that an $_$ is used to denote the arity of a type constructor (\rightarrow p.185).

- $_ list$ 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. $_ list$ 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 as functions taking terms to a Boolean), and so is \rightarrow , the constructor (\rightarrow p.196) of the function space between two types (\rightarrow p.171).

- a set of constants (\rightarrow p.172) $Const$ and a set of variables (\rightarrow p.172) Var ,

we define

- Polymorphic types²⁰⁴: $\tau ::= (\rightarrow \text{p.155}) \alpha \mid \alpha :: \kappa \mid (\tau, \dots, \tau) T$
- Raw (\rightarrow p.171) terms (as before (\rightarrow p.171)): $e ::= (\rightarrow \text{p.155}) x \mid c \mid (ee) \mid (\lambda x^\tau (\rightarrow \text{p.171}). e)$

(α is type variable, $T \in \mathcal{B}$ (\rightarrow p.196), $\kappa \in \mathcal{K}$ (\rightarrow p.195), $x \in Var$, $c \in Const$ (\rightarrow p.172))

In grammars and elsewhere, T is the letter we use for “type constructor”.

²⁰³In λ^\rightarrow , types were built from base types using a “special symbol” \rightarrow (\rightarrow p.171).

When we generalize λ^\rightarrow to a λ -calculus with polymorphism, this “special symbol” becomes a type constructor. However, the syntax (\rightarrow p.196) is still special, and it is interpreted in a particular way (\rightarrow p.171).

²⁰⁴ $\tau ::= (\rightarrow \text{p.155}) \alpha \mid \alpha :: \kappa \mid (\tau, \dots, \tau) T \quad (\alpha \text{ is type variable})$

is a grammar defining what polymorphic types are (syntactically). As before (\rightarrow p.173), 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

Type Assignment Calculus with Type Classes

Assume some syntax for declaring $\tau :: \kappa$ (\rightarrow p.194) and $\kappa \prec \kappa'$ (\rightarrow p.192). In addition introduce the rule

$$\frac{\tau :: \kappa \quad \kappa \prec \kappa'}{\tau :: \kappa'} \text{subclass}$$

Type assignment rules as before (\rightarrow p.187), but type substitution Θ in

$$\frac{c : \tau \in (\rightarrow \text{p.176})\Sigma}{\Gamma \vdash c : \tau\Theta} \text{assum}$$

must respect class constraints (\rightarrow p.192): for each $\alpha :: \kappa$ occurring in τ where $\alpha\Theta = \sigma$, judgement (\rightarrow p.195) $\sigma :: \kappa$ must hold.

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 (\rightarrow p.192) $:: \kappa$ (recall (\rightarrow p.195) 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 (\rightarrow p.192).

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 (\rightarrow p.196).

Example

Suppose that by virtue of declarations, we have $\mathbb{N} :: \textit{order}$, $\textit{order} \prec \textit{ord}$, and $\leq : \alpha :: \textit{ord} \rightarrow \alpha \rightarrow \textit{bool} \in \Sigma$. Derive

$$\frac{\mathbb{N} :: \textit{order} \quad \textit{order} \prec \textit{ord}}{\mathbb{N} :: \textit{ord}} \textit{subclass}$$

and then $(\Theta = [\alpha \leftarrow \mathbb{N}])$

$$\frac{(\leq : (\alpha :: \textit{ord}) \rightarrow \alpha \rightarrow \textit{bool}) \in \Sigma}{\vdash \leq : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \textit{bool}} \textit{assum}$$

which respects the class constraint since the judgement $\mathbb{N} :: \textit{ord}$ was derived above.

8.4 Higher-Order Unification

The λ -calculus is “the” (\rightarrow p.151) metalogic. Hence we now (sometimes) call its variables “metavariables” (\rightarrow p.31) for emphasis and we precede them with “?”. E.g. they can stand for object (\rightarrow p.31)-level formulae (\rightarrow p.21). More details later (\rightarrow p.204).

Two issues concerning metavariables are:

- suitable renamings²⁰⁵ of metavariables;
- unification²⁰⁶ before rule application.

²⁰⁵Whenever 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.

²⁰⁶The mechanism to instantiate metavariables as needed is called (higher-order) unification. Unification is the process of finding a substitution (\rightarrow p.159) that makes two terms equal.

We will now see more formally what it is and later also where it is used (\rightarrow p.236).

What Is Higher-Order Unification?

Unification of terms e, e' : find substitution (\rightarrow p.157) θ for metavariables such that $e\theta =_{\alpha\beta\eta} e'\theta$.

Examples²⁰⁷:

$$\begin{aligned} ?X + ?Y &=_{\alpha\beta\eta} x + x \\ \textcolor{red}{?P}(x) &=_{\alpha\beta\eta} x + x \\ f(?X\ x) &=_{\alpha\beta\eta} ?Y\ x \\ ?F(?G\ x) &=_{\alpha\beta\eta} f(g(x)) \end{aligned}$$

Why higher-order (\rightarrow p.228)? Metavariables may be instantiated to functions, e.g. $[?P \leftarrow \lambda y.y + y]$.

207

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 $f(?X\ x) =_{\alpha\beta\eta} ?Y\ x$ is $[?X \leftarrow (\lambda z.z), ?Y \leftarrow (\lambda z.f\ z)]$.

Three solutions for $?F(?G\ x) =_{\alpha\beta\eta} f(g(x))$ are

$$\begin{aligned} &[?F \leftarrow f, ?G \leftarrow g], \\ &[?F \leftarrow (\lambda x.f(g\ x)), ?G \leftarrow (\lambda x.x)], \\ &[?F \leftarrow (\lambda x.x), ?G \leftarrow (\lambda x.f(g\ x))], \end{aligned}$$

Higher-Order Unification: Facts

- Unification modulo²⁰⁸ $\alpha\beta$ (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 (\rightarrow p.286)) are decidable.
- HO-unification has possibly infinitely many solutions.

We will look at some of these issues again later (\rightarrow p.276).

²⁰⁸Unification 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 $\alpha\beta\eta$ means finding a substitution σ for metavariables such that $\sigma(e) =_{\alpha\beta\eta} \sigma(e')$.

8.5 Summary on λ -Calculus

λ -calculus is a formalism for writing functions (\rightarrow p.152).

β -reduction (\rightarrow p.163) is the notion of “computing” in λ -calculus.

λ -calculus is Turing-complete (\rightarrow p.169).

λ^\rightarrow (\rightarrow p.170) restricts syntax to “meaningful” λ -terms.

Extension of typed λ -calculus (\rightarrow p.184) used to represent syntax of object logics. λ -terms²⁰⁹ stand for object terms/-formulae, possibly containing “distinguished occurrences” (\rightarrow p.111) of (object) variables. This will be explained thoroughly next lecture (\rightarrow p.204).

HO-unification (\rightarrow p.200) important in constructing proofs.

²⁰⁹So just like first-order logic (\rightarrow p.69), 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 (\rightarrow p.301), we have no syntactic category called “formula” for the λ -calculus.

9 Encoding Syntax

Metatheory: Motivation

Previously (\rightarrow p.150), we have seen the (polymorphically (\rightarrow p.185)) typed λ -calculus (\rightarrow p.170) (with type classes (\rightarrow p.188)).

Now, we will see how the typed λ -calculus can be used as a metalanguage (\rightarrow p.114) for representing²¹⁰ the syntax of an object logic, e.g. first-order logic (\rightarrow p.62).

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 $\ulcorner \phi \urcorner : o$ ²¹¹.

Later (\rightarrow p.296): How do we represent the proofs/provability?

²¹⁰In the following, we will distinguish between the object logic and the metalogic. We have already seen this kind of distinction before (\rightarrow p.31).

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 (\rightarrow p.296) are about.

²¹¹

$\phi \in Prop$ 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 Have a Metalogic?

Why should we have a meta- or framework logic rather than implementing provers for each object logic individually?

- + Implement ‘core’²¹² only once
- + Shared support for automation²¹³
- + Conceptual framework²¹⁴ for exploring what a logic is

But

- + / – Metalayer²¹⁵ between user and logic
- Makes assumptions²¹⁶ about structure of logic

9.1 λ^{\rightarrow} : Review

²¹²By 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.

²¹³There 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.

²¹⁴By implementing various object logics within the same metalogic, we can compare the object logics in a more formal way.

²¹⁵Having 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.

²¹⁶Designing 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

λ^\rightarrow is sufficient for presentation here (no polymorphism (\rightarrow p.185), type classes (\rightarrow p.188)).

- Syntax for types (\mathcal{B} a set of base types (\rightarrow p.170), $T \in \mathcal{B}$)

$$\tau ::= T \mid \tau \rightarrow \tau \quad (\rightarrow \text{ p.155})$$

Examples: \mathbb{N} , $\mathbb{N} \rightarrow \mathbb{N}$ (\rightarrow p.171), $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$, $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ (\rightarrow p.171)

- Syntax for terms: λ -calculus (\rightarrow p.154) augmented with types (\rightarrow p.171)

$$e ::= (\rightarrow \text{ p.155}) \ x \mid c \mid (ee) \mid (\lambda x^\tau. e)$$

$$(x \in \text{Var}, c \in \text{Const} \quad (\rightarrow \text{ p.172}))$$

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].

Type Assignment

- Signature (\rightarrow p.173) $\Sigma ::= \langle \rangle \mid \Sigma, c : \tau$ (\rightarrow p.173).
- Context (\rightarrow p.173) $\Gamma ::= \langle \rangle \mid \Gamma, x : \tau$ (\rightarrow p.173).
- Type assignment rules (\rightarrow p.175)

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau} \text{assum} \quad \Gamma, x : \tau, \Delta \vdash x : \tau \text{hyp}$$

$$\frac{\Gamma \vdash e : \sigma \rightarrow \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash ee' : \tau} \text{app} \quad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x^\sigma. e : \sigma \rightarrow \tau} \text{abs}$$

9.2 Representing Syntax of Propositional Logic

Let the following fragment of propositional logic²¹⁷, which we call *Prop*, be our object logic (\rightarrow p.205):

$$P ::= (\rightarrow \text{p.155}) x \mid \neg P \mid P \wedge P \mid P \rightarrow P$$

Let λ^{\rightarrow} ²¹⁸ be our metalogic (\rightarrow p.205). Declare

- $\mathcal{B} = \{o\}$ (\rightarrow p.205).
- Signature (\rightarrow p.173) assigns types to constants²¹⁹:

$$\Sigma = \langle not : o \rightarrow o, and : o \rightarrow o \rightarrow o, imp : o \rightarrow o \rightarrow o \rangle$$

²¹⁷We consider here the fragment of propositional logic containing the logical symbols (\rightarrow p.105) $\neg, \wedge, \rightarrow$, 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 (\rightarrow p.20).

²¹⁸We 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 (\rightarrow p.229).

²¹⁹Now 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 metalogic (\rightarrow p.205) λ^{\rightarrow} , *not*, *and*, and *imp* are constants (\rightarrow p.173). However, these constants represent the logical symbols (\rightarrow p.105) of the object logic.

- Context (\rightarrow p.173) assigns types to variables²²⁰.

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 (\rightarrow p.164) propositions and returns a proposition.

²²⁰We 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., $a : o \rightarrow o$.

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 (\rightarrow p.602) 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

$a : o \vdash \text{imp} (\text{not } a) a : o$ ²²¹

$$\frac{\frac{a : o \vdash \text{imp} : o \rightarrow o \rightarrow o \quad \frac{a : o \vdash \text{not} : o \rightarrow o \quad a : o \vdash a : o}{a : o \vdash \text{not } a : o}}{a : o \vdash \text{imp} (\text{not } a) : o \rightarrow o} \quad a : o \vdash a : o}{a : o \vdash \text{imp} (\text{not } a) a : o}$$

Applications of *hyp* (\rightarrow p.180) and *assum* (\rightarrow p.182) suppressed. Otherwise always rule *app* (\rightarrow p.175).

²²¹ $a : o \vdash \text{imp} (\text{not } a) a : o$ is a judgement (\rightarrow p.174) 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.

Non-example of First-Order Syntax

$$a : o \vdash \text{not } (\text{imp } a) a : o^{222}$$

$$\frac{a : o \vdash \text{not } : o \rightarrow o \quad \frac{a : o \vdash \text{imp } : o \rightarrow o \rightarrow o \quad a : o \vdash a : o}{a : o \vdash \text{imp } a : o \rightarrow o}}{???}$$

No proof possible! (Requires analysis of normal forms²²³.)

²²² $a : o \vdash \text{not } (\text{imp } a) a : o$ is a judgement (\rightarrow p.174) 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 $\text{not } (\text{imp } a) a$.

²²³Generally, it is difficult to prove that a proof of a given judgement within a given proof system (\rightarrow p.16) 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

Bijection between *Prop* and *o*

We desire bijection²²⁴ $\ulcorner \cdot \urcorner : Prop \rightarrow o$ that is

- adequate: each proposition in *Prop* can be represented by a λ^\rightarrow -term of type *o*:

If $P \in Prop$ then $\Gamma \vdash \ulcorner P \urcorner : o$

- faithful: each λ^\rightarrow term of type *o* represents a proposition in *Prop*:

If $\Gamma \vdash t : o$ then $\ulcorner t \urcorner^{-1} \in Prop$

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].

²²⁴In 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 $a \in A$.

Adequacy of Bijection

Example: $(\neg a) \rightarrow b \in Prop$ therefore $imp (not\ a)\ b : o$

Formalize mapping $\ulcorner \cdot \urcorner$:

$$\begin{aligned}\ulcorner x \urcorner &= x && \text{for } x \text{ a variable} \\ \ulcorner \neg P \urcorner &= not\ \ulcorner P \urcorner \\ \ulcorner P \wedge Q \urcorner &= and\ \ulcorner P \urcorner\ \ulcorner Q \urcorner \\ \ulcorner P \rightarrow Q \urcorner &= imp\ \ulcorner P \urcorner\ \ulcorner Q \urcorner\end{aligned}$$

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 induction²²⁵.

²²⁵If $P \in Prop$, 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* (\rightarrow p.175).

Induction step: Suppose the claim holds for $P \in Prop$ and $Q \in Prop$.

Consider the propositional formula $\neg P$. We have $\ulcorner \neg P \urcorner = not\ \ulcorner P \urcorner$. 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 $\Gamma \vdash not : o \rightarrow o$ by rule *assum* (\rightarrow p.175), and so $\Gamma \vdash not\ \ulcorner P \urcorner : o$ by rule *app* (\rightarrow p.175).

Now consider the propositional formula $P \wedge Q$. We have $\ulcorner P \wedge Q \urcorner = and\ \ulcorner P \urcorner\ \ulcorner Q \urcorner$. 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 $\Gamma \vdash \ulcorner Q \urcorner : o$. More-

Faithfulness of Bijection

Define $\ulcorner \cdot \urcorner^{-1}$

$$\begin{aligned}\ulcorner x \urcorner^{-1} &= x && \text{for } x \text{ a variable} \\ \ulcorner \text{not } P \urcorner^{-1} &= \neg \ulcorner P \urcorner^{-1} \\ \ulcorner \text{and } P \text{ } Q \urcorner^{-1} &= \ulcorner P \urcorner^{-1} \wedge \ulcorner Q \urcorner^{-1} \\ \ulcorner \text{imp } P \text{ } Q \urcorner^{-1} &= \ulcorner P \urcorner^{-1} \rightarrow \ulcorner Q \urcorner^{-1}\end{aligned}$$

For bijection (\rightarrow p.214), should have $\ulcorner \ulcorner P \urcorner^{-1} \urcorner^{-1} = P$ and $\ulcorner \ulcorner t \urcorner^{-1} \urcorner^{-1} = t$. Former is trivial²²⁶, but what about latter?

over $\Gamma \vdash \text{and} : o \rightarrow o \rightarrow o$ by rule *assum* (\rightarrow p.175), and so $\Gamma \vdash \text{and } \ulcorner P \urcorner^{-1} \ulcorner Q \urcorner^{-1} : o$ by two applications of rule *app* (\rightarrow p.175).

The case $P \rightarrow Q$ is completely analogous.

²²⁶By the definition of *Prop* (\rightarrow p.209) and the definition of $\ulcorner \cdot \urcorner$ (\rightarrow p.215), it is clear that $\ulcorner P \urcorner$ is defined for all $P \in \text{Prop}$. It is very easy to show by induction on *Prop* that $\ulcorner \ulcorner P \urcorner^{-1} \urcorner^{-1} = P$.

Here is an example of a proof by induction on *Prop*. (\rightarrow p.215)

Obviously, everything we say here depends on the particular fragment (\rightarrow p.209) of propositional logic, but in an inessential way. It would be trivial to adapt to other fragments.

$\ulcorner t \urcorner^{-1}$ Is not Total

Example: For $t = \text{not}((\lambda x^o. x)a)$, we have $a : o \vdash t : o$

$$\frac{\frac{\frac{a : o, x : o \vdash x : o}{a : o \vdash \lambda x^o. x : o \rightarrow o} \text{abs} \quad a : o \vdash a : o}{a : o \vdash (\lambda x^o. x) a : o} \text{app} \quad a : o \vdash \text{not} : o \rightarrow o}{a : o \vdash \text{not}((\lambda x^o. x) a) : o} \text{app}$$

But $\ulcorner t \urcorner^{-1}$ is undefined!

Normal Forms

If $t : o$, then there exists a t' such that $t =_{\beta\eta} (\rightarrow \text{p.165})t'$, where $t' : o$ and t' is in canonical ($\beta\eta$ -long) normal²²⁷ form, e.g.

$$\begin{aligned} \text{not } ((\lambda x^o. x) a) &=_{\beta\eta} \text{not } a \\ \text{not} &=_{\beta\eta} \lambda x^o. \text{not } x \\ \text{imp } (\text{not } ((\lambda x^o. x) a)) &=_{\beta\eta} \lambda x^o. \text{imp } (\text{not } a) x \end{aligned}$$

227

A canonical $\beta\eta$ -long normal form of a λ -term is obtained by applying first β -reduction as long as possible, and then computing the maximal η -expansion ($\rightarrow \text{p.165}$).

You may wonder: Why is there such a thing as a maximal η -expansion? Can't I expand a λ -term to $\lambda x_1 \dots x_n. M x_1 \dots x_n$ for arbitrary n ? In the untyped λ -calculus, this is indeed the case. But in the typed λ -calculus, the answer is no! Consider this example:

not can be expanded to $\lambda x. \text{not } x$ since not is of function type: it has type $o \rightarrow o$ ($\rightarrow \text{p.209}$). Therefore, $\text{not } x$ can be assigned a type ($\rightarrow \text{p.175}$), which is an intermediate step in typing $\lambda x. \text{not } x$:

$$\frac{\Gamma, x : o \vdash \text{not} : o \rightarrow o \quad \Gamma, x : o \vdash x : o}{\Gamma, x : o \vdash \text{not } x : o} \text{ app} \\ \frac{\Gamma, x : o \vdash \text{not } x : o}{\Gamma \vdash \lambda x. \text{not } x : o \rightarrow o} \text{ abs}$$

But we cannot, say, expand not to $\lambda xy. \text{not } x y$ since it is impossible to assign a type to $\text{not } x y$.

Bijection Theorem

The encoding $\ulcorner \cdot \urcorner$ is a bijection between propositional formulae with variables in Γ ²²⁸ and canonical terms t' , where $\Gamma \vdash t' : o$.

Proof: Based on normalization (\rightarrow p.213)

$$\frac{\frac{x : \sigma \vdash e : \tau}{\vdash \lambda x^\sigma. e : \sigma \rightarrow \tau} \text{abs} \quad \vdash e' : \sigma}{\vdash (\lambda x^\sigma. e) e' : \tau} \text{app} \quad \Rightarrow^{229} \quad \vdash e[x \leftarrow e'] : \tau$$

Corollary: If $t : o$ ²³⁰ then $t =_{\beta\eta} t'$ and $\ulcorner t' \urcorner^{-1} \in Prop$ for some canonical t' .

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 (\rightarrow p.168).

²²⁸Saying 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 $x : o \in \Gamma$.

²²⁹What this picture says is that if the left hand side is a fragment from a proof tree, deriving the judgement (\rightarrow p.174) $\vdash (\lambda x^\sigma. e) e' : \tau$, 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.

²³⁰Simply 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 $Prop$.

9.3 Representing Syntax of First-Order Logic

In *Prop*, we only have the syntactic category (\rightarrow p.21) of formulae (propositions), represented in λ^\rightarrow by the type o (\rightarrow p.205).

In first-order²³¹ logic, we also have the syntactic category (\rightarrow p.69) 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 (\rightarrow p.212), $\Gamma \vdash t : i$ means that t represents a term.

²³¹In the previous section (\rightarrow p.209), 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 (\rightarrow p.209) 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 (\rightarrow p.229) 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.

Example: First-Order Arithmetic (FOA)

Following fragment of FOA is our object (\rightarrow p.205) level language²³²:

Terms $T ::= (\rightarrow$ p.155) $x \mid 0 \mid s^{233} T \mid T + T \mid T \times T$

Formulae $F ::= T = T \mid \neg F \mid F \wedge F \mid F \rightarrow F$

In λ^\rightarrow (on metalevel), define signature (\rightarrow p.209) $\Sigma = \Sigma_{\mathcal{F}}^{234} \cup \Sigma_{\mathcal{P}} \cup \Sigma_{\mathcal{C}}$:

$\Sigma_{\mathcal{F}} = \langle zero : i, succ : i \rightarrow i, plus : i \rightarrow i \rightarrow i, times : i \rightarrow i \rightarrow i \rangle$

$\Sigma_{\mathcal{P}} = \langle eq : i \rightarrow i \rightarrow o \rangle$

$\Sigma_{\mathcal{C}} = \langle not : o \rightarrow o, and : o \rightarrow o \rightarrow o, imp : o \rightarrow o \rightarrow o \rangle$

²³²With 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 $\mathcal{F} = \{0, s, +, \times\}$ (\rightarrow p.69) and $\mathcal{P} = \{=\}$ (\rightarrow p.69). However, the way we defined first-order logic (\rightarrow p.71), 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.

²³³ s is a unary prefix (\rightarrow p.67) function, so s applied to T is written sT .

²³⁴We have defined

$\Sigma_{\mathcal{F}} = \langle zero : i, succ : i \rightarrow i, plus : i \rightarrow i \rightarrow i, times : i \rightarrow i \rightarrow i \rangle$

$\Sigma_{\mathcal{P}} = \langle eq : i \rightarrow i \rightarrow o \rangle$

$zero : i$ means: viewed on the object level, 0 is a term.

$plus : i \rightarrow i \rightarrow i$ means: viewed on the object level, $plus$ is a function that takes two (\rightarrow p.164) terms and returns a term. $eq : i \rightarrow i \rightarrow o$ means: viewed on the object level, $=$

Example: $\ulcorner x + s\ 0 \urcorner^{235} = \textit{plus } x (\textit{succ } zero)$.

is a predicate that takes two (\rightarrow p.164) terms and returns a proposition.

On the metalevel (level of λ^{\rightarrow}), *zero*, *plus* and *eq* are constants. Note that we could also formalize them as variables.

Recall that we encoded the non-logical (\rightarrow p.105) 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 $\Gamma_{\mathcal{F}} \cup \Gamma_{\mathcal{P}}$ and instead of our $\Sigma_{\mathcal{F}} \cup \Sigma_{\mathcal{P}}$. This is in line with Perlis' epigram (\rightarrow p.182). We will sometimes take this approach in the exercises as the encoding of λ^{\rightarrow} in Isabelle makes it more straightforward to play around with different Γ 's than with different Σ 's.

²³⁵We extend the definition of $\ulcorner \cdot \urcorner$ (\rightarrow p.215) as follows:

$$\begin{aligned}\ulcorner x \urcorner &= x \\ \ulcorner 0 \urcorner &= \textit{zero}\end{aligned}$$

Encoding FOL in General

In general, to encode some first-order language (\rightarrow p.69), we must define $\Sigma_{\mathcal{F}}$ and $\Sigma_{\mathcal{P}}$ so that for each n -ary $f \in \mathcal{F}$, $p \in \mathcal{P}$

$$f_{enc} : \underbrace{i \rightarrow \dots \rightarrow i}_{n \text{ times}} \rightarrow i \in \Sigma_{\mathcal{F}},$$

$$p_{enc} : \underbrace{i \rightarrow \dots \rightarrow i}_{n \text{ times}} \rightarrow o \in \Sigma_{\mathcal{P}},$$

and then $\lceil f(t_1, \dots, t_n) \rceil = f_{enc} \lceil t_1 \rceil \dots \lceil t_n \rceil$ and $\lceil p(t_1, \dots, t_n) \rceil = p_{enc} \lceil t_1 \rceil \dots \lceil t_n \rceil$.

Abusing notation, we might skip the subscript *enc*.

$$\begin{aligned} \lceil s \ t \rceil &= succ \ \lceil t \rceil \\ \lceil r + t \rceil &= plus \ \lceil r \rceil \ \lceil t \rceil \\ \lceil r \times t \rceil &= times \ \lceil r \rceil \ \lceil t \rceil \end{aligned}$$

Note that here, on the object level, x is a first-order variable (a variable is a term (\rightarrow p.64)), and hence on the metalevel, it has type i (\rightarrow p.220).

Quantifiers in First-Order Syntax

Along the same lines (\rightarrow p.221), one might suggest

$$all : var \rightarrow o \rightarrow o, \quad \text{so} \quad \ulcorner \forall x. P \urcorner = all\ x \ulcorner P \urcorner$$

But this approach has some problems:

- Variables are also terms, so “ $var \subseteq i$ ”²³⁶? No subtyping!
- *all* is not a binding operator (\rightarrow p.72) in λ^\rightarrow . E.g., $(p(x) \wedge \forall x. q(x))[x \leftarrow a]$ cannot be modeled²³⁷ as $(and\ (p\ x)(all\ x\ (q\ x\ a))$.

²³⁶In first-order logic, variables are not a syntactic category (\rightarrow p.69) 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 .

²³⁷There is a notion of substitution (\rightarrow p.159) 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 (\rightarrow p.157).

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,

9.4 Higher-Order Abstract Syntax (HOAS)

Example, full FOA (\rightarrow p.221): $F ::= \dots \forall x. A \mid \exists x. A$
 $\Sigma = \Sigma_{\mathcal{F}} (\rightarrow \text{p.221}) \cup \Sigma_{\mathcal{P}} (\rightarrow \text{p.221}) \cup \Sigma_{\mathcal{C}} (\rightarrow \text{p.221}) \cup \Sigma_{\mathcal{Q}}$:

$$\Sigma_{\mathcal{Q}} = \langle all : (i \rightarrow o^{238}) \rightarrow o, exists : (i \rightarrow o) \rightarrow o \rangle$$

Extend the definition of $\ulcorner \cdot \urcorner$ (\rightarrow p.215):

$$\begin{aligned} \ulcorner \forall x. P \urcorner &= all (\lambda x^i. \ulcorner P \urcorner) \\ \ulcorner \exists x. P \urcorner &= exists (\lambda x^i. \ulcorner P \urcorner) \end{aligned}$$

should the terms of type *var* be variables or constants?

One could imagine that they are variables. This means that the signature Σ (\rightarrow p.173) 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 $(and (p x)(all x (q x)))$ could only be typed in a context Γ containing $x : var$.

Alternatively, one could imagine that they are constants. The signature Σ (\rightarrow p.173) would contain expressions of the form $x : var$, 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 meta-level is of no use for implementing substitution on the object level.

²³⁸Some intuition: a proposition is represented by a term of type *o*. Now a term of type $i \rightarrow o$ represents a proposition

Adequacy and faithfulness as before²³⁹.

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 (\rightarrow p.231).

all is a constant which can be applied to a term of type $i \rightarrow o$.

²³⁹Terms and formulae are represented by (canonical) members of i and o . The principle is similar as for *Prop* (\rightarrow p.214).

Examples

$$\begin{aligned} \lceil \forall x. x = x \rceil (\rightarrow \text{p.225}) &= all(\lambda x^i. eq\ x\ x (\rightarrow \text{p.225})) \\ \lceil \forall x. \exists y. \neg(x + x = y) \rceil (\rightarrow \text{p.225})^\top &= \\ &all(\lambda x^i. exists(\lambda y^i. not\ (eq\ (plus\ x\ x)\ y))) \end{aligned}$$

Example derivation (all but one steps use rule *app* (\rightarrow p.175)):

$$\frac{\frac{\frac{x : i \vdash eq : i \rightarrow i \rightarrow o \quad x : i \vdash x : i}{x : i \vdash eq\ x : i \rightarrow o} \quad x : i \vdash x : i}{x : i \vdash eq\ x\ x : o} \quad \frac{}{\vdash \lambda x^i. eq\ x\ x : i \rightarrow o} \text{abs}}{\vdash all : (i \rightarrow o) \rightarrow o} \quad \vdash all(\lambda x^i. eq\ x\ x) : o$$

Order

Order of a type: For type τ written $\tau_1 \rightarrow \dots \rightarrow \tau_n$, right associated (\rightarrow p.171), $\tau_n \in \mathcal{B}$:

- $Ord(\tau) = 0$ if $\tau \in \mathcal{B}$, i.e., if $n = 1$;
- $Ord(\tau) = 1 + \max(Ord(\tau_i))$,

Intuition: “functions as arguments”²⁴⁰.

A type of order 1 is first-order, of order 2 second-order etc.

A type of order > 1 is called higher order.

²⁴⁰A term of first-order type is a function taking (an arbitrary number of (\rightarrow p.164)) arguments all of which must be of base type.

A term of second-order type is a function taking (an arbitrary number of (\rightarrow p.164)) 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 (\rightarrow p.164)) 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.

Why “Higher Order”?

Constants representing propositional operators (\rightarrow p.209) (logical symbols) or non-logical symbols (\rightarrow p.221) are first-order (hence first-order syntax):

$$and : o \rightarrow o \rightarrow o$$

Variable binding operators are higher-order (hence higher-order syntax):

$$all : (i \rightarrow o) \rightarrow o$$

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 (\rightarrow p.160) λ -operator.

Thanks to this technique, standard operations on syntax need no special encoding (\rightarrow p.224), but are supported implicitly by λ^\rightarrow .

We will now see this.

Binding

Binding (\rightarrow p.72) on the object level and metalevel coincide.

So in $\forall x.P$, all occurrences of x in P are bound, and likewise, in $all(\lambda x^i.\ulcorner P \urcorner)$, all occurrences of x in $\ulcorner P \urcorner$ are bound.

This provides support for substitution (\rightarrow p.224).

Substitution

Recall rules for \forall (\rightarrow p.81):

$$\frac{\forall x. P(x)}{P(t)} \forall\text{-E} \quad \rightsquigarrow \quad \frac{all\ P}{P(t)} \forall\text{-E}$$

$$\frac{\forall x. x = x}{0 = 0x = x[x \leftarrow 0]} \forall\text{-E} \quad \rightsquigarrow \quad \frac{all\ (\lambda x^i. eq\ x\ x)}{eq\ zero\ zero(\lambda x^i. eq\ x\ x)\ zero} \forall\text{-E}$$

Now apply substitution...

Now apply β -reduction...

We now understand “marked positions in a formula” (\rightarrow p.113).

Equivalence under Bound Variable Renaming

On the object level, formulae are equivalent under renaming of bound variables:

$$(\forall x. P \leftrightarrow (\rightarrow \text{p.88}) \forall y. P[x \leftarrow y])$$

Likewise, on the metalevel, formulae obtained by bound variable renaming are α -equivalent (\rightarrow p.165):

$$all(\lambda x^i. P) =_{\alpha} all(\lambda y^i. P[x \leftarrow y])$$

9.5 Summary of Encoding Syntax

Object Language	Metalinguage
Syntactic category (\rightarrow p.220)	Type declaration (\rightarrow p.220)
$Term, Prop$	$\{i, o\} = \mathcal{B}$
Variable x	Variable ²⁴¹ x
Non-logical	1st-order constant (\rightarrow p.221)
symb. (\rightarrow p.221) +	$plus : i \rightarrow i \rightarrow i$
Logical	1st-order constant (\rightarrow p.209)
symbol (\rightarrow p.209) \wedge	$and : o \rightarrow o \rightarrow o$
Binding	2nd-order const. (\rightarrow p.225)
operator (\rightarrow p.225)	$all : (i \rightarrow o) \rightarrow o$
\forall	
Meaningful expr. (\rightarrow p.209)	Member of type (\rightarrow p.212)
$a \wedge b \in Prop$	$(and\ a\ b) : o$

²⁴¹Although propositional variables (\rightarrow p.18) and first-order variables (\rightarrow p.64) 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 (\rightarrow p.210) and first-order variables (\rightarrow p.222). 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 .

10 Resolution

Three Sections on Deduction Techniques

After encoding syntax (\rightarrow p.204), the next topic in the theory is encoding proofs (\rightarrow p.296).

But before, we look at some more practical issues:

- Resolution (\rightarrow p.236)
- Proof search (\rightarrow p.249)
- Term rewriting (\rightarrow p.276)

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 (\rightarrow p.200) a certain part of the current goal (state) with a certain part of a rule, and replacing that part of the current goal.

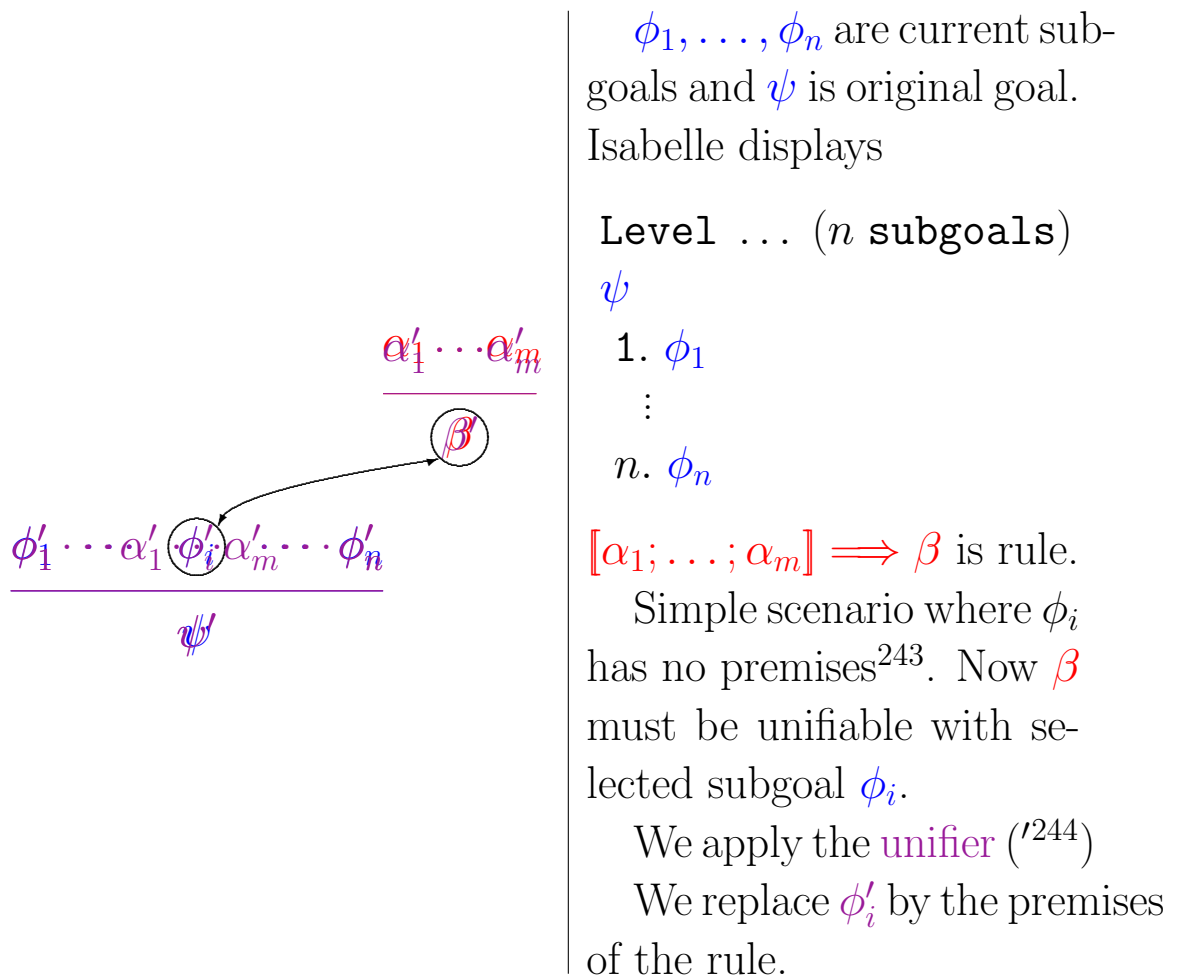
We have already explained this in the labs and you have been working with it all the time, but now we want to understand it more thoroughly (in the next lecture (\rightarrow p.296), we will look at it more abstractly).

We look at several variants of resolution.

Resolution (rtac, as in Prolog²⁴²)

²⁴²Prolog 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 $[\alpha_1; \dots; \alpha_m] \implies \beta$).



²⁴³ 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 \Rightarrow (metalevel implication). The form of the other subgoals $\phi_1, \dots, \phi_{i-1}, \phi_{i+1}, \dots, \phi_n$ is arbitrary.

²⁴⁴In all illustrations that follow, we use ' to suggest the application of the appropriate unifier.

Resolution (with Lifting over Parameters)

$$\begin{array}{c}
 \frac{\bigwedge x.\alpha'_1[x] \cdots \bigwedge x.\alpha'_m[x]}{\bigwedge x.\beta[x]} \\
 \\
 \frac{\phi'_1 \cdots \bigwedge x.\alpha'_1[\bigwedge x.\phi_i]x.\alpha'_m[x] \cdots \phi'_n}{\psi}
 \end{array}$$

Now suppose the i 'th (selected) subgoal is preceded by \bigwedge (metalevel universal quantifier²⁴⁵).

Rule is lifted²⁴⁶ 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 $\bigwedge x$.

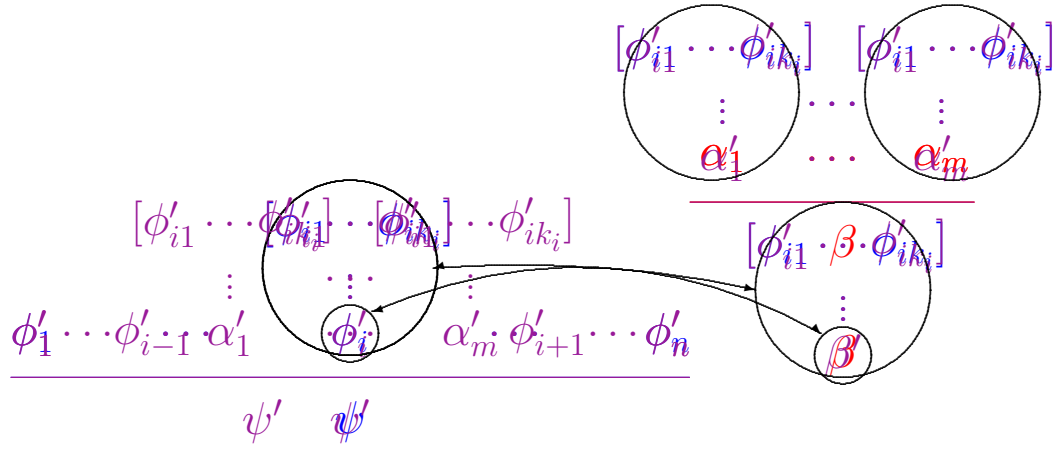
²⁴⁵ \bigwedge is the metalevel universal quantification (also written !!). If a goal is preceded by $\bigwedge x$, this means that Isabelle must be able to prove the subgoal in a way which is independent from x , i.e., without instantiating x .

²⁴⁶The 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 .

Resolution (with Lifting over Assumptions)



Now, suppose the i 'th (selected) subgoal has assumptions $\phi_{i1}, \dots, \phi_{ik_i}$.

As before, we have a rule. In general β is not²⁴⁷ unifiable with the i 'th subgoal, even assuming that β is unifiable with ϕ_i .

Rule must be lifted over assumptions²⁴⁸. No unification so far!

Now, subgoal and rule conclusion (below the bar) are unifiable²⁴⁹.

Non-trivially²⁵⁰, β must be unifiable with ϕ_i .

We apply the **unifier**.

We replace the subgoal.

²⁴⁷The 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 (\rightarrow p.194) 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”.

²⁴⁸Each 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

$$\begin{array}{ccc}
 [\phi_{i1} \cdots \phi_{ik_i}] & & [\phi_{i1} \cdots \phi_{ik_i}] \\
 \vdots & \cdots & \vdots \\
 \alpha_1 & \cdots & \alpha_m \\
 \hline
 [\phi_{i1} \cdots \phi_{ik_i}] \\
 \vdots \\
 \beta
 \end{array}$$

may look different from any rules you have seen so far, but it can be formally derived from the rule:

$$\begin{array}{ccc}
 \alpha_1 & \cdots & \alpha_m \\
 \hline
 \beta
 \end{array}$$

²⁵⁰Both 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.

Rule Premises Containing \Rightarrow

$$\frac{\begin{array}{c} [\phi'_{i_1} \cdots \phi'_{i_{k_i}}] \gamma'_1 \cdots \gamma'_l \\ \vdots \\ \phi'_1 \cdots [\gamma_1; \dots; \gamma_l] \Rightarrow \delta \cdots \phi'_n \end{array}}{\psi'}$$

What if some α'_j has the form $[\gamma_1; \dots; \gamma_l] \Rightarrow \delta$?

Is this what we get?

Well, we write : for \Rightarrow , and use $A \Rightarrow B \Rightarrow C \equiv [A; B] \Rightarrow C$ ²⁵¹.

²⁵¹Generally, Isabelle makes no distinction between

$$[\psi_1; \dots; \psi_n] \Rightarrow [\mu_1; \dots; \mu_k] \Rightarrow \phi$$

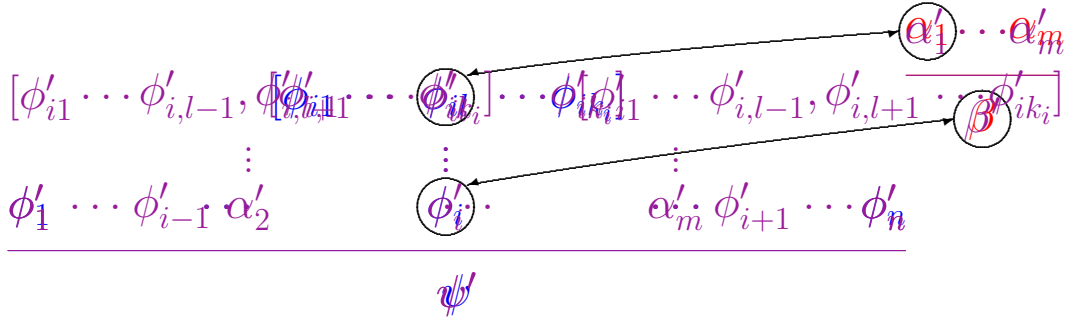
and

$$[\psi_1; \dots; \psi_n; \mu_1; \dots; \mu_k] \Rightarrow \phi$$

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.

Elimination-Resolution



Same scenario as before²⁵², 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 the first²⁵³. $\alpha'_2, \dots, \alpha'_m$ inherit the assumptions of ϕ'_i , except ϕ'_{il} .

²⁵²So the scenario looks as for resolution with lifting over assumptions (\rightarrow p.242). However, this time we do not show the lifting over assumptions in our animation.

²⁵³Elimination-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

$$\frac{\begin{array}{c} [P; Q] \\ \vdots \\ P \wedge Q \end{array} \quad R}{R} \wedge\text{-E}$$

Destruct-Resolution

$$\begin{array}{c}
 \begin{array}{c} \alpha' \end{array} \\
 \swarrow \\
 \frac{[\phi'_{i1} \cdots \phi'_{il} \cdots \phi'_{ik_i}] \quad \beta}{\psi'}
 \end{array}$$

\vdots
 $\phi'_1 \quad \cdots \quad \phi'_i \quad \cdots \quad \phi'_n$

Simple rule, and α must be unifiable with ϕ_{il} , for some l .

We apply the **unifier**.

We replace premise²⁵⁴ ϕ'_{il} with the conclusion of the rule.

10.1 Summary on Resolution

- Build proof resembling sequent style notation (\rightarrow p.48);
- technically: replace goals with rule premises, or goal premises with rule conclusions;

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.

²⁵⁴Destruct-resolution is used to eliminate a connective in the premises. The difference compared to elimination-resolution (\rightarrow p.246) can be seen in the following example. Unlike elimination-resolution, destruct-resolution “throws information away”.

- metavariables and unification (\rightarrow p.200) to obtain appropriate instance of rule, delay commitments;
- lifting over parameters (\rightarrow p.241) and assumptions (\rightarrow p.242);
- various techniques to manipulate premises or conclusions, as convenient: **rtac** (\rightarrow p.239), **etac** (\rightarrow p.246), **dtac** (\rightarrow p.247).

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

$$\frac{P \wedge Q}{P} \text{conjunct2}$$

11 Automation by Proof Search

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 .

Outline of this Part

- Proof search (à la tableau proving (\rightarrow p.262)) and backtracking
- Classifying rules (\rightarrow p.260)
- Proof procedures (\rightarrow p.273)

11.1 Proof Search and Backtracking

- Need for more automation²⁵⁵
- Some aspects in proof construction are highly non-deterministic:
 - unification: which unifier (\rightarrow p.200) to choose?

²⁵⁵We 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.

- resolution: where²⁵⁶ to apply a rule (which 'subgoal')?
- which rule to apply?
- How to organize proof-search technically²⁵⁷?

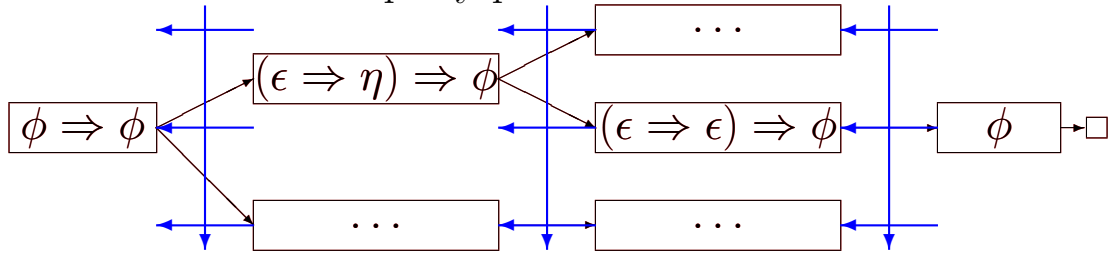
²⁵⁶We have seen in the exercises (and also in the lecture (→ p.236)) that one can choose the subgoal to which one wants to apply a rule.

²⁵⁷We have seen in the previous lecture (→ p.236) 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?

Organizing Proof Search Conceptually

Organize proof search as a tree²⁵⁸ of theorems²⁵⁹ (**thm**'s).

A sketch of an exemplary proof search:



²⁵⁸We have seen in the previous lecture (\rightarrow p.236) 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.

²⁵⁹Technically, a proof state is an Isabelle theorem, (**thm**), i.e. something which Isabelle (\rightarrow p.253) regards as true.

Goal ϕ will create the proof state $\phi \Longrightarrow \phi$ ²⁶⁰.

One tactic step (**by** ...) transforms into proof state $(\epsilon \Rightarrow \eta) \Rightarrow \phi$.

Next tactic step yields dead-end (no tactic application possible).

back(); tries an alternative successor of $(\epsilon \Rightarrow \eta) \Rightarrow \phi$.

Now $(\epsilon \Rightarrow \epsilon) \Rightarrow \phi$ is solvable using **atac**²⁶¹. **qed** (\rightarrow p.253).

Use **undo**(); three times to go to previous proof states.

Use **back**(); to try alternative successor. ...

Summary:

back(); (\rightarrow p.253) to try alternative successors²⁶² (\Rightarrow different unifiers (\rightarrow p.200)).

undo(); (\rightarrow p.253) to go to previous proof state.

²⁶⁰The initial proof state is $\phi \Longrightarrow \phi$. Isabelle (\rightarrow p.253) 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 $\phi \Longrightarrow \phi$ as true. The aim of a proof search is to transform $\phi \Longrightarrow \phi$ (ϕ can be shown if I assume ϕ) into ϕ (ϕ can be shown if I assume nothing).

²⁶¹For more details on Isabelle technicalities, you should consult the reference manual [Pau05].

²⁶²Note 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.

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 (\rightarrow p.200)).
- Explicit tree representation²⁶³ expensive in time and space.

As an aside²⁶⁴, it is also possible to understand proof search more abstractly. But we are interested in the operational aspects.

²⁶³Obviously, 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.

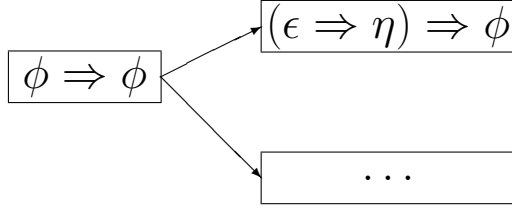
²⁶⁴The explicit tree representation is not very abstract in that each node has a defined order of the children (first successor, second successor, ...). This order is an artefact of the order in which unifiers are enumerated (\rightarrow p.200) by the unification algorithm used. It is inessential for 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)

$$prooftrees = \mathcal{P}(\mathbf{thm} \times \mathbf{thm})$$

More precisely, one can look at a fragment of a tree of theorems as before (\rightarrow p.252), e.g.:

Organizing Proof Search Operationally



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 (\rightarrow p.252), 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 ”)
- $(\phi \Rightarrow \phi, \phi) (\rightarrow \text{p.253}) \in PT^* \quad \equiv \quad \text{“there is a proof for } \phi\text{”}$

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:

Organize proof search as a function on theorems²⁶⁵ (**thm**'s)

type tactic = thm → thm seq

where **seq**²⁶⁶ is the type constructor for infinite lists.

-
- Union \cup is difficult to implement (needs comparison with all previous results since one wants to avoid duplicates).
 - More operational (\rightarrow p.256), strategic interpretations of union \cup are desirable (try this — then that, interleave attempts in PT_1 with attempts in PT_2 , and so forth).

²⁶⁵This way of understanding and organizing proof search is not so abstract (\rightarrow p.255), 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 (\rightarrow p.252).

²⁶⁶For any type τ , the type τ **seq** (recall the notation (\rightarrow p.185)) is the type of (possibly) infinite lists of elements of type τ . This is of course an abstract datatype.

This allows us to have tacticals²⁶⁷:

- **THEN**
- **ORELSE**
- **REPEAT**
- **INTLEAVE, BREADTHFIRST, DEPTHFIRST, ...**

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.

²⁶⁷

- **THEN**
- **ORELSE**
- **REPEAT**
- **INTLEAVE, BREADTHFIRST, DEPTHFIRST, ...**

are called tacticals.

Tacticals are operations on tactics. They play an important role in automating proofs in Isabelle (➔ p.253). 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,

11.2 Classifying Rules

How to Organize Proof Rules?

Observation: Some rules can always be applied blindly in backward reasoning, e.g. \rightarrow -I (\rightarrow p.49) or \wedge -I (\rightarrow p.49). Others are problematic, e.g. \wedge -EL (\rightarrow p.49) or \wedge -ER (\rightarrow p.49) (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 (\rightarrow p.48).

while tac_1 **ORELSE** tac_2 applies tac_1 if possible and otherwise applies tac_2 [Pau05, Ch. 4].

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 calculus²⁶⁸, one writes \wedge -E²⁶⁹ as:

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

This mimics²⁷⁰ the effect of using \wedge -E (**conjE** of Isabelle), here written in sequent style notation (\rightarrow p.48),

$$\frac{\Gamma \vdash A \wedge B \quad A, B, \Gamma \vdash C}{\Gamma \vdash C} \wedge\text{-E} (\rightarrow \text{p.45})$$

in combination with **etac** (\rightarrow p.246). We now derive \wedge -E'.

²⁶⁸Tableau proving is a derivation system [Fit96].

It turns out that the language of tableaux is equivalent to the sequent calculus (\rightarrow p.267) (recall our use of sequent style notation (\rightarrow p.48)) [Pau97]. The techniques Isabelle uses for automating proofs can thereby be understood as tableau proving [Pau97].

²⁶⁹In Isabelle (\rightarrow p.253) notation, it looks as follows:

$$[P \& Q; [P; Q] \Longrightarrow R] \Longrightarrow R$$

(see **IFOL_lemmas.ML** (\rightarrow p.272)).

²⁷⁰That is to say, \wedge -E' behaves for the sequent notation as **conjE+etac** (\rightarrow p.246) behaves for Isabelle.

Deriving $\wedge\text{-}E'$ from $\wedge\text{-}E$

$$\frac{A \wedge B, \Gamma \vdash A \wedge B \quad \frac{A, B, \Gamma \vdash C}{A, B, A \wedge B, \Gamma \vdash C} \text{weaken } (\rightarrow \text{ p.49})}{A \wedge B, \Gamma \vdash C} \wedge\text{-}E'$$

If we replace Γ with $A \wedge B, \Gamma$ (just instantiation (\rightarrow p.31)), then one part holds by the assumption rule (\rightarrow p.49), and we can apply weakening (\rightarrow p.49).

A Proof by Blind Rule Application

$$\frac{\frac{\frac{\rho, \phi, \psi \vdash \phi}{\rho \wedge \phi, \psi \vdash \phi} \wedge\text{-}E' \quad \frac{\rho \wedge \phi \vdash \psi \rightarrow \phi}{\vdash (\rho \wedge \phi) \rightarrow \psi \rightarrow \phi} \rightarrow\text{-}I}{\vdash (\rho \wedge \phi) \rightarrow \psi \rightarrow \phi} \rightarrow\text{-}I$$

The topmost connective is \rightarrow , which asks for $\rightarrow\text{-}I$ (\rightarrow p.52).

Again $\rightarrow\text{-}I$.

To decompose²⁷¹ the assumption $\rho \wedge \phi$, use $\wedge\text{-}E'$ (\rightarrow p.262).

The proof can now be completed by the assumption rule (\rightarrow p.49).

²⁷¹See now that we first derived the rule $\wedge\text{-}E'$ (\rightarrow p.262), 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 $\wedge\text{-}E'$ should be used generally instead of $\wedge\text{-}EL$ or $\wedge\text{-}EL$, 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 (\rightarrow p.246). This is indeed the case for $\wedge\text{-}E'$. We remove the assumption $\phi \wedge \psi$, but we get the two conjuncts ϕ and ψ as assumptions instead.

The rule $\wedge\text{-}E'$ mimics the effect of using $\wedge\text{-}E$ in combination with **etac** (\rightarrow p.246), which you can see by looking again at the exercises on **etac**.

Safe and Unsafe Rules

Combined tactics (\rightarrow p.274) rely on classification of rules, maintained in Isabelle (\rightarrow p.253) data structure **claset**²⁷², and accessed by functions²⁷³ of type **claset** * **thm list** \rightarrow **claset**.

Class:	To add use function:
Safe introduction rules	addSIs
Safe elimination rules	addSEs
Unsafe introduction rules	addIs
Unsafe elimination rules	addEs

²⁷²**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.

²⁷³The functions **addSIs**, **addSEs**, **addIs**, **addEs** are all of type **claset** * **thm list** \rightarrow **claset**. They add rules to the current classifier set. For example, **addSIs** adds a rule as safe introduction rule.

Adapting Rules for Automated Proof Search

As seen for \wedge - E (\rightarrow p.262), rules must be suitably adapted in order to be useful in automated proof search. Another example:

$$\frac{\frac{\frac{\neg\alpha, \alpha, \beta \vdash \beta}{\neg(\alpha \rightarrow \beta), \beta \vdash \alpha} \rightarrow\text{-swap}E^{275}}{\neg(\alpha \rightarrow \beta) \vdash \beta \rightarrow \alpha} \rightarrow\text{-I}}{\vdash (\alpha \rightarrow \beta) \vee (\beta \rightarrow \alpha)} \vee\text{-swap}^{274}$$

Neither \vee - IL nor \vee - IR would work here. Uses classical (\rightarrow p.42) logic.

²⁷⁴The rule \vee -swap is

$$\frac{\neg A, \Gamma \vdash B}{\Gamma \vdash A \vee B} \vee\text{-swap}$$

To derive it you need classical (\rightarrow p.42) 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** (\rightarrow p.272))!

²⁷⁵The rule \rightarrow -swap E is

$$\frac{A, \neg C, \Gamma \vdash B}{\neg(A \rightarrow B), \Gamma \vdash C} \rightarrow\text{-swap}E$$

To derive it you need classical (\rightarrow p.42) 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 (\rightarrow p.267). For dealing with negated formulas in the premises of the current subgoal, introduction rules are combined with **swap** using **etac**.

Principle: Emulate sequent calculus²⁷⁶ with derived rules.

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 [Pau05, Section 11.2].

²⁷⁶The 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 (\rightarrow p.48).

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

Handling Quantifiers

Can derive²⁷⁷ $\forall\text{-}E'$ (\equiv `allE`²⁷⁸) using $\forall\text{-}E$ (\equiv `spec`):

$$\frac{\frac{\forall x.A(x)}{B} \quad \begin{array}{c} \vdots \\ B \end{array}}{B} \forall\text{-}E'\forall\text{-}dupE$$

important technique for combined tactics (\rightarrow p.274).

The sequent calculus inherently relies on classical (\rightarrow p.42) reasoning [Pau05, Ch. 11].

²⁷⁷You should do it in Isabelle. The rule is:

$$\llbracket \text{ALL } x. P(x); P(x) \Longrightarrow R \rrbracket \Longrightarrow R$$

²⁷⁸As 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 $\rightarrow\text{-}E$ 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 (\rightarrow p.274) in Isabelle use derived rules such as \wedge - E (\rightarrow p.262) (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.

What is the difference²⁷⁹ to \exists - E ²⁸⁰?

Problem: $\forall x.A(x)$ may still be needed.

²⁷⁹The difference between

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

and

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

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 (\rightarrow p.270).

²⁸⁰The rule

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

Principle: Introduce duplicating²⁸¹ rules. Turns search

 was derived previously (\rightarrow p.94) (but in Isabelle, it is a basic rule in **IFOL.ML** (\rightarrow p.272)). 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' .

²⁸¹You should recall that elimination rules are used in combination with **etac** (\rightarrow p.246). 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

infinite²⁸²!

Check out `allE` and `all_dupE` in `IFOL_lemmas.ML`²⁸³!

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.

²⁸²Given only the rules so far (in combination with the appropriate tactics, `rtac` and `etac` (\rightarrow p.236), and swapping (\rightarrow p.267)), excluding \forall -*dupE*, the proof search would be finite.

The rule \forall -*dupE* 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.

²⁸³These files should be contained in your Isabelle (\rightarrow p.253) distribution. Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

11.3 Proof Procedures (Simplified)

Tactics in Isabelle (\rightarrow p.253) are performed in order²⁸⁴:

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**

²⁸⁴Tactics in Isabelle (\rightarrow p.253) 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** (\rightarrow p.236), or, if that is not possible, a safe elimination rule with **etac** (\rightarrow p.246). 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** (\rightarrow p.236), or, if that is not possible, an unsafe elimination rule with **etac** (\rightarrow p.246).

Combined Proof Search Tactics (→ p.253)

- **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)

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.

11.4 Summary on Automated Proof Search

- Proof search can be organized as a tree of theorems (\rightarrow p.252).
- Calculi can be set up to facilitate proof search (although this must be done by specialists).
- Combined with search strategies (\rightarrow p.274), 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 interpret²⁸⁵.

²⁸⁵**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 (\rightarrow p.265) wrongly, the search strategy (\rightarrow p.274) was not adequate for the problem, enumeration of unifiers (\rightarrow p.200) in a bad order. Or a combination thereof. Or it might be that too many unsafe steps (\rightarrow p.273) are needed, since **fast_tac** limits their number.

12 Term Rewriting

Outline of this Part

- Higher-order rewriting
- Extensions (\rightarrow p.285): Ordered, pattern, congruence, splitting rewriting
- Organizing simplification rules (\rightarrow p.291)

In this context, a term is a λ -term, since we use the λ -calculus to encode object logics (\rightarrow p.204).

12.1 Higher-Order Rewriting

Motivation:

- Simplification is a very important part of deduction, e.g.:

$$0 + (x + 0)^{286} = x$$

$$[a, b, d] @ [a, b]^{287} = [a, b, d, a, b]$$

- Based on rewrite rules as in functional programming²⁸⁸:

$$x + 0 = x,$$

$$0 + x = x$$

$$[] @ X = X,$$

$$(x :: X) @ Y = x :: (X @ Y)$$

²⁸⁶Simplifying $0 + (x + 0)$ 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.

²⁸⁷Lists are a common datatype in functional programming. $[a, b, d, a, b]$ is a list. Actually, this notation is syntactic sugar (\rightarrow p.21) 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 (\rightarrow p.69). In the λ -calculus, we would speak of a (special kind of) constant (this will become clear later (\rightarrow p.281)).

²⁸⁸For example, the lines

$$\begin{aligned} [] @ X &= X \\ (x :: X) @ Y &= x :: (X @ Y) \end{aligned}$$

Term Rewriting: Foundation

- Recall (\rightarrow p.122): An equational theory consists of rules (\rightarrow p.107)

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

$$\frac{x = y \quad P(x)}{P(y)} \text{ subst } (\rightarrow \text{ p.113})$$

- 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 rules²⁸⁹, i.e. they are applied from left to right.

define the list concatenation function @.

²⁸⁹An equational theory is a formalism based on equational rules of the form $\phi_1 = \psi_1, \dots, \phi_n = \psi_n \Longrightarrow \phi = \psi.$

A term rewriting system (to be defined shortly) is another formalism, based of rewrite rules. They also have the form $\phi_1 = \psi_1, \dots, \phi_n = \psi_n \Longrightarrow \phi = \psi$, 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.

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 $e(t)$ (\rightarrow p.113) (resp. $e'(t)$ (\rightarrow p.113))
 - (b) for a rewrite rule $\phi_1 = \psi_1, \dots, \phi_n = \psi_n \implies \phi = \psi$,
match²⁹⁰ (unify) ϕ against t , i.e., find θ such that $\phi\theta = t$
 - (c) solve²⁹¹ $(\phi_1 = \psi_1, \dots, \phi_n = \psi_n)\theta$
 - (d) replace $e(t)$ (\rightarrow p.113) by $e(\psi\theta)$ (\rightarrow p.113) (resp. $e'(t)$ (\rightarrow p.113) by $e'(\psi\theta)$ (\rightarrow p.113))

²⁹⁰Given two terms s and t , a unifier (\rightarrow p.200) 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).

²⁹¹This means that the procedure is called recursively for the conditions of the rewrite rule.

3. goto 1

This procedure + the rules define a term rewriting system²⁹².

²⁹²The procedure defines a term rewriting system [BN98, Klo93].

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 (\rightarrow p.184) 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

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 (\rightarrow p.166)), e.g. rules $a = b$,

from that, there may be function symbols (\rightarrow p.69) in some object logic. On the metalevel (and hence also for the purpose of term rewriting), these would be constants (\rightarrow p.284).

- There may be predicate symbols (\rightarrow p.69) in some object logic. On the metalevel (and hence also for the purpose of term rewriting), these would be constants (\rightarrow p.284).
- A constant is a λ -term consisting of just one symbol from a set *Const*. Constants (\rightarrow p.284) 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 (\rightarrow p.278) for lists (\rightarrow p.278). 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 (\rightarrow p.278).

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 (\rightarrow p.281) of a term.

One could also imagine that object-level function and pred-

$$a = c^{293}.$$

- Providing criteria for terminating and confluent rule sets is an active research area (see [BN98, Klo93], RTA, ...).

icate symbols are represented as variables, as is done in LF. Recall Perlis' epigram (\rightarrow p.182).

²⁹³For a rewriting system consisting of rules $a = b$, $a = c$, one cannot rewrite $b = c$ to prove the equality, although it holds:

$$\frac{\frac{a = b}{b = a} \text{ sym} \quad a = c}{b = c} \text{ trans}$$

12.2 Extensions of Rewriting

- Symmetric rules are problematic, e.g. ACI:²⁹⁴

$$(x + y) + z = x + (y + z) \quad (\text{A})$$

$$x + y = y + x \quad (\text{C})$$

$$x + x = x \quad (\text{I})$$

- Idea: apply only if replaced term gets smaller w.r.t. some term ordering. In example, if $(y + x)\theta$ (\rightarrow p.280) is smaller than $(x + y)\theta$ (\rightarrow p.280).
- Ordered rewriting solves rewriting modulo ACI²⁹⁵, using derived rules (exercise).

²⁹⁴ACI 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 $+$ (\rightarrow p.281) is written infix (\rightarrow p.67).

²⁹⁵Consider an equational theory consisting only of those rules (apart from *refl*, *sym*, *trans*, *subst* (\rightarrow p.122)). 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 norm²⁹⁶. For example, the inductive definition of a norm $|_|$ (\rightarrow p.286) might include the line:

$$|s + t| := 2|s| + |t|$$

This means that if $|s| > |t|$, then $|s + t| = 2|s| + |t| > 2|t| + |s| = |t + s|$.

Extension: HO-Pattern Rewriting

Rules such as $F(G\ c) = \dots$ ²⁹⁷ lead to highly ambiguous matching (\rightarrow p.280) 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 (\rightarrow p.163); and
- any free (\rightarrow p.158) F in t occurs in a subterm $F\ x_1 \dots x_n$ where the x_i are η -equivalent (\rightarrow p.165) to distinct bound variables.

Matching (unification) (\rightarrow p.280) is decidable, unitary ('unique') and efficient algorithms exist.

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 (\rightarrow p.121)) is used in standard mathematics for the absolute value of a number and is standard for norms as well.

²⁹⁷For 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:

HO-Pattern Rewriting (Cont.)

A rule $\dots \Rightarrow \phi = \psi$ is a HO-pattern rule if:

- ϕ is a HO-pattern;
- all free (\rightarrow p.158) variables in ψ occur also in ϕ ; and
- ϕ is constant-head, i.e. of the form $\lambda x_1 \dots x_m. c p_1 \dots p_n$ (where c is a constant (\rightarrow p.281), $m \geq 0$, $n \geq 0$).

Example:²⁹⁸ $(\forall x. Px \wedge Qx) = (\forall x. Px) \wedge (\forall x. Qx)$

Result: HO-pattern allows for very effective quantifier reasoning.

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 (\rightarrow p.281) very inefficient.

²⁹⁸Further 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 (\rightarrow p.184), and hence also for the sake of term rewriting, \forall, \exists are constants (\rightarrow p.281).

Extensions Related to `if – then – else`

The `if-then-else` construct will play an important role later (\rightarrow p.397). It asks for special rewrite rules.

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` (\rightarrow p.272)) to get a flavor of this. The principle was explained thoroughly before (\rightarrow p.225).

Extension: Congruence Rewriting

Problem :

if A **then** P **else** $Q = \text{if } A \text{ then } P' \text{ else } Q$

where $P = P'$ under condition A

is not a rule²⁹⁹.

Solution in Isabelle (\rightarrow p.253): explicitly admit this extra class of rules (congruence rewriting)

$\llbracket A \Longrightarrow P = P' \rrbracket \Longrightarrow$

if A **then** P **else** $Q = \text{if } A \text{ then } P' \text{ else } Q$

²⁹⁹Rewrite rules (\rightarrow p.279) have the form $\phi_1 = \psi_1, \dots, \phi_n = \psi_n \Longrightarrow \phi = \psi$ (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

if A **then** P **else** $Q = \text{if } A \text{ then } P' \text{ else } Q$

where $P = P'$ under condition A

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 (\rightarrow p.287)).

Solution in Isabelle (\rightarrow p.253): explicitly admit this extra class of rules (case splitting).

12.3 Organizing Simplification Rules

- Standard (HO-pattern conditional ordered rewrite (\rightarrow p.286)) rules;
- congruence rules (\rightarrow p.289);
- splitting rules (\rightarrow p.290).

Isabelle (\rightarrow p.253) data structure: **simpset**³⁰⁰. Some operations³⁰¹:

- `addsimps : simpset * thm list \rightarrow simpset`
- `delsimps : simpset * thm list \rightarrow simpset`
- `addcongs : simpset * thm list \rightarrow simpset`
- `addsplits : simpset * thm list \rightarrow simpset`

³⁰⁰The **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 (\rightarrow p.265).

³⁰¹These function manipulate the simplifier set, in analogy to the classifier set (\rightarrow p.265).

Commutativity (\rightarrow p.285) can be added without losing termination.

How to Apply the Simplifier?

Several versions (\rightarrow p.253) of the simplifier:

- `simp_tac : simpset \rightarrow int \rightarrow tactic`
- `asm_simp_tac : simpset \rightarrow int \rightarrow tactic`
(includes assumptions into `simpset`)
- `asm_full_simp_tac : simpset \rightarrow int \rightarrow tactic`
(rewrites assumptions, and includes them into `simpset`)

Using global³⁰² simplifier sets: `Simp_tac`, `Asm_simp_tac`, `Asm_full_simp_tac`.

³⁰²`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 \rightarrow tactic`)

There are analogous capitalized versions for the tactics of the classical reasoner (\rightarrow p.274).

12.4 Summary on Term Rewriting

Simplifier is a powerful proof tool for

- conditional equational formulas (\rightarrow p.286)
- ACI-rewriting (\rightarrow p.285)
- quantifier reasoning (\rightarrow p.287)
- congruence rewriting (\rightarrow p.289)
- automatic proofs by case splitting (\rightarrow p.290)

Fortunately, failure is quite easy to interpret³⁰³.

³⁰³When 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.

12.5 Summary on Last Three Sections

- Although Isabelle is an interactive proof construction, it is a flexible environment with powerful automated proof procedures.
- For classical (\rightarrow p.42) logic and set theory, tableau (\rightarrow p.262)-like procedures (\rightarrow p.273) like **blast_tac** and **fast_tac** decide many tautologies.
- For equational theories (datatypes, evaluating functional programs, but also higher-order logic (\rightarrow p.359)) **simp_tac** (\rightarrow p.293) decides many tautologies (and is fairly easy to control).

13 Isabelle's Metalogic

Representing Syntax and Proofs

- Previously (\rightarrow p.204), we have seen how the (polymorphically (\rightarrow p.18) typed λ -calculus (\rightarrow p.170) 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} , Pure³⁰⁴, HOL.

This lecture is based on Paulson’s work [Pau89]. It is maybe the most challenging lecture of this course.

³⁰⁴In Isabelle jargon, the metalogic is called Pure.

In this course, we will avoid calling the Isabelle metalogic HOL, although you may find such uses in the literature.

In the literature and in Isabelle formalisations, we find various definitions of higher-order logic (HOL) that differ more or less substantially.

But the important point to remember here is this: The Isabelle metalogic \mathcal{M} we study here is not identical to the logic w (\rightarrow p.359)e will study during the entire second half of this course. And the most important difference between \mathcal{M} and HOL is not in the logics themselves, but in the way we use them:

\mathcal{M} is a (the) metalogic!

HOL is an object logic!

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 reasoning³⁰⁵ in propositional logic (\rightarrow p.12) and nothing else.
- We will now introduce a logic \mathcal{M} . Its proof system (\rightarrow p.16) is small!

³⁰⁵We would formalise the language and the proof system as we did in the first lecture (\rightarrow p.17). Any proofs of soundness and completeness or other meta-properties should be rigorous, but they still resort to natural language.

Proof Techniques = Meta-Theorems

- When constructing proofs, there are
 - aspects that are specific to certain logics and its logical symbols (\rightarrow p.105): the proof rules (\rightarrow p.30);
 - aspects that reflect general principles (\rightarrow p.16) of proof building: making and discharging assumptions, substitution (\rightarrow p.84), side conditions (\rightarrow p.81), resolution (\rightarrow p.2

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.

13.1 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 (\rightarrow p.310).

Some variations are possible (mainly: polymorphism/type classes or not), but those are not so important for us.

\mathcal{M} will be based on λ^\rightarrow . Would you call λ^\rightarrow (\rightarrow p.170) a logic?

So far, λ^\rightarrow (\rightarrow p.170) 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} (\rightarrow p.170) where $bool \in \mathcal{B}$, and some³⁰⁶ signature Σ (\rightarrow p.173) where

- $\Rightarrow: bool \rightarrow bool \rightarrow bool$ (\rightarrow p.171) $\in \Sigma$,
- $\equiv_{\sigma}: \sigma \rightarrow \sigma \rightarrow bool \in \Sigma$ for all types σ , and
- $\bigwedge_{\sigma}: (\sigma \rightarrow bool) \rightarrow bool \in \Sigma$ for all types σ .

We usually omit type subscripts³⁰⁷ and write \equiv, \bigwedge .

\Rightarrow, \equiv , and \bigwedge ³⁰⁸ are the logical symbols (\rightarrow p.105) of \mathcal{M} . \Rightarrow and \equiv are written infix (\rightarrow p.67).

Terms of type *bool* are called (meta-)formulae: types generalize syntactic categories (\rightarrow p.69).

³⁰⁶ Σ contains \Rightarrow, \equiv and \bigwedge , but in addition, Σ may specify other symbols.

³⁰⁷Alternatively, 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 (\rightarrow p.186).

³⁰⁸ \Rightarrow is called meta-implication, \equiv is called meta-equality, and \bigwedge is called meta-universal-quantification.

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 (\rightarrow p.25).

This is as formal as we get (for the metalogic): derivation trees in natural deduction style are authoritative.

The judgements³⁰⁹, just like for natural deduction proofs (\rightarrow p.25) in propositional logic or first-order logic, are formulae, i.e., terms of type *bool* (\rightarrow p.301). This is in contrast to derivability judgements (\rightarrow p.48) or type judgements (\rightarrow p.174).

³⁰⁹We define our proof system for \mathcal{M} using natural deduction (\rightarrow p.25).

The judgements are formulae, i.e., term of type *bool* (\rightarrow p.301). 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 (\rightarrow p.48) or type judgement (\rightarrow p.174) or a term of type, say *bool* \rightarrow *bool*.

Rules for \Rightarrow

$$\frac{\begin{array}{c} [\phi] \\ \vdots \\ \psi \end{array}}{\phi \Rightarrow \psi} \Rightarrow\text{-}I \quad \frac{\phi \Rightarrow \psi \quad \phi}{\psi} \Rightarrow\text{-}E$$

Just like rules for \rightarrow (\rightarrow p.33)!

For layout reasons we sometimes use

$$\frac{\phi \quad \phi \Rightarrow \psi}{\psi} \Rightarrow\text{-}E$$

Rules for \bigwedge

Meta-universal-quantification is formalized in the style of higher-order abstract syntax (\rightarrow p.225); may write $\bigwedge x.\phi$ as syntactic sugar (\rightarrow p.21) 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 (\rightarrow p.81).

Rules for \equiv : Equivalence Relation

$$\frac{}{a \equiv a} \equiv\text{-refl} \qquad \frac{a \equiv b}{b \equiv a} \equiv\text{-symm}$$

$$\frac{a \equiv b \quad b \equiv c}{a \equiv c} \equiv\text{-trans}$$

Just like rules for $=$ (\rightarrow p.107).

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 $=_{\alpha, \beta, \eta}$ (\rightarrow p.165).

η is equivalent to extensionality³¹⁰.

³¹⁰Extensionality 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 [HS90, pages 72-74].

Recall that we have used the notion of extensionality before, for sets (\rightarrow p.130). The idea is the same here.

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 (\rightarrow p.163), \equiv is propagated into contexts.

Conversion is built into the proof system!

Recall (\rightarrow p.183) 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

$$\frac{\begin{array}{c} [\phi] \\ \vdots \\ \psi \end{array} \quad \begin{array}{c} [\psi] \\ \vdots \\ \phi \end{array}}{\phi \equiv \psi} \equiv\text{-}I \quad \frac{\phi \equiv \psi \quad \phi}{\psi} \equiv\text{-}E$$

What is the type of ϕ and ψ here? ϕ and ψ are formulae, hence (\rightarrow p.303) *bool* (\rightarrow p.301).

What object-level connective does \equiv correspond to? \leftrightarrow (\rightarrow p.88).

13.2 Encoding Syntax and Provability

We use FOL (\rightarrow p.62) and its subset propositional logic (\rightarrow p.12) (which we call here *Prop*) as exemplary object logic.

We already know how to encode syntax (\rightarrow p.204).

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 (\rightarrow p.170), the term language (the signature (\rightarrow p.209)) and the proof rules. The thus extended logic will be called \mathcal{M}_L .

Encoding Syntax: Review

As before, $i, o \in \mathcal{B}$ (\rightarrow p.220). Previously:

$$\Sigma \supseteq \langle not : o \rightarrow o, and : o \rightarrow o \rightarrow o, imp : o \rightarrow o \rightarrow o, \\ all : (i \rightarrow o \rightarrow o) \rightarrow o, exists : (i \rightarrow o) \rightarrow o \rangle$$

Two types³¹¹ for truth values: o and $bool$.

We now need a more concise (sweeter (\rightarrow p.21)) syntax or things will become hopelessly unreadable.

But this is also quite demanding: you should always be able to “unsugar” the syntax.

³¹¹So 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.

Encoding Syntax Readably

$$\Sigma \supseteq \langle \perp : o, \\ \wedge, \vee, \rightarrow^{312} : o \rightarrow o \rightarrow o, \\ \forall, \exists : (i \rightarrow o) \rightarrow o, \\ \text{true} : o \rightarrow \text{bool} \rangle.$$

\rightarrow is both a constant declared in Σ and the function type arrow (\rightarrow p.171).

$\wedge, \vee, \rightarrow$ will be written infix (\rightarrow p.67), and we may write $\forall x.\phi$ for $\forall(\lambda x.\phi)$, and likewise for \exists .

$\text{true } A^{313}$ is usually written $\llbracket A \rrbracket$.

³¹²We 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

$$\langle \perp : o, \\ \wedge : o \rightarrow o \rightarrow o, \\ \vee : o \rightarrow o \rightarrow o, \\ \rightarrow : o \rightarrow o \rightarrow o, \\ \forall : (i \rightarrow o) \rightarrow o, \\ \exists : (i \rightarrow o) \rightarrow o \\ \text{true} : o \rightarrow \text{bool} \rangle$$

³¹³So 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 $\text{true } A$ and means that A is true”. More precisely, we can

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.

say that $\llbracket A \rrbracket$ is a meta-formula that may or may not be derivable in \mathcal{M}_L (\rightarrow p.310), and that this should reflect derivability of A in L (\rightarrow p.315).

In the file `IFOL.thy` in your Isabelle distribution (\rightarrow p.272), you find

```
Trueprop    :: "o => prop"
```

`Trueprop` corresponds to *true*.

Encoding of the Rules of Propositional Logic

$\bigwedge AB.[A] \Rightarrow ([B] \Rightarrow [A \wedge B])$	$(\wedge-I)$
$\bigwedge AB.[A \wedge B] \Rightarrow [A]$	$(\wedge-EL)$
$\bigwedge AB.[A \wedge B] \Rightarrow [B]$	$(\wedge-ER)$
$\bigwedge AB.[A] \Rightarrow [A \vee B]$	$(\vee-IL)$
$\bigwedge AB.[B] \Rightarrow [A \vee B]$	$(\vee-IR)$
$\bigwedge ABC.[A \vee B] \Rightarrow$ $([A] \Rightarrow [C]) \Rightarrow ([B] \Rightarrow [C]) \Rightarrow [C]$	$(\vee-E)$
$\bigwedge AB.([A] \Rightarrow [B]) \Rightarrow [A \rightarrow B]$	$(\rightarrow-I)$
$\bigwedge AB.[A \rightarrow B] \Rightarrow [A] \Rightarrow [B]$	$(\rightarrow-E)$
$\bigwedge A.[\perp] \Rightarrow [A]$	$(\perp-E)$

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 $Prop$ (\rightarrow p.310).

An Example Proof

$$\begin{array}{c}
 \frac{}{\wedge AB.([A] \Rightarrow [B]) \Rightarrow [A \rightarrow B]} \rightarrow\text{-}I \text{ (}\rightarrow\text{ p.314)} \\
 \frac{}{\wedge B.([P \wedge Q] \Rightarrow [B]) \Rightarrow [P \wedge Q \rightarrow B]} \wedge\text{-}E \text{ (}\rightarrow\text{ p.305)} \\
 \frac{}{([P \wedge Q] \Rightarrow [P]) \Rightarrow [P \wedge Q \rightarrow P]} \wedge\text{-}E \text{ (}\rightarrow\text{ p.305)} \\
 \hline
 [P \wedge Q \rightarrow P] \Rightarrow\text{-}E \text{ (}\rightarrow\text{ p.304)}
 \end{array}
 \qquad
 \begin{array}{c}
 \frac{}{\wedge AB.([A \wedge B] \Rightarrow [A])} \wedge\text{-}EL \text{ (}\rightarrow\text{ p.304)} \\
 \frac{}{\wedge B.([P \wedge B] \Rightarrow [P])} \wedge\text{-}E \text{ (}\rightarrow\text{ p.305)} \\
 \frac{}{([P \wedge Q] \Rightarrow [P])} \wedge\text{-}E \text{ (}\rightarrow\text{ p.305)} \\
 \frac{[[P \wedge Q]]^1}{[P]} \Rightarrow\text{-}E \text{ (}\rightarrow\text{ p.304)} \\
 \frac{}{[P \wedge Q] \Rightarrow [P]} \Rightarrow\text{-}I^1
 \end{array}$$

Example Proof Simplified

$$\begin{array}{c}
 \frac{}{\wedge AB.([A] \Rightarrow [B])} \rightarrow\text{-}I \ (\rightarrow \text{p.314}) \\
 \frac{}{\Rightarrow [A \rightarrow B]} \\
 \frac{}{\wedge B.([P \wedge Q] \Rightarrow [B])} \wedge\text{-}E \ (\rightarrow \text{p.305}) \\
 \frac{}{\Rightarrow [P \wedge Q \rightarrow B]} \\
 \frac{}{([P \wedge Q] \Rightarrow [P])} \wedge\text{-}E \ (\rightarrow \text{p.305}) \\
 \frac{}{\Rightarrow [P \wedge Q \rightarrow P]} \\
 \frac{}{[P \wedge Q \rightarrow P]} \Rightarrow\text{-}E \ (\rightarrow \text{p.305})
 \end{array}
 \qquad
 \begin{array}{c}
 \frac{}{\wedge AB.[A \wedge B]} \wedge\text{-}EL \ (\rightarrow \text{p.305}) \\
 \frac{}{\Rightarrow [A]} \\
 \frac{}{\wedge B.[P \wedge B]} \wedge\text{-}E \ (\rightarrow \text{p.305}) \\
 \frac{}{\Rightarrow [P]} \\
 \frac{}{[P \wedge Q] \Rightarrow [P]} \wedge\text{-}E \ (\rightarrow \text{p.305}) \\
 \frac{}{\Rightarrow [P \wedge Q \rightarrow P]} \Rightarrow\text{-}E \ (\rightarrow \text{p.305})
 \end{array}$$

Remarks about Example Proof

- \wedge - EL (\rightarrow p.314) and \rightarrow - E (\rightarrow p.314) are not object rules but (\rightarrow p.313) meta-axioms!
- The first, more complicated proof corresponds to the construction one would use to show that \mathcal{M}_{Prop} is complete for $Prop$ (\rightarrow p.315).
- Proof fragments of the form

$$\frac{\phi \Rightarrow \psi \quad [\phi]}{\psi} \Rightarrow\text{-}E \ (\rightarrow \text{ p.304})$$

$$\frac{\psi}{\phi \Rightarrow \psi} \Rightarrow\text{-}I \ (\rightarrow \text{ p.304})$$

can be collapsed into $\phi \Rightarrow \psi$: proof normalization.

13.3 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 $\psi \Rightarrow \psi$ ³¹⁴ and using a technique called resolution (\rightarrow p.236)

³¹⁴We have seen this before (\rightarrow p.34) as a proof in propositional logic.

$$\frac{[\psi]^1}{\psi \rightarrow \psi} \Rightarrow\text{-I } (\rightarrow \text{ p.304})^1$$

The Resolution Rule

For any formulae $\psi_1, \dots, \psi_n, \psi, \phi_1, \dots, \phi_m, \phi$ where $FV (\rightarrow \text{p.158})(\phi_1, \dots, \psi_n)$, 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 ($\rightarrow \text{p.313}$) such as $\wedge\text{-EL}$ ($\rightarrow \text{p.314}$), $[\psi_1, \dots, \psi_n] \Rightarrow \psi$ is the current goal (proof state).

Compare to phrasing using \vee^{315} !

We will now derive the rule.

³¹⁵You 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 ($\rightarrow \text{p.62}$), $\phi_1 \rightarrow \dots \rightarrow \phi_m \rightarrow \phi$ 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 ($\rightarrow \text{p.354}$).

Resolution as Derived Meta-Rule

$$\begin{array}{c}
 \frac{\frac{\frac{\frac{\bigwedge x_1 \dots x_k.}{[\phi_1, \dots, \phi_m] \Rightarrow \phi}}{[\phi_1 \theta, \dots, \phi_m \theta] \Rightarrow \phi \theta}}{[\phi_1 \theta]^2 \dots [\phi_m \theta]^2} \quad \frac{[\psi_1]^1 \dots [\psi_{i-1}]^1 \quad [\psi_1, \dots, \psi_n] \Rightarrow \psi}{[\psi_i, \dots, \psi_n] \Rightarrow \psi}}{\phi \theta} \quad \frac{[\psi_{i+1}, \dots, \psi_n] \Rightarrow \psi}{\Rightarrow -E \text{ (}\rightarrow \text{ p.304)}} \quad \frac{[\psi_1]^1 \dots [\psi_{i-1}]^1 \quad [\psi_1, \dots, \psi_n] \Rightarrow \psi}{[\psi_i, \dots, \psi_n] \Rightarrow \psi} \Rightarrow -E \text{ (}\rightarrow \text{ p.305)} \\
 \frac{\frac{[\psi_{i+1}, \dots, \psi_n] \Rightarrow \psi}{[\phi_1 \theta, \dots, \phi_m \theta, \psi_{i+1}, \dots, \psi_n] \Rightarrow \psi} \Rightarrow -I \text{ (}\rightarrow \text{ p.304)}^2}{[\psi_1, \dots, \psi_{i-1}, \phi_1 \theta, \dots, \phi_m \theta, \psi_{i+1}, \dots, \psi_n] \Rightarrow \psi} \Rightarrow -I \text{ (}\rightarrow \text{ p.304)}^1
 \end{array}$$

³¹⁶ Recall that $\phi \theta \equiv \psi_i$.

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 (\rightarrow p.31); 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. $[A] \Rightarrow [B] \Rightarrow [A \wedge B]$ instead of $\bigwedge AB.[A] \Rightarrow [B] \Rightarrow [A \wedge B]$.

In addition: use renaming for freshness³¹⁷.

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.

³¹⁷The schematic form of the resolution rule (\rightarrow p.320) 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 (\rightarrow p.200), i.e. $FV(\rightarrow \text{p.158})([\phi_1, \dots, \phi_m] \Rightarrow \phi) \cap FV(\rightarrow \text{p.158})([\psi_1, \dots, \psi_n] \Rightarrow \psi) = \emptyset$.

This assumption may be justified considering the formal derivation of the resolution rule (\rightarrow p.321). 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 reso-

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 by resolution with $\rightarrow\text{-I}$ (\rightarrow p.314):

$$\begin{array}{c}
 ([A_1] \Rightarrow [B_1]) \quad ([A \wedge B \rightarrow (C \rightarrow A \wedge C)]) \\
 \Rightarrow [A_1 \rightarrow B_1] \quad \Rightarrow [A \wedge B \rightarrow (C \rightarrow A \wedge C)] \quad (\rightarrow \text{ p.314}) \quad (\rightarrow \text{ p.319}) \\
 \hline
 ([A \wedge B] \Rightarrow [C \rightarrow A \wedge C]) \\
 \Rightarrow [A \wedge B \rightarrow (C \rightarrow A \wedge C)]
 \end{array}$$

lution rule (\rightarrow p.321), one can replace

$$\frac{\bigwedge x_1 \dots x_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi}{[\phi_1\theta, \dots, \phi_m\theta] \Rightarrow \phi\theta} \wedge\text{-E} \quad (\rightarrow \text{ p.305})$$

with

$$\frac{\bigwedge x'_1 \dots x'_k. [\phi'_1, \dots, \phi'_m] \Rightarrow \phi'}{[\phi_1\theta, \dots, \phi_m\theta] \Rightarrow \phi\theta} \wedge\text{-E} \quad (\rightarrow \text{ p.305})$$

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

What to do next³¹⁸? Again resolution with \rightarrow -I (\rightarrow p.314).

Problem: the conclusion of \rightarrow -I (\rightarrow p.314) is not unifiable³¹⁹ with $\llbracket A \wedge B \rrbracket \Rightarrow \llbracket C \rightarrow A \wedge C \rrbracket$.

the rule, and these are inadvertently identified, leading to a proof state that is more instantiated than it should be.

³¹⁸On the one hand, we want to resolve

$$(\llbracket A \wedge B \rrbracket \Rightarrow \llbracket C \rightarrow A \wedge C \rrbracket) \Rightarrow \llbracket A \wedge B \rightarrow (C \rightarrow A \wedge C) \rrbracket,$$

i.e., we have to match $(\llbracket A \wedge B \rrbracket \Rightarrow \llbracket C \rightarrow A \wedge C \rrbracket)$ 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.

³¹⁹In our current situation, Isabelle would display:

Level 1(1 subgoal)

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

$$1. A \wedge B \Longrightarrow C \rightarrow A \wedge C$$

Lifting over Assumptions

The rule for lifting an object rule (meta-axiom (\rightarrow p.313)) $[\phi_1, \dots, \phi_m] \Rightarrow \phi$ 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 $\Psi = [\psi]$.

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 (\rightarrow p.323) 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.

Deriving Assumption Lifting for one Assumption

$$\begin{array}{c}
 \frac{[\phi_1, \dots, \phi_m] \Rightarrow \phi \quad \frac{[\psi \Rightarrow \phi_1]^1 \quad [\psi]^2}{\phi_1} \Rightarrow -E (\rightarrow \text{p.304}) \quad \dots \quad \frac{[\psi \Rightarrow \phi_m]^1 \quad [\psi]^2}{\phi_m} \Rightarrow -E (\rightarrow \text{p.304})}{\frac{\phi}{\psi \Rightarrow \phi} \Rightarrow -I (\rightarrow \text{p.304})^2} \Rightarrow -E (\rightarrow \text{p.304}) \\
 \hline
 \frac{}{[\psi \Rightarrow \phi_1, \dots, \psi \Rightarrow \phi_m] \Rightarrow (\psi \Rightarrow \phi)} \Rightarrow -I (\rightarrow \text{p.304})^1
 \end{array}$$

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 - I rule³²⁰ lifted over $A \wedge B$:

$$\begin{array}{c}
 \vdots (\rightarrow \text{ p.324}) \\
 \hline
 \begin{array}{l}
 ([A \wedge B] \Rightarrow ([A_2] \Rightarrow [B_2])) \quad ([A \wedge B] \Rightarrow [C \rightarrow A \wedge C]) \\
 \Rightarrow ([A \wedge B] \Rightarrow [A_2 \rightarrow B_2]) \quad \Rightarrow [A \wedge B \rightarrow (C \rightarrow A \wedge C)]\omega
 \end{array} \\
 \hline
 \begin{array}{l}
 (\Omega[A \wedge B] \Rightarrow [C] \Rightarrow [A \wedge C]) \\
 \Rightarrow \omega[A \wedge B \rightarrow (C \rightarrow A \wedge C)]
 \end{array}
 \end{array}
 \text{ res } (\rightarrow \text{ p.323})$$

Before we proceed, we introduce the abbreviations

$$\omega = [A \wedge B \rightarrow (C \rightarrow A \wedge C)], \quad \Omega = [[A \wedge B], [C]]$$

320

$$([A \wedge B] \Rightarrow ([A_2] \Rightarrow [B_2])) \Rightarrow ([A \wedge B] \Rightarrow [A_2 \rightarrow B_2])$$

is the \rightarrow - I -rule (meta-axiom (\rightarrow p.313)) lifted over the assumption $A \wedge B$ (\rightarrow p.326).

Proof of $A \wedge B \rightarrow (C \rightarrow A \wedge C)$ (3)

We do resolution using the \wedge -I rule³²¹ lifted over Ω :

$$\begin{array}{c}
 (\Omega \Rightarrow [A_3]) \Rightarrow (\Omega \Rightarrow [B_3]) \quad \frac{\quad}{\quad} \vdots (\rightarrow \text{p.328}) \\
 \Rightarrow (\Omega \Rightarrow [A_3 \wedge B_3]) \quad (\Omega \Rightarrow [A \wedge C]) \Rightarrow \omega \\
 \hline
 (\Omega \Rightarrow [A]) \Rightarrow (\Omega \Rightarrow [C]) \Rightarrow \omega \quad \text{res } (\rightarrow \text{p.323})
 \end{array}$$

At this point, Isabelle would display $\Omega \Rightarrow [A]$ and $\Omega \Rightarrow [C]$ as two subgoals.

The next step is to solve $\Omega \Rightarrow [C]$ by assumption, but this must be formalized.

³²¹

$$(\Omega \Rightarrow [A_3]) \Rightarrow (\Omega \Rightarrow [B_3]) \Rightarrow (\Omega \Rightarrow [A_3 \wedge B_3])$$

is the \wedge -I-rule (meta-axiom (\rightarrow p.313)) lifted over the assumption list Ω (\rightarrow p.326). Recall that Ω was an abbreviation for $[[A \wedge B], [C]]$, but this is obviously irrelevant for the process of lifting.

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 (schematic³²²) derivation:

$$\frac{\frac{\frac{[\phi_i]^1}{[\phi_{i+1}, \dots, \phi_m] \Rightarrow \phi_i} \Rightarrow -I}{[\phi_i, \dots, \phi_m] \Rightarrow \phi_i} \Rightarrow -I (\rightarrow \text{p.304})^1}{[\phi_1, \dots, \phi_m] \Rightarrow \phi_i} \Rightarrow -I^{323}$$

³²²The 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 (\rightarrow p.31);
- just like for resolution rule (\rightarrow p.322), 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* (\rightarrow p.301)) and instantiate it later when it is used (\rightarrow p.331) in some resolution step.

³²³Recall here that the rule $\Rightarrow -I$, just like $\rightarrow -I$, allows you to

Proof of $A \wedge B \rightarrow (C \rightarrow A \wedge C)$ (4)

We do resolution using the assumption axiom (\rightarrow p.330):

$$\frac{\Omega \Rightarrow [C] \ (\rightarrow \text{p.330}) \quad \frac{\quad}{\quad} \vdots (\rightarrow \text{p.329})}{[\Omega \Rightarrow [A], \Omega \Rightarrow [C]] \Rightarrow \omega} \text{res } (\rightarrow \text{p.323})$$

$$\frac{\quad}{(\Omega \Rightarrow [A]) \Rightarrow \omega}$$

We used the correct instance of the assumption axiom. Alternatively³²⁴, we could have use the more generic $[A_4, B_4] \Rightarrow B_4$.

What to do next? (Recall that $\Omega = [[A \wedge B], [C]]$.) Resolution with \wedge -*EL*.

discharge zero or more (\rightarrow p.27) assumptions. In the present derivation, we discharge the assumption ϕ_i at some point but we do not discharge any other assumptions.

³²⁴As explained previously (\rightarrow p.330), we could use a more generic variant of the assumption axiom, in that we have variables in it that may become instantiated upon resolution. As in previous proof steps we assume that these variables are suitably renamed; for this purpose we index them by 4.

Note however that the variant is still specific in the sense that $m = 2$. Like in meta-axioms used before, we use letters from the beginning of the alphabet, so the variant of the assumption axiom that we use is $[A_4, B_4] \Rightarrow B_4$. The proof fragment would then look as follows:

$$\frac{[A_4, B_4] \Rightarrow B_4 \ (\rightarrow \text{p.330}) \quad \frac{\quad}{\quad} \vdots (\rightarrow \text{p.329})}{[\Omega \Rightarrow [A], \Omega \Rightarrow [C]] \Rightarrow \omega} \text{res } (\rightarrow \text{p.323})$$

$$\frac{\quad}{(\Omega \Rightarrow [A]) \Rightarrow \omega}$$

where $\theta = \{A_4 \leftarrow [A \wedge B], B_4 \leftarrow [C]\}$.

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 :

$$\frac{(\Omega \Rightarrow \llbracket A \wedge B \rrbracket) \Rightarrow (\Omega \Rightarrow \llbracket A \rrbracket) \quad \frac{\quad}{(\Omega \Rightarrow \llbracket A \rrbracket) \Rightarrow \omega} \vdash (\rightarrow \text{ p.331})}{(\Omega \Rightarrow \llbracket A \wedge B \rrbracket) \Rightarrow \omega} \text{res } (\rightarrow \text{ p.323})$$

What to do next? (Recall that $\Omega = \llbracket A \wedge B \rrbracket, \llbracket C \rrbracket$.) Prove the subgoal by assumption.

Proof of $A \wedge B \rightarrow (C \rightarrow A \wedge C)$ (6)

We do resolution using the assumption axiom (\rightarrow p.330):

$$\frac{\Omega \Rightarrow [A \wedge B] \ (\rightarrow \text{ p.330}) \quad \frac{\quad}{(\Omega \Rightarrow [A \wedge B]) \Rightarrow \omega} \vdash (\rightarrow \text{ p.332})}{\omega} \text{ res } (\rightarrow \text{ p.323})$$

Recall that $\omega = [A \wedge B \rightarrow (C \rightarrow A \wedge C)]$. Done!

Getting Rid of the Magic

In one step (\rightarrow p.332), we had to guess the right instance of \wedge -*EL*. This is not practical.

Solutions:

- Generalize (\rightarrow p.354) the resolution rule (\rightarrow p.323) to allow for instantiation of the current proof state and not just of meta-axioms.
- Derive

$$\bigwedge ABC. [[A \wedge B], ([[A], [B]] \Rightarrow [C])] \Rightarrow [C]$$

which encodes the \wedge -*E* object rule (\rightarrow p.50).

The Whole Proof at a Glance

Compare proof in \mathcal{M}_{Prop} with corresponding proof in $Prop$:

$$\frac{a. (\rightarrow \text{p.330})}{\omega} \quad \frac{\frac{\wedge-EL (\rightarrow \text{p.314})}{\dots \Rightarrow \omega}}{\dots \Rightarrow \omega} \quad \frac{\frac{a. (\rightarrow \text{p.330})}{\dots \Rightarrow \omega}}{\dots =} \quad \frac{\wedge-I (\rightarrow \text{p.314})}{\dots =} \quad \frac{\rightarrow-I (\rightarrow \text{p.314})}{\dots =}$$

$$\frac{\frac{[A \wedge B]^1}{A} \wedge\text{-}EL \quad [C]^2 \wedge\text{-}I}{\frac{A \wedge C}{C \rightarrow A \wedge C} \rightarrow\text{-}I^2} \rightarrow\text{-}I^1$$

“The meta-level proof is the object level proof upside-

down³²⁵.”

³²⁵Intuitively, as far as the order in which the object rules (\rightarrow p.313), resp. meta-axioms (\rightarrow p.313), 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 (\rightarrow p.337).

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 $\omega \Rightarrow \omega$, 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?

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 (\rightarrow p.329) 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. Interleaving³²⁶ is possible.

³²⁶If 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:

$$\frac{\frac{\frac{[A \wedge (B \wedge C)]^1}{A} \wedge\text{-}EL^3 \quad \frac{\frac{[A \wedge (B \wedge C)]^1}{B \wedge C} \wedge\text{-}ER^5 \quad \frac{B \wedge C}{C} \wedge\text{-}ER^4}{A \wedge C} \wedge\text{-}I^2}{A \wedge (B \wedge C) \rightarrow A \wedge C} \rightarrow\text{-}I^1$$

Then in the meta-proof, the meta-axioms might be applied in the following orders:

$$\begin{aligned} &\rightarrow\text{-}I^1, \wedge\text{-}I^2, \wedge\text{-}ER^4, \wedge\text{-}ER^5, \wedge\text{-}EL^3, \text{ or} \\ &\rightarrow\text{-}I^1, \wedge\text{-}I^2, \wedge\text{-}EL^3, \wedge\text{-}ER^4, \wedge\text{-}ER^5, \text{ or} \\ &\rightarrow\text{-}I^1, \wedge\text{-}I^2, \wedge\text{-}ER^4, \wedge\text{-}EL^3, \wedge\text{-}ER^5. \end{aligned}$$

13.4 Quantification

We have to add the following meta-axioms to obtain \mathcal{M}_{FOL} (\rightarrow p.310):

$$\begin{aligned}\bigwedge F.(\bigwedge x.[F\ x]) &\Rightarrow [\forall x.F\ x] && (\forall\text{-I}) \\ \bigwedge Fy.[\forall x.F\ x] &\Rightarrow [F\ y] && (\forall\text{-E}) \\ \bigwedge Fy.[F\ y] &\Rightarrow [\exists x.F\ x] && (\exists\text{-I}) \\ \bigwedge FB.[\exists x.F\ x] &\Rightarrow (\bigwedge x.[F\ x] \Rightarrow [B]) \Rightarrow [B] && (\exists\text{-E})\end{aligned}$$

Similarly as for *Prop* (\rightarrow p.315), one can show that \mathcal{M}_{FOL} is faithful for FOL.

Side condition checking is shifted to the meta-level (\rightarrow p.352).

We now consider resolution proofs (\rightarrow p.319) for FOL.

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.

Proof of $(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)$ (1)

$$(\llbracket A_1 \rrbracket \Rightarrow \llbracket B_1 \rrbracket) \quad (\rightarrow \text{p.314}) \quad \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket \quad (\rightarrow \text{p.319})$$

$$\begin{aligned} & (\llbracket \forall z.G\ z \rrbracket \Rightarrow \llbracket \forall z.G\ z \vee H\ z \rrbracket) \\ & \Rightarrow \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket \end{aligned}$$

What to do next? Resolution with \forall -I (\rightarrow p.339) lifted over assumption $\llbracket \forall z.G\ z \rrbracket$.

Proof of $(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)$ (2)

$$\begin{array}{c}
 \vdots (\rightarrow \text{p.340}) \\
 \hline
 \begin{array}{ccc}
 (\llbracket \forall z.G\ z \rrbracket \Rightarrow (\bigwedge x.\llbracket F_1\ x \rrbracket)) & & (\llbracket \forall z.G\ z \rrbracket \Rightarrow \llbracket \forall z.G\ z \vee H\ z \rrbracket) \\
 \Rightarrow (\llbracket \forall z.G\ z \rrbracket \Rightarrow \llbracket \forall x.F_1\ x \rrbracket) & \Rightarrow \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket & \\
 \hline
 \end{array}
 \end{array}
 \text{res } (\rightarrow \text{p.323})$$

$$\begin{array}{c}
 (\llbracket \forall z.G\ z \rrbracket \Rightarrow (\bigwedge z.\llbracket G\ z \vee H\ z \rrbracket)) \\
 \Rightarrow \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket
 \end{array}$$

The substitution θ (\rightarrow p.323) is $[F_1 \leftarrow \lambda w.G\ w \vee H\ w]$.

We suppress conversion (\rightarrow p.307), assuming terms are in normal form (\rightarrow p.218).

What to do next? Resolution with \vee -*IL* after lifting over assumption (\rightarrow p.326). Problem: the conclusion of \vee -*IL* (\rightarrow p.314) 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 \bigwedge 's are not dropped (\rightarrow p.323). The rule for lifting a meta-axiom (\rightarrow p.313) $\bigwedge y_1 \dots y_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi$ over a parameter z is

$$\frac{\bigwedge y_1 \dots y_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi}{\bigwedge f_1 \dots f_k. [\bigwedge z. \phi'_1, \dots, \bigwedge z. \phi'_m] \Rightarrow (\bigwedge z. \phi')} \text{ } p\text{-lift}$$

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 $[y_1 \leftarrow f_1(z), \dots, y_k \leftarrow f_k(z)]$.

$$\begin{array}{c}
 \frac{\frac{\frac{\Lambda y_1 \dots y_k. [\phi_1, \dots, \phi_m] \Rightarrow \phi}{[\phi'_1, \dots, \phi'_m] \Rightarrow \phi'} \quad \Lambda\text{-E} (\rightarrow \text{p.305}) \quad \frac{[\Lambda z. \phi'_1]^1}{\phi'_1} \quad \Lambda\text{-E} (\rightarrow \text{p.305}) \quad \dots \quad \frac{[\Lambda z. \phi'_m]}{\phi'_m}}{\frac{\phi'}{\Lambda z. \phi'} \quad \Lambda\text{-I} (\rightarrow \text{p.305})} \Rightarrow \\
 \frac{\frac{[\Lambda z. \phi'_1, \dots, \Lambda z. \phi'_m] \Rightarrow \Lambda z. \phi'}{\Lambda f_1 \dots f_k. [\Lambda z. \phi'_1, \dots, \Lambda z. \phi'_m] \Rightarrow \Lambda z. \phi'} \Rightarrow\text{-I} (\rightarrow \text{p.304})^1}{\Lambda f_1 \dots f_k. [\Lambda z. \phi'_1, \dots, \Lambda z. \phi'_m] \Rightarrow \Lambda z. \phi'} \Lambda\text{-I} (\rightarrow \text{p.305})
 \end{array}$$

After parameter lifting, we drop (\rightarrow p.323) outer quantifiers again.

Lifting \vee -IL

Lifting $\bigwedge AB.[A] \Rightarrow [A \vee B]$ (\vee -IL (\rightarrow p.314)) over z gives

$$\bigwedge G_2 H_2. (\bigwedge z. [G_2 z]) \Rightarrow (\bigwedge z. [G_2 z \vee H_2 z]).$$

We drop (\rightarrow p.323) outer quantifiers and lift over assumption (\rightarrow p.326) $[\forall z. G z]$ to obtain

$$\begin{aligned} & ([\forall z. G z] \Rightarrow \bigwedge z. [G_2 z]) \Rightarrow \\ & ([\forall z. G z] \Rightarrow \bigwedge z. [G_2 z \vee H_2 z]) \end{aligned}$$

This rule will be applied in the next step.

Proof of $(\forall z.G z) \rightarrow (\forall z.G z \vee H z)$ (3)

$$\begin{array}{c}
 \begin{array}{c}
 (\forall z.G z] \Rightarrow \wedge z.[G_2 z]) \Rightarrow \\
 (\forall z.G z] \Rightarrow \wedge z.[G_2 z \vee H_2 z]) \quad (\rightarrow \text{p.344})
 \end{array}
 \quad
 \frac{\begin{array}{c} \vdots (\rightarrow \text{p.341}) \\ (\forall z.G z] \Rightarrow (\wedge z.[G z \vee H z])) \\ \Rightarrow [(\forall z.G z) \rightarrow (\forall z.G z \vee H z)] \end{array}}{(\forall z.G z] \Rightarrow \wedge z.[G z]) \Rightarrow [(\forall z.G z) \rightarrow (\forall z.G z \vee H z)]} \text{res}
 \end{array}$$

What to do next? Resolution with \forall -E (\rightarrow p.339) lifted over z . However, this cannot be guessed from looking at $\wedge z.[G z]$, but rather from looking at premise $[\forall z.G z]$.

Lifting of \forall -E over z

Lifting $\bigwedge Fy. [\forall x. F x] \Rightarrow [F y]$ (\forall -E (\rightarrow p.339)) over parameter (\rightarrow p.342) 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 (\rightarrow p.323) outer quantifiers and lift over assumption (\rightarrow p.326) $[\forall z. G z]$ to obtain

$$\begin{aligned} ([\forall z. G z] \Rightarrow \bigwedge z. [\forall x. (G_3 z)x]) \Rightarrow \\ ([\forall z. G z] \Rightarrow \bigwedge z. [G_3 z(f_3 z)]) \end{aligned}$$

This rule will be applied in the next step.

Proof of $(\forall z.G z) \rightarrow (\forall z.G z \vee H z)$ (4)

$$\begin{array}{c}
 \begin{array}{c}
 (\forall z.G z) \Rightarrow \bigwedge z. [\forall x. (G_3 z)x] \Rightarrow \\
 (\forall z.G z) \Rightarrow \bigwedge z. [G_3 z(f_3 z)] \quad (\rightarrow \text{p.346})
 \end{array}
 \quad
 \frac{\begin{array}{c}
 \vdots (\rightarrow \text{p.345}) \\
 (\forall z.G z) \Rightarrow \bigwedge z. [G z] \Rightarrow \\
 [(\forall z.G z) \rightarrow (\forall z.G z \vee H z)]
 \end{array}}{
 \begin{array}{c}
 (\forall z.G z) \Rightarrow \bigwedge z. [\forall x.G x] \Rightarrow \\
 [(\forall z.G z) \rightarrow (\forall z.G z \vee H z)]
 \end{array}
 }_{\text{res (}}
 \end{array}$$

The substitution θ (\rightarrow p.323) is $[f_3 \leftarrow \lambda w.w, G_3 \leftarrow \lambda vw.G w]$.

We suppress conversion (\rightarrow p.307), assuming terms are in normal form (\rightarrow p.218).

What to do next? Since $z \notin FV$ (\rightarrow p.158) $(\forall x.G x)$, we can use a modified assumption axiom (\rightarrow p.330).

Modified Assumption Axiom

$$\frac{}{[\phi_1, \dots, \phi_m] \Rightarrow \bigwedge z. \phi_i} \text{ assum} \quad \text{where } z \notin FV(\phi_i).$$

It has the following derivation:

$$\frac{\frac{\frac{[\phi_i]^1}{\bigwedge z. \phi_i} \wedge\text{-I}}{[\phi_{i+1}, \dots, \phi_m] \Rightarrow \bigwedge z. \phi_i} \Rightarrow\text{-I} (\rightarrow \text{ p.330})}{[\phi_i, \dots, \phi_m] \Rightarrow \bigwedge z. \phi_i} \Rightarrow\text{-I} (\rightarrow \text{ p.304})^1$$

$$\frac{[\phi_i, \dots, \phi_m] \Rightarrow \bigwedge z. \phi_i}{[\phi_1, \dots, \phi_m] \Rightarrow \bigwedge z. \phi_i} \Rightarrow\text{-I} (\rightarrow \text{ p.330})$$

Instance of Modified Assumption Axiom

In the next step, we will use the instance

$$[\forall z.G\ z] \Rightarrow \bigwedge z.[\forall x.G\ x]$$

of

$$[\phi_1, \dots, \phi_m] \Rightarrow \bigwedge z.\phi_i.$$

We identified $\forall z.G\ z$ and $\forall x.G\ x$ by conversion (\rightarrow p.307).

Proof of $(\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z)$ (5)

$$\begin{array}{c}
 \begin{array}{c} \vdots (\rightarrow \text{p.349}) \\ \hline \begin{array}{l} \llbracket \forall z.G\ z \rrbracket \Rightarrow \\ \wedge z.\llbracket \forall x.G\ x \rrbracket \end{array} \end{array}
 \quad
 \begin{array}{c} \vdots (\rightarrow \text{p.347}) \\ \hline \begin{array}{l} (\llbracket \forall z.G\ z \rrbracket \Rightarrow \wedge z.\llbracket \forall x.G\ x \rrbracket) \Rightarrow \\ \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket \end{array} \end{array} \\
 \hline
 \llbracket (\forall z.G\ z) \rightarrow (\forall z.G\ z \vee H\ z) \rrbracket \text{ res } (\rightarrow \text{p.323})
 \end{array}$$

Done!

Remark on Step 2

Recall Step 2 (\rightarrow p.341):

$$\begin{array}{c}
 \vdots (\rightarrow \text{p.340}) \\
 \hline
 (\forall z.G z] \Rightarrow (\bigwedge x.[F_1 x])) \quad (\forall z.G z] \Rightarrow [\forall z.G z \vee H z]) \\
 \Rightarrow (\forall z.G z] \Rightarrow [\forall x.F_1 x]) \quad \Rightarrow [(\forall z.G z) \rightarrow (\forall z.G z \vee H z)] \\
 \hline
 (\forall z.G z] \Rightarrow (\bigwedge z.[G z \vee H z])) \\
 \Rightarrow [(\forall z.G z) \rightarrow (\forall z.G z \vee H z)] \quad \text{res } (\rightarrow \text{p.323})
 \end{array}$$

One could have obtained $\bigwedge z.(\forall z.G z] \Rightarrow ([G z \vee H z]))$ instead of $(\forall z.G z] \Rightarrow (\bigwedge z.[G z \vee H z]))$ by lifting \forall -I in a different way³²⁷. There will be an exercise.

³²⁷In our proof, we lifted \forall -I (\rightarrow p.339) over assumption $[\forall z.G z]$ as follows:

$$(\forall z.G z] \Rightarrow (\bigwedge x.[F_1 x])) \Rightarrow (\forall z.G z] \Rightarrow [\forall x.F_1 x])$$

It would have been possible to derive (formally, in \mathcal{M}) the following rule instead:

$$(\bigwedge x.[\forall z.G z] \Rightarrow [F_1 x]) \Rightarrow ([\forall z.G z] \Rightarrow [\forall x.F_1 x])$$

This is essentially so since $z \notin FV (\rightarrow \text{p.158})[\forall z.G z]$. If we had done it like that, step 2 (\rightarrow p.341) would have looked as follows

$$\begin{array}{c}
 \vdots (\rightarrow \text{p.340}) \\
 \hline
 (\bigwedge x.[\forall z.G z] \Rightarrow [F_1 x]) \quad ([\forall z.G z] \Rightarrow [\forall z.G z \vee H z]) \\
 \Rightarrow ([\forall z.G z] \Rightarrow [\forall x.F_1 x]) \quad \Rightarrow [(\forall z.G z) \rightarrow (\forall z.G z \vee H z)] \\
 \hline
 (\bigwedge z.[\forall z.G z] \Rightarrow [G z \vee H z]) \\
 \Rightarrow [(\forall z.G z) \rightarrow (\forall z.G z \vee H z)] \quad \text{res } (\rightarrow \text{p.323})
 \end{array}$$

The rest of the proof would then have looked slightly differently due to the different scope of the \bigwedge . For example, it

Checking Side Conditions

To demonstrate how side conditions are checked, we show a proof attempt that fails due to a side condition.

Take $\exists u. \forall w. w = u$ in FOL with equality (\rightarrow p.103), so assume we have a meta-axiom (\rightarrow p.313) for reflexivity (\rightarrow p.107):

$$\bigwedge z. [z = z] \text{ (ref)}$$

would have been necessary to lift \forall -IL (\rightarrow p.344) 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* (\rightarrow p.326). 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).

Failed Proof Attempt of $\exists u. \forall w. w = u$

$$\begin{array}{c}
 (\bigwedge x. [F_2 x]) \Rightarrow [\forall x. F_2 x] \quad (\rightarrow \text{p.339}) \\
 \hline
 \frac{
 \begin{array}{c}
 [F_1 y_1] \Rightarrow [\exists x. F_1 x] \quad (\rightarrow \text{p.339}) \quad [\exists u. \forall w. w = u] \Rightarrow [\exists u. \forall w. w = u] \\
 \hline
 [\forall w. w = y_1] \Rightarrow [\exists u. \forall w. w = u]
 \end{array}
 }{
 (\bigwedge x. [x = y_1]) \Rightarrow [\exists y. \forall x. x = y]
 }
 \end{array}$$

Substitution? $[F_1 \leftarrow \lambda v. \forall w. w = v, F_2 \leftarrow \lambda v. v = y_1]$.

What to do next? Resolution with *refl* (\rightarrow p.352) lifted over parameter x (\rightarrow p.342): $\bigwedge x. [g_3 x = g_3 x]$ ³²⁸. But $\bigwedge x. [x = y_1]$ and $\bigwedge x. [g_3 x = g_3 x]$ are not unifiable³²⁹. Proof fails!

³²⁸Note that lifting *refl* (\rightarrow p.342)

$$\bigwedge z. [z = z]$$

over x gives

$$\bigwedge g_3. \bigwedge x. [g_3 x = g_3 x].$$

Here the variable z in *refl* was replaced by the variable g_3 that depends on x . However, we drop (\rightarrow p.323) 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 (\rightarrow p.341)).

³²⁹Recall (\rightarrow p.305) that $\bigwedge x. \phi$ is syntactic sugar (\rightarrow p.21) 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 $\lambda x. y_1$

13.5 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 closure³³⁰ of ψ by $\bigwedge _.\psi$. Then

...

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.

³³⁰The universal closure of a meta-formula ψ is the formula $\bigwedge x_1 \dots x_n.\psi$ where $FV (\rightarrow \text{p.158})(\psi) = \{x_1 \dots x_n\}$.

As might be expected, the same concept is also used for FOL ($\rightarrow \text{p.62}$) formulae where it is defined in analogy using \forall instead of \bigwedge .

Instantiation of the Initial Goal

Previously, when we proved ψ we in fact proved $\bigwedge \neg.\psi$.

Now, allowing for instantiation of ψ , we in fact prove $\bigwedge \neg.\psi\theta$.

$$\frac{\frac{\frac{\bigwedge \neg.\psi \Rightarrow \psi}{\psi \Rightarrow \psi}}{\vdots} \quad \frac{\vdots}{\psi\theta}}{\bigwedge \neg.\psi\theta} \bigwedge\text{-I} (\rightarrow \text{p.305})$$

This may not be what we want³³¹.

Problem: more unifiers, hence bigger search space³³².

³³¹Suppose 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 (\rightarrow p.36).

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 (\rightarrow p.41).

³³²The 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.

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!³³³

Once a theorem is proven, any free variables will be made metavariables³³⁴, and the reading is as for rules (\rightarrow p.323): The theorem is implicitly universally quantified over the free variables.

³³³To 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.

³³⁴Prove $A \wedge B \rightarrow ?P$ in Isabelle and save (**qed**) it as a theorem and then have a look at the theorem.

13.6 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 (\rightarrow p.170) is very powerful for expressing syntax and syntactic manipulations (\rightarrow substitution). \mathcal{M} must be extended by appropriate signature (\rightarrow p.173) for an object logic.
- Rules of the object logic can be encoded and added to \mathcal{M}^{335} as axioms.

³³⁵In 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 (\rightarrow p.21) 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 (\rightarrow p.301) 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 (\rightarrow p.12) or FOL (\rightarrow p.62) in \mathcal{M} , we must add their rules as axioms.

Later (\rightarrow p.359), 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.

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 (➔ p.299): shared support and high degree of confidence.

14 HOL: Foundations

14.1 Overview

HOL is expressive foundation³³⁶ for

- Mathematics: analysis, algebra, ...
- Computer science: program correctness, hardware verification, ...

HOL is very similar to \mathcal{M} (\rightarrow p.296), but it “is” an object logic³³⁷!

- HOL is classical³³⁸.
- Still³³⁹ important: modeling of problems/domains (now within HOL).
- Still important: deriving relevant reasoning principles.

³³⁶Theorem proving in higher-order logic is an active research area with some impressive applications.

³³⁷The differences between \mathcal{M} (\rightarrow p.296) and HOL are subtle and the matter is further complicated by the fact that there are some variations in the way in which the Isabelle metalogic \mathcal{M} on the one hand and the object logic HOL on the other hand 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 (\rightarrow p.12) or first-order logic (\rightarrow p.62). That is to say, we use HOL as object logic.

³³⁸Recall (\rightarrow p.42) the distinction between classical and intuitionistic logics. There is a particular rule (\rightarrow p.387) in HOL from which the rule of the excluded middle (\rightarrow p.426) can be derived. This is in contrast to constructive (\rightarrow p.360) (intuitionistic) logics.

³³⁹We have previously looked at metatheory (\rightarrow p.296), i.e., how can one logic be represented/modeled in a metalogic.

Isabelle/HOL vs. Alternatives

We will use Isabelle/HOL³⁴⁰.

- Could forgo the use of a metalogic³⁴¹ and employ alternatives, e.g., HOL system or PVS, or constructive provers³⁴² such as Coq, Nuprl.
- Choice depends on culture and application.

In particular, we have seen how general reasoning principles (\rightarrow p.299) 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?

³⁴⁰We use Isabelle/HOL, and this means that HOL is an object logic represented by the metalogic \mathcal{M} (\rightarrow p.296).

³⁴¹There 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.

³⁴²Constructive provers are based on intuitionistic logic. The rationale is that one has to give evidence (\rightarrow p.42) for any statement. Coq and Nuprl are examples of such systems.

Safety through Strength

Safety³⁴³ via conservative (definitional) extensions (\rightarrow p.434):

- 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 extensions³⁴⁴: can lead to inconsistency.

Bertrand Russel once likened the advantages of postulation over definition to the advantages of theft over honest toil!

³⁴³The 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 (\rightarrow p.299). However, in that context, we still had to add further axioms (\rightarrow p.357) to \mathcal{M} . Here this is not the case.

Safety through strength means: HOL is strong enough to model interesting systems without having to add further axioms – that's what makes it safe.

³⁴⁴What we attempt to do here has similarities to the process of representing (\rightarrow p.296) 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 (\rightarrow p.434). Other than that, we will not add any axioms (\rightarrow p.357)!

Set Theory as Alternative?

Set theory is the logician's choice as basis for modern mathematics.

- ZFC³⁴⁵ (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 (\rightarrow p.145).
- A class cannot belong to another class (let alone a set)!

³⁴⁵ZFC stands for Zermelo-Fränkel set theory with choice [Dev93, Ebb94].

Finally: We Choose HOL!

HOL developed by [Chu40, Hen50] and rediscovered by [And02, GM93].

- Rationale: one usually works (\rightarrow p.150) with typed entities.
- Reasoning is then easier with support for types.
HOL is classical logic based on λ^{\rightarrow} (\rightarrow p.368).
- Isabelle/HOL also supports “mod cons”³⁴⁶ like polymorphism (\rightarrow p.18) and type classes (\rightarrow p.189)!

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)

³⁴⁶ “Mod cons” stands for “modern conveniences”.

What Does Higher-Order Mean?

“Type” order ³⁴⁷	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	$\forall X. (X(R, S) \leftrightarrow (\forall x. R(x) \rightarrow S(x)))$ $\rightarrow X(R', S') (\equiv subrel(R', S')^{348})$

³⁴⁷Recall the definition of an order on types (\rightarrow p.228).

Assume here, as we did in the lecture on representing syntax (\rightarrow p.220), 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 non-logical symbols (\rightarrow p.105) (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.

³⁴⁸Consider the binary predicate *subrel* which takes two unary relations as arguments. *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 *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'*.

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.

14.2 Syntax

Syntactically, HOL is a polymorphic (although not necessarily) variant of λ^\rightarrow (\rightarrow p.170) with certain default types and constants.

Default constants can be called logical symbols (\rightarrow p.105).

Types (Review)

Given a set of type constructors (\rightarrow p.196), say $\mathcal{B}^{349} = \{bool, _ \rightarrow _ (\rightarrow$ p.196), ind^{350} , $_ \times _^{351}$, $_ list$, $_ set, \dots\}$, polymorphic types (\rightarrow p.197) are defined by $\tau ::= (\rightarrow$ p.155) $\alpha \mid (\tau, \dots, \tau) T (\rightarrow$ p.196), where α is type variable.

- *bool* is also called *o* in literature [Chu40, And02]. Confusingly (\rightarrow p.311) the truth value type in Isabelle/HOL (i.e., object-level) is called *bool*.
- *bool* and \rightarrow always present in HOL; *ind* will also play a special role; other type constructors may be defined.
- Note polymorphism³⁵²!

³⁴⁹As before (\rightarrow p.196), 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 (\rightarrow p.170).

\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.

³⁵⁰*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 (\rightarrow p.372).

³⁵¹For 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 (\rightarrow p.196).

The pair type is not in the core of HOL, but it can be defined (\rightarrow p.455) in it.

³⁵²We have seen the generalization (\rightarrow p.187) of λ^{\rightarrow} to poly-

Terms

Reminder (\rightarrow p.171): $e ::= (\rightarrow$ p.155) $x \mid c \mid (ee) \mid (\lambda x^{\tau^{353}}. e)$

Typing rules as in polymorphic λ -calculus (\rightarrow p.187), with Σ defining and typing (\rightarrow p.173) constants.

Terms of type *bool* are called (well-formed) formulae.

In HOL, Σ always includes:

$$\begin{aligned} \text{True}, \text{False}^{354} &: \text{bool} \\ = &: \alpha \rightarrow \alpha \rightarrow \text{bool} \quad (\text{polymorphic, or set}^{355}) \\ \rightarrow &: \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \\ \epsilon &: (\alpha \rightarrow \text{bool}) \rightarrow \alpha \quad (\text{in Isabelle: } \mathbf{Eps} \text{ or } \mathbf{SOME}^{356}) \end{aligned}$$

morphism.

Note that in order to simplify the presentation, we neglect polymorphism in the section on semantics (\rightarrow p.371). 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.

14.3 Semantics

Intuitively: many-sorted semantics (\rightarrow p.131) + functions

- FOL: structure (\rightarrow p.73) is domain and functions/relations. Many-sorted FOL: domains are sort-indexed

$$\mathcal{A} = \langle \mathcal{D}_1, \dots, \mathcal{D}_n, I_{\mathcal{A}} \rangle$$

- HOL extends idea: \mathcal{D} indexed by (infinitely many) types.
- Complications due to polymorphism (\rightarrow p.369) [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 $X \in \mathcal{U}$ and $Y \neq \emptyset \subseteq X$, 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 set³⁵⁷ I

Choice: There is a function $ch \in \Pi_{X \in \mathcal{U}}.X$.

³⁵⁷The infinity axiom

$\exists f^{(ind \rightarrow ind)} (\rightarrow \text{p.399}). \text{injective } f \wedge \neg \text{surjective } f$ *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 (\rightarrow p.581).

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.

Prod: Encoding $X \times Y$

$X \times Y$ is the Cartesian product, i.e., the set of pairs (x, y) such that $x \in X$ and $y \in Y$.

One can actually “encode” a tuple (x, y) without explicitly postulating the “existence of tuples”³⁵⁸. E.g.: $(x, y) \equiv \{\{x\}, \{x, y\}\}$.

³⁵⁸According 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.

That is, we assume that if one understands what a and b are, then one also understands what the pair (a, b) is. A pair is a “semantic object”.

Ultimately, semantics can only be understood using one’s intuition, and only be explained using natural language (\rightarrow p.298). One should try to base the semantics on a very small number of fundamental concepts.

Therefore, one might want to avoid having a concept “pair” (“tuple”) explicitly, or put differently, one might want to reduce “pairs” to something even more fundamental. That’s what is intended by the encoding $\{\{x\}, \{x, y\}\}$.

Note that this reduction step somehow makes the type discipline (\rightarrow p.377) invisible, because x and y might be semantic objects “of different type”.

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 $ch \in \Pi_{X \in \mathcal{U}}.X$ ³⁵⁹, i.e., ch is of dependent type.

Essentially, the constant ϵ will be interpreted as ch , but you will see the technical details later (\rightarrow p.379).

³⁵⁹When 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 .

Function Space in \mathcal{U}

Define set $X \rightarrow Y$ as (graph of) functions³⁶⁰ from X to Y .

- For nonempty Y ³⁶¹, this set is nonempty and is a subset of $\wp(X \times Y)$.
- From closure conditions (\rightarrow p.372): $X, Y \in \mathcal{U}$ then $X \rightarrow Y \in \mathcal{U}$.

³⁶⁰In 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** (\rightarrow p.372) we have that $X \rightarrow Y \in \mathcal{U}$.

³⁶¹It is crucial in the semantics that any type is inhabited (\rightarrow p.372), i.e., has an element. The function space $X \rightarrow Y$ is empty if X is empty.

This means that $\mathcal{D}_{\tau \rightarrow \sigma}$ (\rightarrow p.377) 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

Distinguished Sets

From

Infty: \mathcal{U} contains a distinguished infinite set (\rightarrow p.372) I

Sub: If $X \in \mathcal{U}$ and $Y \neq \emptyset \subseteq X$, then $Y \in \mathcal{U}$

it follows that the following sets exist in \mathcal{U} :

Unit: A distinguished 1-element³⁶² set $\{1\}$

Bool: A distinguished 2-element set $\{T, F\}$.

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 (\rightarrow p.379) and one concerning sets (\rightarrow p.450).

³⁶²Of 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.

Frames

For semantics, we neglect polymorphism (\rightarrow p.369). τ and σ range over types.

A frame is a collection $\{\mathcal{D}_\tau\}_\tau$ of non-empty sets (domains (\rightarrow p.372)) $\mathcal{D}_\tau \in \mathcal{U}$, 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 (\rightarrow p.375) from \mathcal{D}_τ to \mathcal{D}_σ .
- \mathcal{D}_{ind} (\rightarrow p.369) = I (\rightarrow p.372).

Note: for fundamental reasons discussed later (\rightarrow p.389), one cannot simply define $\mathcal{D}_{\tau \rightarrow \sigma} = \mathcal{D}_\tau \rightarrow \mathcal{D}_\sigma$ at this stage.

Interpretations

An interpretation $\mathfrak{M} = \langle \{\mathcal{D}_\tau\}_\tau, \mathcal{J} \rangle$ 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 $\mathcal{J}(False) = F$;
- $\mathcal{J}(=_{\tau \rightarrow \tau \rightarrow bool})^{363}$ is equality on \mathcal{D}_τ ;
- $\mathcal{J}(\rightarrow)$ is implication function over \mathcal{D}_{bool} . For $b, b' \in \{T, F\}$,

$$\mathcal{J}(\rightarrow)(b, b') = \begin{cases} F & \text{if } b = T \text{ and } b' = F \\ T & \text{otherwise} \end{cases}$$

³⁶³For $=$ and ϵ , we give type subscripts in the presentation of the semantics since we assume, conceptually, that there are infinitely many copies (\rightarrow p.370) of those constants, one for each type. We do this to avoid explicit polymorphism in this presentation.

Interpretations (Cont.)

- $\mathcal{J}(\epsilon_{(\tau \rightarrow bool) \rightarrow \tau} (\rightarrow \text{p.378}))$ is defined by (for $f \in (\mathcal{D}_\tau \rightarrow \mathcal{D}_{bool})$):

$$\mathcal{J}(\epsilon_{(\tau \rightarrow bool) \rightarrow \tau})(f)^{364} = \begin{cases} ch(f^{-1}(\{T\})) & \text{if } f^{-1}(\{T\}) \neq \emptyset \\ ch(\mathcal{D}_\tau) & \text{otherwise} \end{cases}$$

Note: If a frame $\{\mathcal{D}_\tau\}_\tau$ does not contain all of the functions used above, then $\{\mathcal{D}_\tau\}_\tau$ cannot belong to any interpretation.

³⁶⁴We 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 (\rightarrow p.374) 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 (\rightarrow p.451) 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 (\rightarrow p.377) as nonempty set (\rightarrow p.372)). Note the importance of inhabitation.

A Terminological Note

The terminology is slightly different from FOL:

In FOL, “ $\langle \{\mathcal{D}_\tau\}_\tau, \mathcal{J} \rangle$ ” is called structure (\rightarrow p.73) and “ \mathcal{J} ” is called interpretation (\rightarrow p.73).

In HOL, $\langle \{\mathcal{D}_\tau\}_\tau, \mathcal{J} \rangle$ is called interpretation and \mathcal{J} is called denotation function.

The Value of Terms (Naive)

In analogy to FOL (\rightarrow p.74), given an interpretation $\mathfrak{M} = \langle \{\mathcal{D}_\tau\}_\tau, \mathcal{J} \rangle$ and a type-indexed collection of assignments³⁶⁵ $A = \{A_\tau\}_\tau$, 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. $\mathcal{V}_A^\mathfrak{M}(s_{\tau \rightarrow \sigma}{}^{366}t_\tau) = (\mathcal{V}_A^\mathfrak{M}(s))(\mathcal{V}_A^\mathfrak{M}(t))$, 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 $\mathcal{V}_{A[x \leftarrow e]}^\mathfrak{M}{}^{367}(t)$.

What is the problem? Condition 4!

³⁶⁵An assignment (previously called valuation (\rightarrow p.73)) maps variables to elements of a domain (\rightarrow p.372).

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.

³⁶⁶In the presentation of models, we give type subscripts for the cases $\mathcal{V}_A^\mathfrak{M}(s_{\tau \rightarrow \sigma} t_\tau)$ 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.

³⁶⁷ $A[x \leftarrow e]$ denotes the assignment that is identical to A except that $A(x) = e$.

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 (\rightarrow p.372) of \mathcal{U} (since it is required that $\mathcal{V}_A^{\mathfrak{M}}(t_\rho) \in \mathcal{D}_\rho$ for all t , and $\mathcal{D}_\rho \in \mathcal{U}$ (\rightarrow p.377)): this is guaranteed by closure conditions on \mathcal{U} (\rightarrow p.372);
- in a certain domain (\rightarrow p.372) of \mathcal{U} , namely $\mathcal{D}_{\tau \rightarrow \sigma}$ ³⁶⁸; 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)³⁶⁹ model.

³⁶⁸In 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.

³⁶⁹General models must be distinguished from standard models, as well see later (\rightarrow p.383).

We sometimes omit the word “general” in 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 (\rightarrow p.389).

If \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 \mathfrak{M} wrt. A .

Note that in contrast to first-order logic (\rightarrow p.76), “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} (\rightarrow p.381) 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} (\rightarrow p.381) 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 (\rightarrow p.382).

A formula ϕ is valid in the standard sense if it is valid in every standard model (\rightarrow p.383).

Existence of Values

Closure conditions (\rightarrow p.381) 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 $A(y) = 5$.

14.4 Basic Rules

We now give the core calculus of HOL. Its rules can be stated using only the constants $=$, \rightarrow , and ϵ . However, there will be one rule, *tof* (\rightarrow p.387) (“true or false”), which would be hard to read if we did that.

So we allow ourselves to “cheat”³⁷⁰ and also use constants *True*, *False*, \vee to write rule *tof* (\rightarrow p.387).

Later we will define those constants, i.e., regard them as syntactic sugar (\rightarrow p.21).

³⁷⁰Rule *tof* (\rightarrow p.387) can be written as follows:

$$\begin{array}{l} (\lambda\psi. (\phi = (\lambda x.x = \lambda x.x) \rightarrow \psi) \rightarrow \\ \quad (\phi = ((\lambda\eta.\eta) = \lambda x.(\lambda x.x = \lambda x.x)) \rightarrow \psi) \rightarrow \psi) = \\ (\lambda x.(\lambda x.x = \lambda x.x)) \end{array} \quad \text{tof}$$

Our notation for rule *tof* (\rightarrow p.387) is thus based on the following definitions:

$$\begin{aligned} \text{True } (\rightarrow \text{ p.397}) &= (\lambda x^{\text{bool}} (\rightarrow \text{ p.397}).x = \lambda x.x) \\ \text{False } (\rightarrow \text{ p.397}) &= \forall \phi^{\text{bool}} (\rightarrow \text{ p.397}).\phi (\rightarrow \text{ p.397}) \\ \vee (\rightarrow \text{ p.397}) &= \lambda \phi \eta. \forall \psi. (\phi \rightarrow \psi) \rightarrow (\eta \rightarrow \psi) \rightarrow \psi \end{aligned}$$

Basic Rules in Sequent Notation

$$\begin{array}{c}
 \frac{}{\Gamma \vdash \phi = \phi} \text{refl} \qquad \frac{\Gamma \vdash \phi = \eta \quad \Gamma \vdash P(\phi)}{\Gamma \vdash P(\eta)} \text{subst} \\
 \frac{\Gamma \vdash \phi x = \eta x}{\Gamma \vdash \phi = \eta} \text{ext}^{*371} \qquad \frac{\Gamma, \phi \vdash \eta}{\Gamma \vdash \phi \rightarrow \eta} \text{impI} \\
 \frac{\Gamma \vdash \phi \rightarrow \eta \quad \Gamma \vdash \phi}{\Gamma \vdash \eta} \text{mp} \\
 \frac{}{\Gamma \vdash (\phi \rightarrow \eta) \rightarrow (\eta \rightarrow \phi) \rightarrow (\phi = \eta)} \text{iff} \\
 \frac{}{\phi = \text{True} \vee \phi = \text{False}} \text{tof} (\rightarrow \text{p.386}) \qquad \frac{\Gamma \vdash \phi x}{\Gamma \vdash \phi(\epsilon x. \phi x^{373})} \text{selectI}^{372}
 \end{array}$$

³⁷¹The rule

$$\frac{\Gamma \vdash \phi x = \eta x}{\Gamma \vdash \phi = \eta} \text{ext}$$

has the side condition that $x \notin FV(\Gamma)$.

Phrased like

$$\frac{\phi x = \eta x}{\phi = \eta} \text{ext}$$

the rule has the side condition that x must not occur freely (\rightarrow p.72) in the derivation of $\phi x = \eta x$.

³⁷²You may wonder why there is no rule for eliminating ϵ . We will later (\rightarrow p.421) see a rule derivation where an ϵ is effectively eliminated, and we will also see that this is done without requiring a rule explicitly for this purpose.

Apart from that, the ϵ -operator is used in HOL as basis for defining (\rightarrow p.396) \exists and the if-then-else constructs. Once we have derived the appropriate rules for those, we will not explicitly encounter ϵ anymore.

³⁷³For readability, we will frequently use a syntax that one is

Axiom of Infinity

There is one additional rule (axiom) that will give us the existence of infinite sets (\rightarrow p.372):

$$\frac{}{\exists f^{(ind \rightarrow ind)}.injective^{374} f \wedge \neg surjective f} \text{ infty}$$

Has special role. Interesting to look at HOL with or without infinity (\rightarrow p.389). Won't (\rightarrow p.581) consider infinity today.

Note “cheating” (\rightarrow p.396) (use of \exists).

These eight (nine) rules are the entire basis!

more used to than higher-order abstract syntax (\rightarrow p.225):

$\epsilon x.\phi x$ stands for $\epsilon(\phi)$.

$\forall x.\phi(x)$ stands for $\forall(\phi)$, and likewise for \exists .

We have done the same previously (\rightarrow p.312) for \mathcal{M} .

Soundness and Completeness

Soundness is straightforward [And02, p. 240].

Completeness only follows w.r.t. general models (\rightarrow p.382), as opposed to standard (\rightarrow p.383) 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 (\rightarrow p.384) in all standard models, but not in all general models, and which cannot be proven in our calculus (\rightarrow p.386). Our calculus can prove the formulas that are true in all general models including non-standard ones. This reconciles HOL with Gödel's incompleteness theorem³⁷⁵.

If we consider a version of HOL without infinity (\rightarrow p.372), then every model is a standard model³⁷⁶ and so completeness holds.

³⁷⁵This 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 (\rightarrow p.383) models. This resolves the apparent contradiction with Gödel's incompleteness theorem: HOL with infinity contains I (\rightarrow p.372), hence the natural numbers (\rightarrow p.586), 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].

³⁷⁶We might consider a version of HOL without infinity, i.e.,

14.5 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 Isabelle³⁷⁷.

one where each domain (\rightarrow p.372) 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 (\rightarrow p.381) $\mathcal{D}_{\tau \rightarrow \sigma} = \mathcal{D}_{\tau} \rightarrow \mathcal{D}_{\sigma}$.

For details consult [And02, §54].

³⁷⁷This file should be contained in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

Moreover, in `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 HOL.thy of Isabelle 98. E.g., the definition of `Ex_def` is now different from the one presented here.

Note also that here in the slides, we use a style of displaying Isabelle files which uses some symbols beyond the usual

We present language and rules³⁷⁸ using “mathematical” syntax, but also comparing with Isabelle (concrete/HOAS (\rightarrow p.225)) syntax.

We take polymorphism (\rightarrow p.369) back on board.

ASCII set (\rightarrow p.399).

³⁷⁸We will mix natural deduction (\rightarrow p.25) (with discharging assumptions), natural deduction written in sequent style (\rightarrow p.48), 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

$$\frac{\rho, \Gamma \vdash \psi}{\Gamma \vdash \phi}$$

(Central Parts of the) Language

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

$$\frac{\bigwedge x. \psi}{\phi(x)}$$

$\Sigma_0 =$

$$\left\{ \begin{array}{ll} \textit{True}, \textit{False}^{379} & : \textit{bool}, \\ \neg_{-}^{380} & : \textit{bool} \rightarrow \textit{bool}, \\ _ \wedge _, _ \vee _, _ \rightarrow _ & : \textit{bool} \rightarrow \textit{bool} \rightarrow \textit{bool}, \\ \forall _, \exists _ & : (\alpha \rightarrow \textit{bool}) \rightarrow \textit{bool}, \\ \epsilon _ & : (\alpha \rightarrow \textit{bool}) \rightarrow \alpha, \\ \textit{if_then_else_} & : \textit{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha, \\ _ = _ & : \alpha \rightarrow \alpha \rightarrow \textit{bool} \end{array} \right\}$$

We will switch between the various ways of writing the rules! This means in particular that we will use \implies and \bigwedge from Isabelle's metalogic (\rightarrow p.296).

³⁷⁹For 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 (\rightarrow p.21).

³⁸⁰We use a notation with $_$ to indicate the arity and fixity of constants, as this has been done for type constructors (\rightarrow p.196) before.

The whole matter of arity of fixity is one of notational convenience. For example, as the type of \wedge indicates, we should write $(\wedge \phi)\psi$ (Curryd notation (\rightarrow p.164)), but we write $\phi \wedge \psi$ since it is more what we are used to.

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 (➔ p.390).

Basic Rules in Mixed (\rightarrow p.391) Notation

$$\begin{array}{c}
 \frac{}{\phi = \phi} \text{ refl} \qquad \frac{\phi = \eta \quad P(\phi)}{P(\eta)} \text{ subst} \\
 \frac{\phi x = \eta x}{\phi = \eta} \text{ ext}^* \ (\rightarrow \text{ p.387}) \qquad \frac{\phi \Longrightarrow \eta}{\phi \rightarrow \eta} \text{ impI} \\
 \frac{\phi \rightarrow \eta \quad \phi}{\eta} \text{ mp} \\
 \frac{}{(\phi \rightarrow \eta) \rightarrow (\eta \rightarrow \phi) \rightarrow (\phi = \eta)} \text{ iff} \\
 \frac{}{\phi = \text{True} \vee \phi = \text{False}} \text{ tof} \qquad \frac{\phi x}{\phi(\epsilon x. \phi x)} \text{ selectI}
 \end{array}$$

No more “Cheating”: The Definitions

$$\begin{aligned}
True^{381} &=^{382} (\lambda x^{bool} (\rightarrow \text{p.397}).x = \lambda x.x) \\
\forall^{383} &= \lambda \phi^{\alpha \rightarrow bool} (\rightarrow \text{p.370}).(\phi = \lambda x.True) \\
False^{384} &= \forall \phi^{bool^{385}}.\phi^{386} \\
\vee^{387} &= \lambda \phi \eta. \forall \psi. (\phi \rightarrow \psi) \rightarrow (\eta \rightarrow \psi) \rightarrow \psi \\
\wedge^{388} &= \lambda \phi \eta. \forall \psi. (\phi \rightarrow \eta \rightarrow \psi) \rightarrow \psi \\
\neg^{389} &= \lambda \phi. (\phi \rightarrow False) \\
\exists^{390} &= (\lambda \phi. \phi(\epsilon x. \phi x)) \\
If^{391} &= \lambda \phi^{bool} xy. \epsilon z. (\phi = True \rightarrow z = x) \wedge \\
&\quad (\phi = False \rightarrow z = y)
\end{aligned}$$

381

$$True = (\lambda x^{bool}.x = \lambda x.x)$$

The term $\lambda x^{bool}.x = \lambda x.x$ evaluates to T (\rightarrow p.378), 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 (\rightarrow p.440).

Note that $(\lambda x^{bool}.x = \lambda x.x)$ is closed (\rightarrow p.440). Definitions must always be closed (\rightarrow p.440).

³⁸²It is a design choice if we want to add these definitions at the level of the object logic (HOL) (\rightarrow p.359) or at the level of the \mathcal{M} (\rightarrow p.296). 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 (\rightarrow p.21). The second way corresponds to what is done in Isabelle, see `HOL.thy` (\rightarrow p.390). It is technically more convenient since rewriting (\rightarrow p.276) is based on meta-level equalities.

Logically, it is not a big difference which way one chooses.

$If = \lambda\phi xy. \epsilon z. (\phi = True \rightarrow z = x) \wedge (\phi = False \rightarrow z = y)$

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 (\rightarrow p.225).

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 $\lambda\phi xy. \epsilon z. (\phi = True \rightarrow z = x) \wedge (\phi = False \rightarrow z = y)$.

Note: Different Syntaxes

Mathematical vs. Isabelle, e.g.

$\neg\phi$ $\lambda x^{bool}.P$	Not Phi <code>%³⁹²x :: ³⁹³bool.P</code>
------------------------------------	---

HOAS (\rightarrow p.225) vs. concrete, e.g.

$\forall (\lambda x^\tau. (\wedge p(x) q(x)))$ $\epsilon(P)$	$\forall x^\tau. p(x) \wedge q(x)$ $\epsilon x. P(x)$
---	--

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).

³⁹²Note that the λ -binder of the object logic HOL is not distinguished from the λ -binder of Isabelle’s metalogic \mathcal{M} (\rightarrow p.296). One could introduce an object level constant *lambda*, but one quickly sees that it would be an unnecessary overhead.

³⁹³As we have learned previously (\rightarrow p.171), λ -abstracted variables should have a type superscript, although this superscript is often omitted since the type can be inferred (\rightarrow p.370).

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.

14.6 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 (\rightarrow p.389) straightforward.
- Logic complete (\rightarrow p.389) (w.r.t. general models, but not standard (\rightarrow p.383) models).
- Next lecture we will see how all well-known inference rules can be derived.

15 HOL: Deriving Rules

Outline

Last lecture (\rightarrow p.359): Introduction to HOL

- Basic syntax (\rightarrow p.368) and semantics (\rightarrow p.371)
- Basic eight (or nine) rules (\rightarrow p.387)
- Definitions (\rightarrow p.396) of *True*, *False*, \wedge , \vee , \forall ...

Today:

- Deriving rules (\rightarrow p.401) for the defined constants
- Outlook on the rest of this course (\rightarrow p.429)

Reminder: Different Syntaxes

Mathematical

vs. Isabelle, e.g.

$\neg\phi$
 $\lambda x^{bool}.P$

`Not Phi`
`% (→ p.399)x :: (→ p.399)bool. P`

HOAS (→ p.225)

vs. concrete, e.g.

$\forall (\lambda x^\tau (\rightarrow p.399).(\wedge p(x) q(x)))$
 $\epsilon(P)$

$\forall x^\tau (\rightarrow p.399).p(x) \wedge q(x)$
 $\epsilon x.P(x)$

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).

Reminder: Definitions

$$\begin{aligned} True \ (\rightarrow \text{p.397}) &= (\lambda x^{bool} (\rightarrow \text{p.397}).x = \lambda x.x) \\ \forall \ (\rightarrow \text{p.397}) &= \lambda \phi^{\alpha \rightarrow bool} (\rightarrow \text{p.370}).(\phi = \lambda x.True) \\ False \ (\rightarrow \text{p.397}) &= \forall \phi^{bool} (\rightarrow \text{p.397}).\phi \ (\rightarrow \text{p.397}) \\ \vee \ (\rightarrow \text{p.397}) &= \lambda \phi \eta. \forall \psi. (\phi \rightarrow \psi) \rightarrow (\eta \rightarrow \psi) \rightarrow \psi \\ \wedge \ (\rightarrow \text{p.397}) &= \lambda \phi \eta. \forall \psi. (\phi \rightarrow \eta \rightarrow \psi) \rightarrow \psi \\ \neg \ (\rightarrow \text{p.397}) &= \lambda \phi. (\phi \rightarrow False) \\ \exists \ (\rightarrow \text{p.397}) &= (\lambda \phi. \phi(\epsilon x. \phi x)) \\ If \ (\rightarrow \text{p.397}) &= \lambda \phi xy. \epsilon z. (\phi = True \rightarrow z = x) \wedge \\ &\quad (\phi = False \rightarrow z = y) \end{aligned}$$

Derived Rules

The definitions (\rightarrow p.404) 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.

15.1 Equality

- Rule *sym* and ND derivation³⁹⁴

$$\frac{s = t \quad \frac{}{s = s} \text{refl} (\rightarrow \text{p.387})}{t = s} \text{symsubst} (\rightarrow \text{p.387})$$

³⁹⁴We present most of those proofs by giving a derivation tree (\rightarrow p.25) for it, but sometimes, we also give an Isabelle proof script.

Note also the mix of syntaxes (\rightarrow p.391).

- 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* and ND derivation (\rightarrow p.405)

$$\frac{\frac{r = s}{s = r} \text{ sym } (\rightarrow \text{ p.405}) \quad s = t}{r = t} \text{ transsubst } (\rightarrow \text{ p.387})$$

Isabelle rule `[| r=s; s=t |] ==> r=t`

- Congruences (only Isabelle forms):

$(f :: 'a \Rightarrow 'b) = g \Rightarrow f(x) = g(x)$ (*fun_cong*)

$x = y \Rightarrow f(x) = f(y)$ (*arg_cong*)

Isabelle proofs using *subst* (\rightarrow p.387) and *refl* (\rightarrow p.387).

Equality of Booleans (*iffI*)

Rule *iffI* and ND derivation (\rightarrow p.405)

$$\begin{array}{c}
 \frac{\frac{}{(P \rightarrow Q) \rightarrow (Q \rightarrow P) \rightarrow (P = Q)} \text{iff} (\rightarrow \text{p.387}) \quad \frac{\frac{[P] \quad \vdots \quad Q}{P \rightarrow Q} \text{impI} (\rightarrow \text{p.395})}{(Q \rightarrow P) \rightarrow (P = Q)} \text{mp} (\rightarrow \text{p.387})}{P = Q} \quad \frac{\frac{[Q] \quad \vdots \quad P}{Q \rightarrow P}}{Q \rightarrow P}
 \end{array}$$

Isabelle rule `[| P ==> Q; Q ==> P |] ==> P=Q.`

Equality of Booleans (*iffD2*)

Rule *iffD2* and ND derivation (➔ p.405)

$$\frac{\frac{P = Q}{Q = P} \text{sym} (\text{➔ p.405}) \quad Q}{P} \text{iffD2subst} (\text{➔ p.387})$$

Isabelle rule `[| P=Q; Q |] ==> P.`

15.2 *True*

$True = ((\lambda x^{bool}.x) = (\lambda x.x))$

- Rule *TrueI* and ND derivation (\rightarrow p.405)

$$\frac{}{True(\lambda x.x) = (\lambda x.x)} TrueIrefl \ (\rightarrow \text{p.387})$$

- Rule *eqTrueE* and ND derivation (\rightarrow p.405)

$$\frac{P = True \quad \overline{True} \quad TrueI}{P} eqTrueEiffD2 \ (\rightarrow \text{p.409})$$

Isabelle rule **P=True ==> P**.

True (Cont.)

- Rule *eqTrueI* and ND derivation (\rightarrow p.405)

$$\frac{\frac{}{True} \text{ TrueI } (\rightarrow \text{ p.410}) \quad P}{P = True} \text{ eqTrueIffI } (\rightarrow \text{ p.408})$$

Note that 0 assumptions were discharged.

Isabelle rule `P ==> P=True`.

15.3 Universal Quantification

$$\forall P = (P = (\lambda x. True))$$

- Rule *allI* and ND derivation (\rightarrow p.405)

$$\frac{\frac{P(x)}{P(x) = True} \text{eqTrueI } (\rightarrow \text{ p.411})}{\forall P P = \lambda x. True} \text{allIext } (\rightarrow \text{ p.387})$$

Inherits (\rightarrow p.95) the side condition of *ext* (\rightarrow p.387):
 x must not occur freely in the derivation of $P(x)$.

Isabelle rule $(!!x. P(x)) ==> !x. P(x)$.

Example Illustrating Side Condition

$$\frac{\frac{[r(x)]^1}{r(x) \rightarrow r(x)} \rightarrow -I^1}{\forall x. r(x) \rightarrow r(x)} \text{allI}$$

Why is this correct? Let's do it without using *allI* explicitly:

$$\frac{\frac{\frac{[r(x)]^2}{r(x) \rightarrow r(x)} \rightarrow -I^2}{(r(x) \rightarrow r(x)) = \text{True}} \text{eqTrueI}}{\lambda x. (r(x) \rightarrow r(x)) = \lambda x. \text{True}} \text{ext}$$

The side condition is respected.

Universal Quantification (Cont.)

- Rule *spec* (recall (\rightarrow p.387) $\forall P$ means $\forall x.Px$) and ND derivation (\rightarrow p.405)

$$\frac{\frac{\forall P P = \lambda x. True}{P(t) = True} \text{fun_cong } (\rightarrow \text{ p.407})}{P(t)} \text{speceqTrueE } (\rightarrow \text{ p.410})$$

Isabelle rule `!x::'a. P(x) ==> P(x).`

Note: Need universal quantification to reason about *False* (since $False = (\forall P.P)$).

15.4 *False*

$False = (\forall P.P)$ $(= \forall(\lambda P.P) \text{ (} \rightarrow \text{ p.387)})$

- FalseI: No rule!
- Rule *FalseE* and ND derivation (\rightarrow p.405)

$$\frac{False \forall P. P}{P} FalseEspec \text{ (} \rightarrow \text{ p.414)}$$

Isabelle rule **False** \Rightarrow P.

False (Cont.)

- Rule *False_neq_True* and ND derivation (\rightarrow p.405)

$$\frac{False = True}{False} eqTrueE (\rightarrow \text{p.410})$$

$$\frac{False}{P} False_neq_TrueFalseE (\rightarrow \text{p.415})$$

Isabelle rule **False=True ==> P.**

- Similar:

$$\frac{True = False}{P} True_neq_False$$

15.5 Negation

$$\neg P = P \rightarrow False$$

- Rule *notI* and ND derivation (\rightarrow p.405)

$$\frac{\begin{array}{c} [P] \\ \vdots \\ False \end{array}}{\neg P \rightarrow False} notImpI \ (\rightarrow \text{ p.395})$$

Isabelle rule (P ==> False) ==> ~P.

Negation (2)

- Rule *notE* and ND derivation (➔ p.405)

$$\frac{\neg P \quad P \rightarrow False \quad P}{False} mp \text{ (➔ p.387)}$$

$$\frac{False}{R} notEFalseE \text{ (➔ p.415)}$$

Isabelle rule `[| ~P; P |] ==> R.`

Negation (3)

- Rule *True_Not_False* and ND derivation (\rightarrow p.405)

$$\frac{\frac{[True = False]^1}{False} \text{ True_neq_False } (\rightarrow \text{ p.416})}{\neg(True = False)(True = False) \rightarrow False} \text{ True_Not_FalsenotI } (\rightarrow \text{ p.405})$$

Isabelle rule **True** \sim **False**.

15.6 Existential Quantification

$$\exists P = P(\epsilon x.P(x))$$

- Rule *existsI* and ND derivation (\rightarrow p.405)

$$\frac{P(x)}{\exists P P(\epsilon x.P(x))} \text{ existsIselectI } (\rightarrow \text{ p.387})$$

Isabelle rule $P(x) \implies ?x::'a.P(x)$.

Existential Quantification (Cont.)

- Rule *existsE* and ND derivation (\rightarrow p.405)

$$\begin{array}{c}
 [P(x)]^1 \\
 \vdots \\
 Q \\
 \hline
 P(x) \rightarrow Q \quad \text{impI}^1 \\
 \hline
 \forall x.(P(x) \rightarrow Q) \quad \text{allI} \\
 \hline
 \exists x.P(x) \quad \text{spec} \\
 \hline
 P(\epsilon x.P(x)) \rightarrow Q \quad \text{existsEmp} \\
 \hline
 Q
 \end{array}$$

Inherits side condition from *allI* (just like in FOL (\rightarrow p.94)).

Isabelle rule `[| ?x.P(x); !!x.P(x) ==> Q |] ==> Q.`

15.7 Conjunction

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

- Rule *conjI* and ND derivation (\rightarrow p.405)

$$\frac{\frac{\frac{[P \rightarrow Q \rightarrow R]^1 \quad P}{Q \rightarrow R} \text{ mp } (\rightarrow \text{ p.387}) \quad Q}{R} \text{ mp } (\rightarrow \text{ p.387})}{\frac{(P \rightarrow Q \rightarrow R) \rightarrow R}{P \wedge Q \forall R. (P \rightarrow Q \rightarrow R) \rightarrow R} \text{ conjIallI}} \text{ impI } (\rightarrow \text{ p.395})^1$$

Isabelle rule `[| P; Q |] ==> P & Q.`

Conjunction (Cont.)

- Rule *conjEL* and ND derivation (➔ p.405)

$$\frac{\frac{P \wedge Q \forall R. (P \rightarrow Q \rightarrow R) \rightarrow R}{(P \rightarrow Q \rightarrow P) \rightarrow P} \text{spec (➔ p.414)} \quad \frac{\frac{[P]^1}{Q \rightarrow P} \text{I} \quad P \rightarrow Q \rightarrow P}{P} \text{E}}{P}$$

Isabelle rule `P & Q ==> P`.

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* (**→** p.426))

15.8 Disjunction

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

- Rule *disjIL* and ND derivation (\rightarrow p.405)

$$\frac{\frac{\frac{[P \rightarrow R]^1 \quad P}{R} \text{ mp } (\rightarrow \text{ p.387})}{(Q \rightarrow R) \rightarrow R} \text{ impI } (\rightarrow \text{ p.395})}{\frac{(P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R}{P \vee Q \forall R. (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R} \text{ impI } (\rightarrow \text{ p.395})^1} \text{ disjILallI}$$

Isabelle rule $P \implies P \mid Q$.

Disjunction (Cont.)

- $Q \implies P \vee Q$ (*disjIR*) similar
- Rule *disjE* and ND derivation (\rightarrow p.405)

$$\frac{
 \frac{
 \frac{
 P \vee Q \forall R. (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R
 }{
 (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R
 } \text{spec } (\rightarrow \text{ p.414})
 }{
 (Q \rightarrow R) \rightarrow R
 } \text{impI } (\rightarrow)
 }{
 R
 } \text{mp } (\rightarrow)$$

Isabelle rule `[| P | Q; P ==> R; Q ==> R |] ==> R.`

- $P \vee \neg P$ (*excluded middle*). Follows using *tof* (\rightarrow p.387).

15.9 Miscellaneous Definitions

See HOL.thy (➔ p.390)!

Typical example (if-then-else (➔ p.397)):

$$\begin{aligned} If = \lambda\phi^{bool}xy.\epsilon z. & \ (\phi = True \rightarrow z = x) \\ & \wedge \ (\phi = False \rightarrow z = y) \end{aligned}$$

The way rules are derived should now be clear. E.g.,

$$\frac{P = True}{(If\ P\ x\ y) = x} \qquad \frac{P = False}{(If\ P\ x\ y) = y}$$

15.10 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 (\rightarrow p.427)) can be defined.
- Rich theories can be obtained by a method we see next lecture (\rightarrow p.434).

15.11 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 (\rightarrow p.434) (< 1 week)
- How the different parts of mathematics are encoded in the Isabelle/HOL library (\rightarrow p.463) (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 (\rightarrow p.467)
- Sets (\rightarrow p.476)
- Functions (\rightarrow p.487)
- (Least) fixpoints and induction (\rightarrow p.503)
- (Well-founded) recursion (\rightarrow p.533)
- Arithmetic (\rightarrow p.577)
- Datatypes (\rightarrow p.600)

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 (\rightarrow p.??)
- Z^{395} 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.

³⁹⁵ Z and CSP are specification languages. CSP stands for communicating sequential processes.

Conservative Extensions: Motivation

But first, conservative extensions.

Stage of our course before studying HOL:

- fairly small theories,
- “intuitive” models, (e.g. naïve set theory (→ p.127)),
- but inconsistent (→ p.147) (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.

Reuse: Isabelle supports libraries and retrieval utilities.

Safe, well-understood integration mechanisms: Isabelle supports conservative theory extensions.

Topic of next lecture (➔ p.434).

16 Conservative Theory Extensions

Outline

In the previous lecture (\rightarrow p.401), 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.

16.1 Conservative Theory Extensions: Basics

Some definitions [GM93, Hué]

Definition (theory): A (syntactic) theory T is a triple (\mathcal{B}, Σ, A) , where \mathcal{B} is a type signature (\rightarrow p.369), Σ a signature (\rightarrow p.370) and A a set of axioms³⁹⁶.

Definition (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

³⁹⁶Isabelle-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 (\rightarrow p.49) 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 (\rightarrow p.370) above the line and one formula (\rightarrow p.370) below the line.

$$\Sigma \subseteq \Sigma' \text{ and } A \subseteq A'.$$

Definitions (Cont.)

Definition (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 formulas³⁹⁷ *Th* we have

$$Th(T) = Th(T') \mid_{\Sigma},$$

where \mid_{Σ} filters away all formulas not belonging to Σ .

Counterexample:

$$\overline{\forall f^{\alpha \rightarrow \alpha}. Y f = f (Y f)} \text{ fix}_{398}$$

³⁹⁷The derivable formulas are terms of type *bool* derivable using the inference rules of HOL (\rightarrow p.387). We write $Th(T)$ for the derivable formulas of a theory T .

³⁹⁸Given 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

$$\overline{\forall f^{\alpha \rightarrow \alpha}. Y f = f (Y f)} \text{ fix}$$

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 (\rightarrow p.503).

Consistency Preserved

Definition (consistency): *If T' is a conservative extension of T , then*

$$False \notin Th(T) \Rightarrow False \notin Th(T').$$

Syntactic Schemata for Conservative Extensions

- Constant definition (\rightarrow p.440)
- Type definition (\rightarrow p.445)
- Constant specification
- Type specification

Will look at first two schemata now.

For the other two see [GM93].

16.2 Constant Definition

Definition (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 $c \notin \text{dom}^{399}(\Sigma)$;
- $A' = A \cup \{c = E\}$;
- E does not contain (\rightarrow p.442) c and is closed⁴⁰⁰;
- no subterm of E has a type containing a type variable (\rightarrow p.442) that is not contained in the type of c .

³⁹⁹The domain of Σ , denoted $\text{dom}(\Sigma)$, is $\{c \mid c : A \in \Sigma \text{ for some } A\}$.

Likewise, the domain of Γ , denoted $\text{dom}(\Gamma)$, is $\{x \mid x : A \in \Gamma \text{ for some } A\}$.

Note the abuse of notation (\rightarrow p.176).

⁴⁰⁰A term is closed or ground if it does not contain any free (\rightarrow p.72) variables.

Constant Definitions Are Conservative

Definition (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 conditions⁴⁰¹ 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 (\rightarrow p.375) by only one term, and τ is a type inhabited (\rightarrow p.375) by at least two terms. Then we would have:

$$\begin{aligned} c = c & \quad \text{holds by } \textit{refl} \ (\rightarrow \text{ p.387}) \\ \implies (\exists x^\sigma y^\sigma. x \neq y) &= (\exists x^\tau y^\tau. x \neq y) \\ \implies \textit{False} &= \textit{True} \\ \implies \textit{False} & \end{aligned}$$

This explains definition of \textit{True} ⁴⁰². Other (standard) example later (\rightarrow p.583).

⁴⁰¹By 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 (\rightarrow p.440).

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 (\rightarrow p.198). 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 (\rightarrow p.171).

⁴⁰² \textit{True} is defined as $\lambda x^{\textit{bool}}.x = \lambda x.x$ (\rightarrow p.397) and not $\lambda x^\alpha.x = \lambda x.x$. The definition must be type-closed.

Constant Definition: Examples

Definitions of *True*, *False*, \wedge , \vee , \forall ... (\rightarrow p.396)

Here the original (\rightarrow p.390) Isabelle syntax (**Ex_def** changed (\rightarrow p.390))

```
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 existence⁴⁰³:

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 (\rightarrow p.192); also recall []
syntax (\rightarrow p.193).

⁴⁰³We have never used unique existential quantification ($\exists!$) before. $\exists!x_1, \dots, x_n. \phi(x_1, \dots, x_n)$ is defined as $\exists x_1, \dots, x_n. \phi(x_1, \dots, x_n) \wedge (\forall y_1, \dots, y_n. \phi(y_1, \dots, y_n) \rightarrow x_1 = y_1 \wedge \dots \wedge x_n = y_n)$.

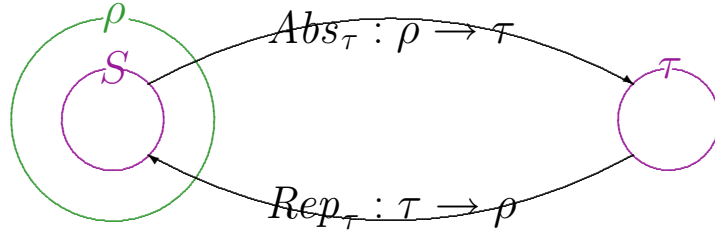
Note that in general $\exists!x. (\exists!y. \phi)$ is not the same as $\exists!xy. \phi$.

16.3 Type Definitions

Type definitions, explained intuitively: we have

- an existing type ρ ;
- a predicate $S : \rho \rightarrow \text{bool}$, defining a non-empty “subset”⁴⁰⁴ of ρ ;
- axioms stating an isomorphism between S and the new type τ .

⁴⁰⁴Although 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 (\rightarrow p.131).



Type Definition: Definition

Definition (type definition): Assume a theory $T = (\mathcal{B}, \Sigma, A)$ and a type ρ and a term S^{405} such that $\Sigma \vdash (\rightarrow \text{p.198}) S : \rho \rightarrow \text{bool}$.

A theory extension $T' = (\mathcal{B}', \Sigma', A')$ of T is a type definition for type τ^{406} (where τ fresh⁴⁰⁷), iff

⁴⁰⁵Here, S is any “predicate” (\rightarrow p.365), i.e., term of type $\rho \rightarrow \text{bool}$, not necessarily a constant.

⁴⁰⁶A type definition is supposed to define a type constructor (\rightarrow p.369) (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 (\rightarrow p.197) (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.

⁴⁰⁷The type constructor τ must not occur in \mathcal{B} .

$$\begin{aligned}
\mathcal{B}' &= \mathcal{B} \uplus^{408} \{\tau\}, \\
\Sigma' &= \Sigma \cup \{Abs_\tau^{409} : \rho \rightarrow \tau, Rep_\tau (\rightarrow p.447) : \tau \rightarrow \rho\} \\
A' &= A \cup \{\forall x. Abs_\tau(Rep_\tau x) = x^{410}, \\
&\quad \forall x. S x \rightarrow Rep_\tau(Abs_\tau x) = x (\rightarrow p.447)\}
\end{aligned}$$

Proof obligation⁴¹¹ $\exists x. S x$ can be proven inside HOL!

⁴⁰⁸The 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.

⁴⁰⁹Of 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” (\rightarrow p.445) 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.

⁴¹⁰The formulas

$$\begin{aligned}
&\forall x. Abs_\tau(Rep_\tau x) = x \\
&\forall x. S x \rightarrow Rep_\tau(Abs_\tau x) = x
\end{aligned}$$

Type Definitions Are Conservative

Definition (type definitions): *Type definitions are conservative.*

Proof see [GM93, pp.230].

state that the “set” S (\rightarrow p.445) and the new type τ are isomorphic. Note that Abs_τ should not be applied to a term not in “set” S (\rightarrow p.445). Therefore we have the premise Sx 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 .

⁴¹¹We have said previously (\rightarrow p.445) that S should be a non-empty “subset” (\rightarrow p.445) of τ . Therefore it must be proven that $\exists x. Sx$. This is related to the semantics (\rightarrow p.372).

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 (\rightarrow p.458). Using this syntax, the “author” of a type definition can either explicitly provide a proof (see **Product_Type.thy** (\rightarrow p.459)), or the proof is so easy that Isabelle can do it automatically (see

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* (\rightarrow p.369) and \rightarrow (\rightarrow p.369), the types in HOL are already very rich.

We now give three examples to convince you.

`Sum_Type.thy` (\rightarrow p.461)).

Example: Typed Sets

General scheme, substituting $\rho \equiv \alpha \rightarrow \text{bool}$ (α is any type variable (\rightarrow p.197)), $\tau \equiv \alpha \text{ set}$ (\rightarrow p.446) (or set (\rightarrow p.446)),
 $S \equiv \lambda x^{\alpha \rightarrow \text{bool}}. \text{True}$

$$\mathcal{B}' = \mathcal{B} \uplus \{\tau \text{ set}\},$$

$$\Sigma' = \Sigma \cup \{ \text{Abs}_{\tau \text{ set}} : \rho(\alpha \rightarrow \text{bool}) \rightarrow \tau \alpha \text{ set}, \\ \text{Rep}_{\tau \text{ set}} : \tau \alpha \text{ set} \rightarrow \rho(\alpha \rightarrow \text{bool}) \}$$

$$A' = A \cup \{ \forall x. \text{Abs}_{\tau \text{ set}}(\text{Rep}_{\tau \text{ set}} x) = x, \\ \forall x. S x \text{True} \rightarrow \text{Rep}_{\tau \text{ set}}(\text{Abs}_{\tau \text{ set}} x) = x \}$$

Simplification since $S \equiv \lambda x. \text{True}$. Proof obligation (\rightarrow p.447):
 $(\exists x. Sx)$ trivial since $(\exists x. \text{True}) = \text{True}$. Inhabitation⁴¹²
 crucial!

⁴¹²We 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 (\rightarrow p.375) for the importance of inhabitation.

Sets: Remarks

Any function $r : \alpha \rightarrow bool$ can be interpreted as a set of α ; r is called characteristic function. That's what $Abs_{set} r$ does; Abs_{set} is a wrapper saying “interpret r as set”.

$S \equiv \lambda x. True$ and so S is trivial⁴¹³ in this case.

⁴¹³We 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” (\rightarrow p.445) 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.

More Constants for Sets

For convenient use of sets, we define more constants:

$$\begin{aligned}
\{x \mid f\ x\} &= \textit{Collect}^{414} f = \textit{Abs}_{\textit{set}} f \\
x \in A &= (\textit{Rep}_{\textit{set}} A)^{415} x \\
A \cup B \ (\rightarrow \text{p.134}) &= \{x \mid x \in A \vee x \in B\} \\
&\vdots
\end{aligned}$$

Consistent set theory⁴¹⁶ adequate for most of mathematics

⁴¹⁴We have seen *Collect* before in the theory file `NSet.thy` (naïve set theory (\rightarrow p.127)).

Collect *f* is the set whose characteristic function (\rightarrow p.451) is *f*. There is also a concrete (\rightarrow p.399) (i.e., according to mathematical practice) syntax $\{x \mid f\ x\}$. It is called set comprehension. The correspondence between the HOAS (\rightarrow p.399) *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 (\rightarrow p.128).

Note also that *Collect* is the same (\rightarrow p.454) as *Abs_{set}* here.

The file `Set.thy` should be contained in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

⁴¹⁵We define

$$x \in A = (\textit{Rep}_{\textit{set}} A) x$$

and computer science.

In Isabelle/HOL however, sets are a special case⁴¹⁷.

Here, sets are just an example to demonstrate type definitions. Later (\rightarrow p.476) we study them for their own sake.

Since Rep_{set} has type (\rightarrow p.450) $\alpha\ set \rightarrow (\alpha \rightarrow bool)$, this means that (\rightarrow p.198) 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 (\rightarrow p.67)).

In the Isabelle theory file **Set.thy** (\rightarrow p.453), 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 (\rightarrow p.454) directly corresponding to Rep_{set} .

⁴¹⁶Typed set theory is a conservative extension (\rightarrow p.448) of HOL and hence consistent (\rightarrow p.438).

Recall the problems with untyped set theory (\rightarrow p.147).

⁴¹⁷We presented a definition of $\alpha\ set$ according to the scheme of type definitions (\rightarrow p.445). However, in Isabelle/HOL (**Set.thy** (\rightarrow p.453)), it is not done exactly like that. The reason will become clear later (\rightarrow p.458).

The type $\alpha\ set$ is defined in Isabelle/HOL in a way which essentially corresponds to the type definition scheme, but

Example: Pairs

Consider type $\alpha \rightarrow \beta \rightarrow \text{bool}$. We can regard a term $f : \alpha \rightarrow \beta \rightarrow \text{bool}$ 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:

is different in the technical details. In particular, there are no constants Abs_{set} and Rep_{set} . Instead, we have $Collect$ (\rightarrow p.453) and the \in -sign (\rightarrow p.453). We will now explain how.

Concerning Abs_{set} , there is no worry, since it corresponds exactly to $Collect$ (\rightarrow p.453).

Rep_{set} is related to the \in -sign (\rightarrow p.453) via

$$x \in A = (Rep_{set}\ A)\ x$$

Let us see that this setup is equivalent to the scheme of type definitions (\rightarrow p.445). There are two axioms in `Set.thy` (\rightarrow p.453):

`axioms`

```
mem_Collect_eq [iff]: "(a : {x. P(x)}) = P(a)"
Collect_mem_eq [simp]: "{x. x:A} = A"
```

- For given a and b , there is exactly one⁴¹⁸ such f (namely, $\lambda x^\alpha y^\beta. x = a \wedge y = b$).
- Some functions of type $\alpha \rightarrow \beta \rightarrow \text{bool}$ represent pairs and others don't (e.g., the function $\lambda xy. \text{True}$ 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 .

We translate these axioms using the definitions (\rightarrow p.452):

$$\begin{aligned}
a \in \{x \mid P x\} &= P a \rightsquigarrow \\
a \in (\text{Collect } P) &= P a \rightsquigarrow \\
a \in (\text{Abs}_{\text{set}} P) &= P a \rightsquigarrow \\
\text{Rep}_{\text{set}}(\text{Abs}_{\text{set}} P) a &= P a \rightsquigarrow \\
\text{Rep}_{\text{set}}(\text{Abs}_{\text{set}} P) &= P
\end{aligned}$$

The last step uses extensionality (\rightarrow p.387).

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* (\rightarrow p.450).

⁴¹⁸When 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 $(\lambda x^\alpha y^\beta. x = a \wedge y = b) =$

Type Definition for Pairs

This gives rise to a type definition where S (\rightarrow p.445) is non-trivial:

$$\begin{aligned}\rho &\equiv \alpha \rightarrow \beta \rightarrow \text{bool} \\ S &\equiv \lambda f^{\alpha \rightarrow \beta \rightarrow \text{bool}}. \exists ab. f = \lambda x^\alpha y^\beta. x = a \wedge y = b \\ \tau &\equiv \alpha \times \beta \qquad (\times \text{ infix})\end{aligned}$$

It is convenient to define a constant **Pair_Rep** (not to be confused with Rep_\times ⁴¹⁹) as $\lambda a^\alpha b^\beta. \lambda x^\alpha y^\beta. x = a \wedge y = b$ ⁴²⁰.

Then **Pair_Rep** $a\ b = \lambda x^\alpha y^\beta. x = a \wedge y = b$.

$(\lambda x^\alpha y^\beta. y = b \wedge x = a)$ is derivable in HOL.

⁴¹⁹ Rep_\times would be the generic name for one of the two isomorphism-defining functions (\rightarrow p.447).

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.

⁴²⁰We 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 .

Now in Isabelle

Isabelle has a special set-based⁴²¹ syntax for type definitions:

```
typedef (T)  
   $\langle typevars \rangle$  "T"  $\langle fixity \rangle$   
  = " $\{x.\phi\}$ "
```

How is this linked to our scheme (\rightarrow p.446):

- the new type is called T' (\rightarrow p.457);
- ρ is the type of x (inferred (\rightarrow p.370));
- S is $\lambda x.\phi$;
- constants (\rightarrow p.457) **Abs** $_T$ and **Rep** $_T$ are automatically generated.

⁴²¹The syntax " $\{x.\phi\}$ " does not just look like a set comprehension (\rightarrow p.453), it is one!

So, since the **typedef** syntax is based on sets, sets themselves could not have been defined using that syntax. This is the reason why in Isabelle/HOL, sets are a special case (\rightarrow p.454) of a type definition.

See **Typedef.thy**, which should be contained in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

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**⁴²² introduces a constant definition. The definition and use of **Pair_Rep** (\rightarrow p.457) is for convenience. There are “two names” (\rightarrow p.457) ***** and **Prod**. See **Product_Type.thy**⁴²³.

⁴²²In Isabelle theory files, **consts** is the keyword preceding a sequence of constant declarations (i.e., this is where the Σ (\rightarrow p.435) is defined), and **defs** is the keyword preceding the axioms that define these constants (i.e., this is where the A (\rightarrow p.435) 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 $id \equiv \lambda x.x$.

⁴²³This file should be contained in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

Example: Sums

An element of $(\alpha, \beta) \text{ sum}^{424}$ 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 (\rightarrow p.455).

⁴²⁴Idea 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 (\rightarrow p.184), 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

datatype $(\alpha, \beta) \text{ sum} = Inl\ \alpha \mid Inr\ \beta$

So an element of $(\alpha, \beta) \text{ sum}$ is either $Inl\ a$ where $a : \alpha$ or $Inr\ b$ where $b : \beta$.

Isabelle Syntax for Sum Example

```
constdefs
  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`⁴²⁵.

How would you define⁴²⁶ a type **even** based on **nat**?

⁴²⁵This file should be contained in your Isabelle distribution.
Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

⁴²⁶Suppose 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 (\rightarrow p.446) 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**,

16.4 Summary on Conservative Extensions

We have seen two schemata:

- Constant definition (\rightarrow p.440): new constant must be defined using old constants. No recursion! Subtle side conditions (\rightarrow p.442) concerning types.
- Type definition (\rightarrow p.445): new type must be isomorphic to a “subset” (\rightarrow p.445) 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 (\rightarrow p.449).

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** (\rightarrow p.457), but we could have used the same name twice as well.

17 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!⁴²⁷

⁴²⁷It 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.

The Roadmap

- Orders (**→** p.467)
- Sets (**→** p.476)
- Functions (**→** p.487)
- (Least) fixpoints and induction (**→** p.503)
- (Well-founded) recursion (**→** p.533)
- Arithmetic (**→** p.577)
- Datatypes (**→** p.600)

18 Orders

The Roadmap

We are looking at how the different parts of mathematics are encoded in the Isabelle/HOL library (\rightarrow p.463).

- Orders
- Sets (\rightarrow p.476)
- Functions (\rightarrow p.487)
- (Least) fixpoints and induction (\rightarrow p.503)
- (Well-founded) recursion (\rightarrow p.533)
- Arithmetic (\rightarrow p.577)
- Datatypes (\rightarrow p.600)

Three Order Classes

We first define a syntactic class (\rightarrow p.192) **ord**. It is the class of types for which symbols $<$ and \leq exist.

We then define two axiomatic classes (\rightarrow p.193) **order** and **linorder** for which $<$ and \leq are required to have certain properties, that of being a partial order (\rightarrow p.116), or a linear order (\rightarrow p.119), resp.

Orders (in HOL.thy⁴²⁸)

```
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` (\rightarrow p.459) syntax and note two uses of $<$ ⁴²⁹.

⁴²⁸ In previous versions of Isabelle (\rightarrow p.390), there used to be a theory file `Ord.thy`. Nowadays orders are defined in `HOL.thy`.

⁴²⁹The line

```
axclass order < ord
```

in the theory file states that `order` is a subclass (\rightarrow p.192) 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 (\rightarrow p.390), it used to be called `term`.

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` (➔ p.470), 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 (➔ p.370).

Monotonicity

In `Ord.thy` (\rightarrow p.470), 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 Theorems⁴³⁰ 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>	$\text{min } x x = x$
<code>le_min_iff_conj</code>	$(z \leq \text{min } x y) = (z \leq x \wedge z \leq y)$

18.1 Summary on Orders

Type classes are a structuring mechanism in Isabelle:

⁴³⁰In the rest of the course, we will mostly be dealing with Isabelle HOL, and so when we speak of a theorem, we ususally mean an Isabelle theorem, i.e., a theorem in Isabelle's metalogic (\rightarrow p.296), what we also call a **thm** (\rightarrow p.252). 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 (\rightarrow p.387) and \forall (\rightarrow p.412). 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 “;”.

- Syntactic classes (\rightarrow p.192) (e.g. $t :: \alpha :: \text{ord}$ as in Haskell [HHPW96]): merely a mechanism to structure visibility of operations.
- Axiomatic classes (\rightarrow p.193) (e.g. $t :: \alpha :: \text{order}$): a mechanism for structuring semantic knowledge⁴³¹ in types (foundation to be discussed later (\rightarrow p.487)).

⁴³¹The Isabelle type system records for any type variable what class constraints (\rightarrow p.192) 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 "($x :: 'a :: \text{order}$) $< y \implies x \leq y$ ";

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** (\rightarrow p.471) can only be used to prove $y \leq y$ 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 ($y \leq y$ holds).

19 Sets

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library (➔ p.463).

- Orders (➔ p.467)
- Sets
- Functions (➔ p.487)
- (Least) fixpoints and induction (➔ p.503)
- (Well-founded) recursion (➔ p.533)
- Arithmetic (➔ p.577)
- Datatypes (➔ p.600)

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 (\rightarrow p.452) to Abs_{set} (\rightarrow p.453) and Rep_{set} (\rightarrow p.453).

Sets Are a Special Case

Recall that the **typedef** (\rightarrow p.458) syntax is based on set comprehension (\rightarrow p.458). Therefore, sets are a special case (\rightarrow p.454) of type definitions.

In deviation from our conservative approach (\rightarrow p.386), sets are axiomatized as follows:

axioms

`mem_Collect_eq [iff]: "(a : {x. P(x)}) = P(a)"`

`Collect_mem_eq [simp]: "{x. x:A} = A"`

One can see though that this is equivalent (\rightarrow p.454) to the type definition scheme (\rightarrow p.445).

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 (\rightarrow p.390) here but only since it is more concise.

In what follows, recall that

$$\{x \mid f x\} = \textit{Collect} \ (\rightarrow \text{p.453}) \ f = \textit{Abs}_{\textit{set}} f$$

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 <= B == ALL x:A. x:B"
Compl_def:     "- A == {x. ~x:A}"
set_diff_def:  "A - B == {x. x:A & ~x:B}"
UNION_def:     "UNION A B == {y. EX x:A. y: B(x)}"
INTER_def:     "INTER A B == {y. ALL x:A. y: B(x)}"
```

Note use of \leq ⁴³² instead of \subseteq !

Nice syntax:

$$\text{UN } x : A. S x \quad \text{or} \quad \bigcup_{x \in A}. S x \quad \text{for} \quad \text{UNION } A S$$

$$\text{INT } x : A. S x \quad \text{or} \quad \bigcap_{x \in A}. S x \quad \text{for} \quad \text{INTER } A S$$

⁴³²Sets are an instance of the type class **ord** (\rightarrow p.467), 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 (\rightarrow p.193) **order**. This is non-obvious and must be proven, which is done not in **Set.thy** itself but in **Fun.thy**, later (\rightarrow p.487). This is a technicality of Isabelle.

Set.thy: Constant Definitions (3)

```
Union_def: "Union S == (UN x:S. x)"
Inter_def: "Inter S == (INT x:S. x)"
Pow_def:    "Pow A == {B. B <= A}"
image_def:  "f`A == {y. EX x:A. y = f(x)}"
```

Nice syntax:

$\bigcup S$ for `Union S`

$\bigcap S$ for `Inter S`

Some Theorems (\rightarrow p.474) 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 (→ p.474) in Set.thy

<code>insert_iff</code>	$(a \in \text{insert } b \ A) = (a = b \vee a \in A)$
<code>image_Un</code>	$f'(A \cup B) = f'A \cup f'B$
<code>Inter_lower</code>	$B \in A \implies \bigcap A \subseteq B$
<code>Inter_greatest</code>	$(\bigwedge X. X \in A \implies C \subseteq X) \implies C \subseteq \bigcap A$

19.1 Summary on Sets

Rich and powerful set theory available in HOL:

- No problems with consistency (→ p.453)
- Weaker than ZFC (→ p.362) (since typed set-theory:)
there is no “union of sets⁴³³”; but: complement-closed⁴³⁴
- Good mechanical support (→ p.249) for many set tautologies (`Fast_tac` (→ p.274), `fast_tac set_cs`, `fast_tac eq_cs`,
... `simp_tac set_ss` (→ p.293) ...)

⁴³³In 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 (→ p.362).

⁴³⁴The 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 (→ p.362), the complement construction is not generally allowed since it opens the door to Russell’s Paradox (→ p.147).

- Powerful basis for many problems in modeling

20 Functions

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library (\rightarrow p.463).

- Orders (\rightarrow p.467)
- Sets (\rightarrow p.476)
- Functions
- (Least) fixpoints and induction (\rightarrow p.503)
- (Well-founded) recursion (\rightarrow p.533)
- Arithmetic (\rightarrow p.577)
- Datatypes (\rightarrow p.600)

Fun.thy

The theory **Fun.thy**⁴³⁵ defines some important notions on functions, such as concatenation, the identity function, the image of a function, etc.

We look at it briefly.

⁴³⁵This file should be contained in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

Fun.thy builds on **Set.thy** (➔ p.453), and it is here that it is proven and used that sets are an instance of the type class **order**.

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` (➔ p.459) syntax.

Instantiating an Axiomatic Class

Sets are partial orders (\rightarrow p.116): `set` is an instance of the axiomatic class `order` (\rightarrow p.469).

For some reason (\rightarrow p.482), 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 obligations⁴³⁶.
- These are discharged⁴³⁷ whenever instance is stated.
- Type-checking (\rightarrow p.475) has access to the established properties.

⁴³⁶To claim that a type is an instance of an axiomatic class (\rightarrow p.193), 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.

⁴³⁷The 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 (\rightarrow p.474) `subset_refl`, `subset_trans`, `subset_antisym`, and `psubset_eq`.

20.1 Conclusion of Orders, Sets, Functions

- Theory says: conservative extensions can be used (➔ p.434) to build consistent libraries.
- Sets as one important package (➔ p.476) of Isabelle/HOL library:
 - Set theory is typed, but very rich and powerfully supported.
 - Sets are instance of **ord** (➔ p.467) and **order** (➔ p.491) type class, demonstrates type classes as structuring mechanism in Isabelle.
- Will see more examples: Isabelle/HOL contains some 10000 **thm** (➔ p.252)'s.

21 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 (➔ p.463).

- Orders (➔ p.467)
- Sets (➔ p.476)
- Functions (➔ p.487)
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic (➔ p.577)
- Datatypes (➔ p.600)

Recursion Based on Set Theory

Current stage of our course:

- On the basis of conservative extensions (\rightarrow p.434), set theory (\rightarrow p.476) 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 definition⁴³⁸.

Then we could easily derive

$$\forall F^{\alpha \rightarrow \alpha}. Y F = F (Y F)^{439}.$$

- Why are we interested in Y ?
- What is the problem with such a definition?

⁴³⁸The axiom

$$Y = \lambda F.F(Y F)$$

is not a constant definition (\rightarrow p.440), since Y occurs again on the right-hand side.

⁴³⁹In words, this says that $Y F$ is a fixpoint of F .

Why Are We Interested in Y ?

First, why are we interested in recursion (solutions to recursive equations⁴⁴⁰)?

- Recursively defined (\rightarrow p.557) functions are solutions of such equations (example: fac ⁴⁴¹).
- Inductively defined (\rightarrow p.522) sets are solutions of such

⁴⁴⁰By 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 (\rightarrow p.440).

⁴⁴¹In the following explanations, any constants like 1 or + or **if-then-else** are intended to have their usual meaning.

A fixpoint combinator (\rightarrow p.496) 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

$$fac\ n = (\mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * fac\ (n - 1)).$$

We now massage this equation a bit. Doing λ -abstraction (\rightarrow p.163) on both sides we get

$$\lambda n. fac\ n = (\lambda n. \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * fac(n - 1))$$

which is the η -conversion (\rightarrow p.165) of

$$fac = (\lambda n. \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * fac(n - 1))$$

which in turn is a β -reduction (\rightarrow p.163) of

$$fac = \underline{((\lambda f. \lambda n. \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * f(n - 1))\ fac)} \quad (3)$$

We are looking for a solution to (3). We abbreviate the underlined expression by Fac . We claim $fac = Y\ Fac$, i.e., it is a solution to (3). Simply replacing fac with $Y\ Fac$ in (3) we get

$$Y\ Fac = Fac\ (Y\ Fac)$$

equations (example: $Fin A^{442}$, all finite subsets of A).

We are interested in Y because it is the mother of all

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).

⁴⁴²We 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  $a \in A$  do  
    foreach  $B \in S$  do  
      add  $(\{a\} \cup B)$  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:

$$Fin\ A = \{\{\}\} \cup \bigcup x \in A. ((\mathbf{insert} \ (\rightarrow \text{p.481})\ x) \, ' (Fin\ A))$$

Recall that $'$ is nice syntax for *image* (\rightarrow p.483), defined in `Set.thy` (\rightarrow p.453).

The above is a β -reduction (\rightarrow p.163) of

$$Fin\ A = (\lambda X. \{\{\}\} \cup \bigcup x \in A. ((\mathbf{insert} \ (\rightarrow \text{p.481})\ x) \, ' X)) (Fin\ A)$$

(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

$$Y\ FA = FA(Y\ FA),$$

recursions. With Y , recursive axioms can be converted⁴⁴³ into constant definitions (\rightarrow p.440).

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.

⁴⁴³Any 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* (\rightarrow p.497) or *Fin A* as an example.

What's the Problem with such an Axiom?

Such a definition would lead to inconsistency (→ p.437).

This is not surprising because not all functions have a fixpoint.

Therefore we only consider special forms (→ p.575) of fixpoint combinators.

We consider two approaches: Least fixpoints (→ p.503) (Tarski) and well-founded (→ p.533) orderings.

22 Least Fixpoints

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library (\rightarrow p.463).

- Orders (\rightarrow p.467)
- Sets (\rightarrow p.476)
- Functions (\rightarrow p.487)
- (Least) fixpoints and induction
- (Well-founded) recursion (\rightarrow p.533)
- Arithmetic (\rightarrow p.577)
- Datatypes (\rightarrow p.600)

22.1 First Approach: Least Fixpoints (Tarski)

- Restriction: F (\rightarrow p.496) 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 monotone⁴⁴⁴, and: this is not obvious (requires non-trivial proof)!

This leads us towards recursion and induction (\rightarrow p.522).

⁴⁴⁴A function f is monotone w.r.t. a partial order (\rightarrow p.116) \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.

```

Lfp = Product_Type +
constdefs
  lfp :: ['a set => 'a set] => 'a set
  "lfp(f) == Inter({u. f(u) <= u})"

```

Note:

- \Rightarrow is function type arrow (\rightarrow p.192).
- \leq (\subseteq) (\rightarrow p.482)) is a partial order (\rightarrow p.116).
- `Inter` (\bigcap) (\rightarrow p.483) gives a “minimum”: $\forall A \in S. (\bigcap S) \subseteq A$.
- Definition of *lfp* is conservative (\rightarrow p.440). That’s fine.
But is it a fixpoint combinator? (\rightarrow p.512)

⁴⁴⁵These files should be contained in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

22.2 Tarski's Fixpoint Theorem

Definition (Tarski): *If f is monotone (\rightarrow p.505), then $\text{lfp } f = f (\text{lfp } f)$.*

In Isabelle, the theorem is shown in `Lfp.ML` (\rightarrow p.506) 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 $f A \subseteq A$ then $\text{lfp } f \subseteq A$.

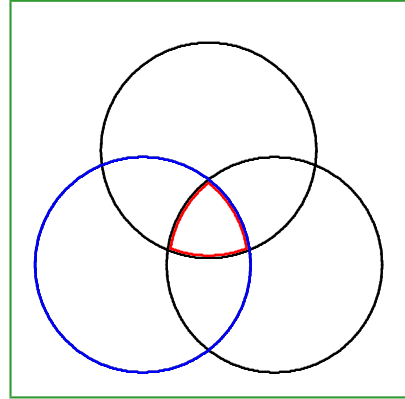
The box denotes “the set” α ⁴⁴⁶. The three circles⁴⁴⁷ denote the sets A for which $f A \subseteq A$.

By definition (\rightarrow p.506), *lfp f* is the intersection.

Pick an A for which $f A \subseteq A$.

Clearly, *lfp f* \subseteq A.

Or as proof tree (\rightarrow p.514).



⁴⁴⁶ 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** (\rightarrow p.481). **UNIV** of type τ **set** is the set containing all terms of type τ .

⁴⁴⁷In 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 $f A \subseteq A$.

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 $A \subseteq \text{lfp } f$.

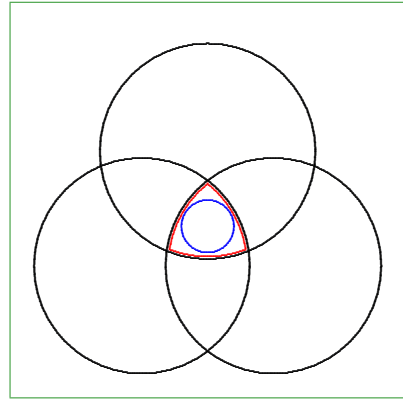
The three circles denote the sets U for which $f U \subseteq U$.

By hypothesis, $A \subseteq U$ for each U (1st, 2nd, 3rd ...).

By definition (\rightarrow p.506), *lfp* f is the intersection.

Clearly, $A \subseteq \text{lfp } f$.

Or as proof tree (\rightarrow p.515).



Tarski's Fixpoint Theorem (3)

Claim 3: If f is monotone (\rightarrow p.505) 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 (\rightarrow p.508), $\text{lfp } f \subseteq U$.

$f U \subseteq U$ (hypothesis).

$f(\text{lfp } f) \subseteq f U$

(monotonicity (\rightarrow p.505)).

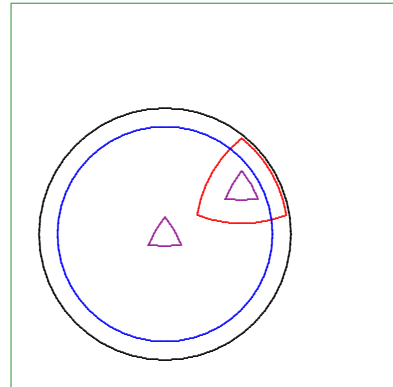
$f(\text{lfp } f) \subseteq U$

(transitivity (\rightarrow p.487) of \subseteq).

Claim 3* shown.

By Claim 2 (\rightarrow p.509) (letting $A :=$

$f(\text{lfp } f)$), $f(\text{lfp } f) \subseteq \text{lfp } f$.



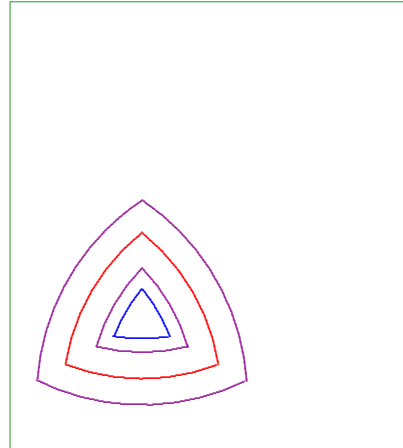
Tarski's Fixpoint Theorem (4)

Claim 4: If f is monotone (\rightarrow p.505) then $\text{lfp } f \subseteq f(\text{lfp } f)$.

By Claim 3 (\rightarrow p.510), $f(\text{lfp } f) \subseteq \text{lfp } f$.

By monotonicity (\rightarrow p.505),
 $f(f(\text{lfp } f)) \subseteq f(\text{lfp } f)$.

By Claim 1 (\rightarrow p.508) (letting $A := f(\text{lfp } f)$), $\text{lfp } f \subseteq f(\text{lfp } f)$.



Or as proof tree (\rightarrow p.517).

Tarski's Fixpoint Theorem: QED

Claim 3 (\rightarrow p.510) ($lfp\ f \subseteq f(lfp\ f)$) and Claim 4 (\rightarrow p.511) ($f(lfp\ f) \subseteq lfp\ f$) together give the result:

If f is monotone, then $lfp\ f = f(lfp\ f)$.

So under appropriate conditions, lfp is a fixpoint combinator (\rightarrow p.497).
We will later reuse Claim 1 (\rightarrow p.508).

Alternative: A Natural-Deduction Style Proof

The proof can also be presented in natural deduction style (➔ p.25).

Tarski's Fixpoint Theorem (1)

Claim 1 (“*lfp* lower bound”): If $f A \subseteq A$ then $\text{lfp } f \subseteq A$.

$$\begin{array}{c}
 \frac{[f A \subseteq A]^1}{A \in \{u. fu \subseteq u\}} \text{CollectI } (\rightarrow \text{ p.484}) \\
 \frac{\bigcap \{u. fu \subseteq u\} \subseteq A}{\text{lfp } f \subseteq A} \text{Inter_lower } (\rightarrow \text{ p.484}) \\
 \frac{\text{lfp } f \subseteq A}{f A \subseteq A \rightarrow \text{lfp } f \subseteq A} \text{Def. } \text{lfp } (\rightarrow \text{ p.506}) \\
 \frac{f A \subseteq A \rightarrow \text{lfp } f \subseteq A}{f A \subseteq A \rightarrow \text{lfp } f \subseteq A} \rightarrow\text{-I } (\rightarrow \text{ p.395})^1
 \end{array}$$

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 $A \subseteq \textit{lfp } f$.

$$\begin{array}{c}
 \frac{[\forall x. f x \subseteq x \rightarrow A \subseteq x]^1}{\forall x. x \in \{u. f u \subseteq u\} \rightarrow A \subseteq x} \textit{subst } (\rightarrow \text{ p.395}), \textit{CollectI } (\rightarrow \text{ p.484}) \\
 \frac{\quad}{A \subseteq \cap \{u. f u \subseteq u\}} \textit{Inter_greatest } (\rightarrow \text{ p.484}) \\
 \frac{\quad}{A \subseteq \textit{lfp } f} \textit{Def. lfp } (\rightarrow \text{ p.506}) \\
 \hline
 (\forall x. f x \subseteq x \rightarrow A \subseteq x) \rightarrow A \subseteq \textit{lfp } f \quad \rightarrow\text{-}I (\rightarrow \text{ p.395})^1
 \end{array}$$

Tarski's Fixpoint Theorem (3)

Claim 3: If f is monotone (\rightarrow p.505) then $f(\text{lfp } f) \subseteq \text{lfp } f$.

$$\begin{array}{c}
 \frac{[fx \subseteq x]^2}{[mono\ f]^1 \quad \text{lfp } f \subseteq x} \\
 \hline
 \frac{f(\text{lfp } f) \subseteq f\ x \quad [fx \subseteq x]^2}{f(\text{lfp } f) \subseteq x} \text{ order_trans } (\rightarrow \text{ p.471}) \\
 \hline
 \frac{\forall x. fx \subseteq x \rightarrow f(\text{lfp } f) \subseteq x}{f(\text{lfp } f) \subseteq \text{lfp } f} \forall\text{-}I\ (\rightarrow \text{ p.390}), \rightarrow\text{-}I\ (\rightarrow \text{ p.395})^2 \\
 \hline
 \frac{f(\text{lfp } f) \subseteq \text{lfp } f}{mono\ f \rightarrow f(\text{lfp } f) \subseteq \text{lfp } f} \text{lfp_greatest } (\rightarrow \text{ p.515}), \rightarrow\text{-}E\ (\rightarrow \text{ p.395}) \\
 \hline
 \frac{}{mono\ f \rightarrow f(\text{lfp } f) \subseteq \text{lfp } f} \rightarrow\text{-}I\ (\rightarrow \text{ p.395})^1
 \end{array}$$

Tarski's Fixpoint Theorem (4)

Claim 4: If f is monotone (\rightarrow p.505) then $lfp\ f \subseteq f(lfp\ f)$.

$$\begin{array}{c}
 \frac{[mono\ f]^1}{f(lfp\ f) \subseteq lfp\ f} \text{ Claim 3 } (\rightarrow \text{ p.516}), \rightarrow\text{-}E (\rightarrow \text{ p.395}) \\
 \frac{\frac{f(f(lfp\ f)) \subseteq f(lfp\ f)}{lfp\ f \subseteq f(lfp\ f)} \text{ lfp_lowerbound } (\rightarrow \text{ p.514}), \rightarrow\text{-}E (\rightarrow \text{ p.395})}{mono\ f \rightarrow lfp\ f \subseteq f(lfp\ f)} \text{ monoD } (\rightarrow \text{ p.474}) \rightarrow\text{-}I (\rightarrow \text{ p.395})^1
 \end{array}$$

Completing Proof Tree

$$\begin{array}{c}
 \frac{[mono\ f]^1}{lfp\ f \subseteq f(lfp\ f)} \text{ Claim 4 } (\rightarrow \text{ p.517}) \quad \frac{[mono\ f]^1}{f(lfp\ f) \subseteq lfp\ f} \text{ Claim 3 } (\rightarrow \text{ p.516}) \\
 \hline
 \frac{lfp\ f = f(lfp\ f)}{mono\ f \rightarrow lfp\ f = f(lfp\ f)} \rightarrow\text{-}I\ (\rightarrow \text{ p.395})^1
 \end{array}$$

22.3 Induction Based on Lfp.thy

Definition (lfp induction): *If*

- *f is monotone (\rightarrow p.505), 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 Isabelle⁴⁴⁸, 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 (\rightarrow p.507).

⁴⁴⁸The theorem is phrased a bit differently in the “mathematical” version we give here and in the Isabelle version (see `Lfp.ML` (\rightarrow p.506)). This is convenient for the graphical illustration of the proof.

The “mathematical phrasing” corresponding closely to the Isabelle version would be the following:

Definition (Induct (alternative)): *If*

- *$a \in \text{lfp } f$, and*
- *f is monotone (\rightarrow p.505), 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:

Definition (Induct (alternative)): *If*

- *$a \in \text{lfp } f$, and*
- *f is monotone (\rightarrow p.505), and*

Showing `lfp_induct`

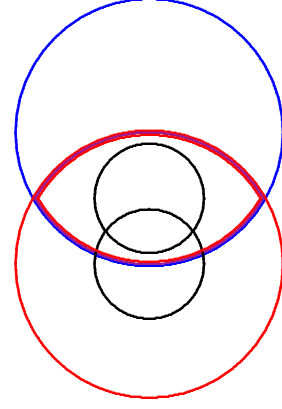
Circles denote $\text{lfp } f$ and $\{x \mid P x\}$.

By monotonicity⁴⁴⁹,
 $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq f(\text{lfp } f)$. By
 Tarski (\rightarrow p.507), $\text{lfp } f = f(\text{lfp } f)$. Hence
 $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \text{lfp } f$.

By hypothesis (\rightarrow p.519), $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \{x \mid P x\}$, 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 1⁴⁵⁰, $\text{lfp } f \subseteq \text{lfp } f \cap \{x \mid P x\}$
 and so⁴⁵¹ $\text{lfp } f = \text{lfp } f \cap \{x \mid P x\}$.

Conclusion: $\text{lfp } f \subseteq \{x \mid P x\}$.



-
- $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \{x \mid P x\}$

then $P a$ holds.

Definition (Induct (alternative)): *If*

- f is monotone (\rightarrow p.505), 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 $P x$.

⁴⁴⁹ $\text{lfp } f \cap \{x \mid P x\} \subseteq \text{lfp } f$, so by
 monotonicity (\rightarrow p.519), $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq f(\text{lfp } f)$.

⁴⁵⁰We have just seen $f(\text{lfp } f \cap \{x \mid P x\}) \subseteq \text{lfp } f \cap \{x \mid P x\}$.

By Claim 1 (\rightarrow p.508) (setting $A := \text{lfp } f \cap \{x \mid P x\}$),
 this implies $\text{lfp}(f) \subseteq \text{lfp } f \cap \{x \mid P x\}$.

⁴⁵¹We 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 (\rightarrow p.487).

Approximating Fixpoints

Looking ahead: Suppose we have the set \mathbb{N} of natural numbers (the type is formally introduced later (\rightarrow p.586)).

The theorem **approx**

$$(\forall S. f(\bigcup_{n \in \mathbb{N}} (f^n \{ \})) = \bigcup_{n \in \mathbb{N}} (f^n \{ \})) \implies \bigcup_{n \in \mathbb{N}} (f^n \{ \}) = \text{lfp } f$$

shows a way of approximating $\text{lfp } f$, which is important for algorithmic solutions⁴⁵² (e.g. in program analysis).

There will be an exercise on this.

⁴⁵²The theorem

$$(\forall S. f(\bigcup_{n \in \mathbb{N}} (f^n \{ \})) = \bigcup_{n \in \mathbb{N}} (f^n \{ \})) \implies \bigcup_{n \in \mathbb{N}} (f^n \{ \}) = \text{lfp } f$$

says that under a certain condition, $\text{lfp } f$ 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 $\bigcup_{i \leq k} f^i \{ \} \subseteq \bigcup_{n \in \mathbb{N}} f^n \{ \}$.

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 (\rightarrow p.497) $S = F(S)$.

Induction and recursion are two faces of the same coin.

Lfp.thy for Inductive Definitions (→ p.506)

Least fixpoints are for building inductive definitions of sets in a definitional way⁴⁵³: $S := \text{lfp } F$.

This is obviously (→ p.506) well-defined, so why this fuss about monotonicity (→ p.505) and Tarski (→ p.507)?

Tarski (→ p.507) allows us to exploit the equation $\text{lfp } f = f(\text{lfp } f)$ in proofs about S ! That's what lfp is all about.

⁴⁵³Recall why we were interested (→ p.497) in fixpoints.

The problem with Y (→ p.502) is that it leads to inconsistency (→ p.437) (and of course (→ p.441), the definition of Y is not a constant definition (→ p.440)/conservative extension.).

The definition of lfp is conservative.

And in appropriate situations, it can be used to define recursive functions.

Compared to Y (→ p.493), the type of lfp is restricted (→ p.505).

This restriction means that there is no obvious way to use lfp for defining recursive numeric functions such as fac (→ p.497).

Example (Revisited) (→ p.499)

The set of all finite subsets of a set A :

$$\textit{Fin } A = \textit{lfp } F$$

where $F = \lambda X. \{\{\}\} \cup \bigcup x \in A. ((\textit{insert } (\rightarrow \text{p.481}) x) ' X)$.

Thus we can do using *lfp* what we would have wanted to do using Y (→ p.499).

To show: F is monotone⁴⁵⁴!

In the Isabelle library⁴⁵⁵, this is done a bit differently⁴⁵⁶.

There will be an exercise on this.

⁴⁵⁴This proof is of course done in Isabelle.

⁴⁵⁵This file should be contained in your Isabelle distribution.

Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

⁴⁵⁶Above, 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 *Finites* is defined. It has polymorphic type $\alpha \textit{ set set}$. We have $A \in \textit{Finites}$ if and only if A is a finite set. However, it would be wrong to think of *Finites* as one single set that contains all finite sets. Instead, for each τ , there is a polymorphic instance (→ p.369) of *Finites* of type $\tau \textit{ set set}$ containing all finite sets of element type τ .

In `Finite_Set.thy` we find the lines

22.4 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 (\rightarrow p.524) into expression using *lfp*.

```
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, **Finites** is a structure (\rightarrow p.526) (module), and **defs** is a value (component) of this structure (\rightarrow p.526).

As a sanity-check, consider the type (\rightarrow p.368) of this expression. The expression $insert\ a\ A$ forces A to be of type $\tau\ set$ for some τ and a to be of type τ . Next, $insert\ a\ A$ is of type $\tau\ set$, and hence x is also of type $\tau\ set$. Moreover, the expression $A \in S$ forces S to be of type $\tau\ set\ set$. The expression $\{x \mid x = \{\} \vee (\exists A\ a. x = insert\ a\ A \wedge A \in S)\}$ is of type $\tau\ set\ set$. Next, G is of type $\tau\ set\ set \rightarrow \tau\ set\ set$, and so finally, $Finities$ is of type $\tau\ set\ set$. 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** (\rightarrow p.252) 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

Package relies on proven lemma⁴⁵⁷ `lfp_unfold` (→ p.507).

rather

`thm "C_def"`

You should be aware of such problems, but we do not treat them in this course.

⁴⁵⁷If you look around in the `ML`-files of the Isabelle/HOL library, you might not find any uses of `lfp_unfold` (→ p.507), 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.

Technical Support for Inductive Definitions

Support important in practice since many constructions are based on inductively defined sets (datatypes (\rightarrow p.600), ...). Support provided for:

- Automatic proof of monotonicity
- Automatic proof of induction rule (\rightarrow p.519), for example⁴⁵⁸:

$$\llbracket xa \in \textit{Fin } A; P \{\}; \bigwedge ab. \llbracket a \in A; b \in \textit{Fin } A; P b \rrbracket \implies P(\textit{insert } a b) \rrbracket \implies P xa$$

⁴⁵⁸The theorem

$$\llbracket xa \in \textit{Fin } A; P \{\}; \bigwedge ab. \llbracket a \in A; b \in \textit{Fin } A; P b \rrbracket \implies P(\textit{insert } a b) \rrbracket \implies P xa$$

is an instance of the general induction scheme (\rightarrow p.519). That is to say, if we take the general induction scheme `lfp_induct` (\rightarrow p.519)

$$\llbracket a \in \textit{lfp } f; \textit{mono } f; \bigwedge x. x \in f(\textit{lfp } f \cap \{x.P x\}) \implies P x \rrbracket \implies P a$$

and instantiate f to $\lambda X. \{\{\}\} \cup \bigcup x \in A. ((\textit{insert } x) ' X)$ (\rightarrow p.524) 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 (\rightarrow p.524) is shown by Isabelle once and for all. This is one aspect of what we mean by special proof support for inductive definitions (\rightarrow p.525).

The least fixpoint of the functional is $\textit{Fin } A$ (the set of finite subsets of A) in this case.

This works also for mutually recursive⁴⁵⁹ definitions, co-inductive⁴⁶⁰ definitions, ...

⁴⁵⁹Two functions f and g are mutually recursive if f is defined in terms of g and vice versa.

⁴⁶⁰Co-induction is a construction analogous to induction but using greatest fixpoints.

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 (\rightarrow p.??) study a small imperative programming language.

22.5 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 (\rightarrow p.502), but we have, by conservative extension (\rightarrow p.440), least fixpoints for defining sets.

There is an induction scheme (lfp induction (\rightarrow p.519)) for proving theorems about an inductively defined set.

Restriction of F to set type (\rightarrow p.505) ($\alpha \text{ set} \rightarrow \alpha \text{ set}$) means that least fixpoints are not generally suitable for defining functions ...

23 Well-Founded Recursion

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library (➔ p.463).

- Orders (➔ p.467)
- Sets (➔ p.476)
- Functions (➔ p.487)
- (Least) fixpoints and induction (➔ p.503)
- (Well-founded) recursion
- Arithmetic (➔ p.577)
- Datatypes (➔ p.600)

Well-Founded Recursion

After least fixpoints (\rightarrow p.503), 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.

23.1 Prerequisite: Relations

We need some standard operations on binary relations (sets of pairs (\rightarrow p.455)), such as converse, composition, image of a set and a relation, the identity relation, ...

These are provided by `Relation.thy`⁴⁶¹.

⁴⁶¹ This file should be contained in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

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 O 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` (➔ p.487).

23.2 Prerequisite: Closures

We need the transitive, as well as the reflexive transitive closure of a relation.

These are provided by `Transitive_Closure.thy`⁴⁶².

How would you define those inductively, ad-hoc?⁴⁶³

⁴⁶² This file should be contained in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

⁴⁶³ r^* is the smallest set such that:

- $Id \ (\rightarrow \text{p.536}) \subseteq r^*$;
- if $r' \subseteq r^*$ then $r' \cup r \circ r' \subseteq r^*$.

Or, in line with the schema for inductive definitions (\rightarrow p.522):

- $\emptyset \subseteq r^*$;
- if $r' \subseteq r^*$ then $(\lambda s. Id \ (\rightarrow \text{p.536}) \cup (r \circ s))r' \subseteq r^*$.

The latter form corresponds to the definition in `Transitive_Closure.thy` (\rightarrow p.538).

The definition of r^+ is similar.

Transitive_Closure.thy (Fragment)

```
consts
  rtrancl :: "('a * 'a) set => ('a * 'a) set"
              ("(_^*)" [1000] 999)

inductive "r^*"
  intros
    rtrancl_refl [...]: "(a, a) : r^*"
    rtrancl_into_rtrancl [...]: "(a, b) : r^* ==>
                                   (b, c) : r ==> (a, c) : r^*"

```

Transitive_Closure.thy (Fragment Cont.)

```
consts
  trancl :: "('a * 'a) set => ('a * 'a) set"
           ("(_^+)" [1000] 999)

inductive "r^+"
  intros
    r_into_trancl [...]: "(a, b) : r ==>
                           (a, b) : r^+"
    trancl_into_trancl [...]: "(a, b) : r^+ ==>
                               (b, c) : r ==> (a,c) : r^+"
```

23.3 Well-Founded Orderings

Defined in `Wellfounded_Recursion.thy`⁴⁶⁴.

```
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 scheme⁴⁶⁵.

⁴⁶⁴This file should be contained in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

In older versions the file used to be called `WF.thy`.

⁴⁶⁵For 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 (\rightarrow p.440) (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 $\phi(r, P)$.

Example: Is \emptyset well-founded⁴⁶⁶?

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 $\phi(r, P)$.

⁴⁶⁶The definition of wf is:

Let's instantiate r to \emptyset .

$wf(r\emptyset) \equiv (\forall P. True(\forall x. (\forall y. True False(y, x) \in r\emptyset \rightarrow P(y)) \rightarrow P(x)) \rightarrow$

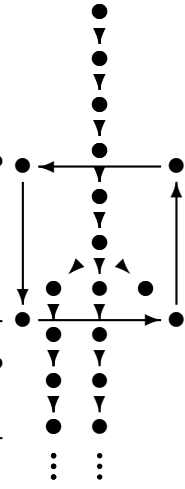
So the empty set is well-founded.

Intuition of Well-Foundedness

Intuition of wf : All descending chains are finite.

But: Cannot express infinity; must look for alternatives⁴⁶⁷.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$ (\rightarrow p.535)?
- 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 formalized⁴⁶⁸ (already in previous point).



⁴⁶⁷We 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.

⁴⁶⁸In 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).

The Characterisation

All these attempts are just necessary but not sufficient conditions for well-foundedness.

Here is a characterization⁴⁶⁹:

$$(\forall Q. x \in Q \rightarrow (\exists z \in Q. \forall y. (y, z) \in r \rightarrow y \notin Q))$$

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.

⁴⁶⁹The 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” (\rightarrow p.540) definition is somewhat indirect since it defines well-foundedness by what one can do with it (\rightarrow p.540).

A Theorem⁴⁷⁰ on the Empty Set

`wf_empty wf {}`

Proof sketch: `wf_empty`: substitute r into definition, simplify.

⁴⁷⁰The theorems (\rightarrow p.474) we present here are proven in `Wellfounded Recursion.ML`.

This file should be contained in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

`http://isabelle.in.tum.de/library/`

but in older versions the file used to be called `WF.ML`

A Theorem (→ p.544) for Induction

By massage⁴⁷¹ 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**

$$\llbracket wf\ r; \bigwedge x.\forall y.(y, x) \in r \rightarrow P y \Longrightarrow P x \rrbracket \Longrightarrow P a.$$

This is a form suitable for doing induction proofs in Isabelle.

⁴⁷¹As far as the induction principle is concerned, **induct_wf** states the same as the very definition of **wf** (→ p.540). All that happens is that some explicit universal object-level quantifiers are removed (→ p.474) and the according variables are (implicitly) universally quantified on the meta-level, and some shifting (→ p.474) from object-level implications to meta-level implications using **mp**. This is why we dare say “logical massage”. See **Wellfounded Recursion.ML** (→ p.544).

Induction Theorem as Proof Rule

The Isabelle theorem `wf_induct` (\rightarrow p.545)

$$\llbracket wf\ r; \bigwedge x. \forall y. (y, x) \in r \rightarrow P\ y \Longrightarrow P\ x \rrbracket \Longrightarrow P\ a.$$

as proof rule (\rightarrow p.25):

$$\frac{wf\ r \qquad \begin{array}{c} [\forall y. (y, x) \in r \rightarrow P\ y] \\ \vdots \\ P\ x \end{array}}{P\ a} \text{wf_induct}$$

A Theorem (→ p.544) on Antisymmetry

wf_not_sym $\llbracket wf\ r; (a, x) \in r \rrbracket \implies (x, a) \notin r$

Proof sketch:

$$\begin{array}{c}
 [\forall y. (y, x) \in r \rightarrow (\forall z. (y, z) \in r \rightarrow (z, y) \notin r)] \\
 \vdots \\
 wf\ r \quad \forall z. (x, z) \in r \rightarrow (z, x) \notin r \\
 \hline
 \forall z. (a, z) \in r \rightarrow (z, a) \notin r \quad \text{wf_induct}
 \end{array}$$

The induction part needs classical reasoning (→ p.267). We will now give it at a level of detail that shows the essential reasoning but hides all the swapping (→ p.267) involved in the Isabelle proof.

A variation will be done as exercise.

The Induction Part in More Detail

$$\frac{\frac{\forall y.(y, x) \in r \rightarrow (\forall z.(y, z) \in r \rightarrow (z, y) \notin r)}{(w, x) \in r \rightarrow (\forall z.(w, z) \in r \rightarrow (z, w) \notin r)} \forall\text{-E}}{\frac{(w, x) \notin r \vee (\forall z.(w, z) \in r \rightarrow (z, w) \notin r)}{\equiv} \phi} \text{(c)}^{472}$$

“(c)” stands for classical reasoning steps.

$$\frac{\phi \quad \frac{[(w, x) \notin r]^1}{(x, w) \in r \rightarrow (w, x) \notin r} \text{impl}^2 \quad \frac{\frac{[\forall z.(w, z) \in r \rightarrow (z, w) \notin r]^1}{\forall z.(z, w) \in r \rightarrow (w, z) \notin r} \text{(c)}^{473}}{(x, w) \in r \rightarrow (w, x) \notin r} \forall\text{-E}}{\frac{(x, w) \in r \rightarrow (w, x) \notin r}{\forall z.(x, z) \in r \rightarrow (z, x) \notin r} \forall\text{-I}} \text{disjE}^1$$

Theorems (\rightarrow p.544) on Absence of Cycles

`wf_not_refl` $wf\ r \implies (a, a) \notin r$

`wf_trancl` $wf\ r \implies wf(r^+)$

`wf_acyclic` $wf\ r \implies acyclic\ r$

($acyclic\ r \equiv \forall x.(x, x) \notin r^+$ (\rightarrow p.540))

`wf_not_refl`: Corollary of `wf_not_sym`.

Proof sketch: `wf_trancl`: Uses induction.

`wf_acyclic`: Apply `wf_not_refl` and `wf_trancl`

Ergo: Definition of wf (\rightarrow p.540) 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 $\phi \equiv (\exists x.\forall y.(y, x) \notin r^+)$:

$$\frac{\frac{\frac{wf(r)}{wf(r^+)} \bullet (\rightarrow p.549) \quad \frac{\frac{\phi \vee \neg\phi}{\phi} \bullet (\rightarrow p.426) \quad [\phi]^2}{\phi} \quad \frac{\frac{\frac{[\neg\phi]^2}{\forall x.\exists y.(y, x) \in r^+} \cdots \frac{[\neg\phi]^2 \quad [\frac{\forall w.(w, v) \in r^+ \rightarrow \phi}{\neg\exists w.(w, v) \in r^+}]^1}{\neg\exists w.(w, v) \in r^+}}{\phi} \quad \frac{False}{\phi} \quad FalseE (\rightarrow p.415)}{disjE (\rightarrow p.426)^2} \quad \frac{\phi}{\phi} \quad wf_minimalwf_induct (\rightarrow p.545)^1}{\phi}$$

⁴⁷⁴In 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 (\rightarrow p.552))

In detail, the sub-proof looks as follows:

$$\frac{\neg\phi \quad \frac{\frac{[\exists w.(w, v) \in r^+]^3}{\phi} \quad \frac{\frac{[(w, v) \in r^+]^4 \quad \frac{\forall w.(w, v) \in r^+ \rightarrow \phi}{(w, v) \in r^+ \rightarrow \phi} \quad spec (\rightarrow p.38)}{mp (\rightarrow p.38)} \quad \phi}{existsE (\rightarrow p.421)^4} \quad \phi}{notE (\rightarrow p.418)} \quad \frac{False}{\neg\exists w.(w, v) \in r^+} \quad notI (\rightarrow p.417)^3$$

This is what we must construct.

Note “special case”: w and v do not occur in ϕ !

This is **wf_tranc1**.

We now try a proof by case distinction on ϕ .

Classical (\rightarrow p.359) reasoning.

Using some elementary equivalences⁴⁷⁵.

This step works for any ϕ . Think semantically or check!

It is routine to derive *False*.

This completes the proof by case distinction ...

...and the proof by induction.

Remarks on the Proof

We used an instance of **wf_induct** (\rightarrow p.545), 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 .

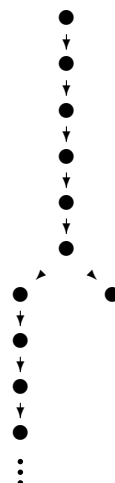
Still this is a “proper” induction proof: Although ϕ does not contain the “induction variables”, the proof does depend on the actual form of ϕ ! (Try doing it without induction . . .)

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 (\rightarrow p.81) are very subtle in this proof. Underlines the importance of machine-checked proofs.

Remarks on wf_minimal

Ergo: Definition of **wf** (\rightarrow p.540) fulfills the condition corresponding to our first attempt (\rightarrow p.542) 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 (→ p.544)

The theorem `wf_eq_minimal` is characterization of well-foundedness (→ p.

$$wf\ r = (\forall Q. x \in Q \rightarrow (\exists z \in Q. \forall y. (y, z) \in r \rightarrow y \notin Q))$$

Proof uses split =⁴⁷⁶, `wf_def` (→ p.540), rest routine.

Ergo: Definition of `wf` (→ p.540) meets textbook definitions “every non-empty set Q has a minimal element in r ”.

⁴⁷⁶By this we simply mean to split a proof of $\phi = \psi$ into two proofs $\phi \implies \psi$ and $\psi \implies \phi$.

A Theorem (\rightarrow p.544) 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** (\rightarrow p.550) and **wf_subset** (\rightarrow p.555).

This implies $wf\ r \implies \forall p. p \subseteq r \rightarrow \exists x. \forall y. (x, y) \notin p$ (\rightarrow p.542).

Ergo: Definition of **wf** (\rightarrow p.540) fulfills the condition corresponding to our second attempt (\rightarrow p.542) of characterizing well-foundedness using minimal elements.

However, this formalization still (\rightarrow p.553) had a problem: The minimum could be an isolated element, unrelated to the subrelation.

23.4 Defining Recursive Functions

Idea of well-founded recursion: Wish to define f by recursive equation (\rightarrow p.497) $f = e$, e.g. (\rightarrow p.497)

$$fac = (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * fac(n - 1))$$

Define $F = \lambda f. e$, e.g. (α -conversion (\rightarrow p.165) of what you have seen (\rightarrow p.498))

$$Fac = (\lambda fac. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * fac(n - 1))$$

We say: F is the functional defining f .

Recall (\rightarrow p.497) that $Y F$ would solve $f = e$, but we don't have (\rightarrow p.502) 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 .

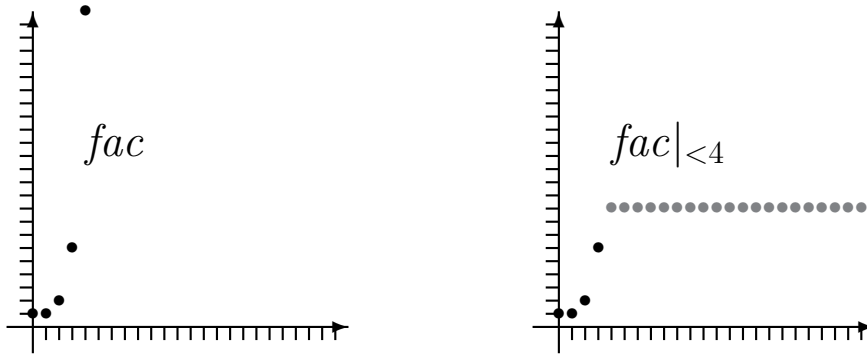
If F is coherent, then we would expect that for any a ,

$$f\ a = (F\ f)\ a = (F\ f|_{<a})\ a. \quad (5)$$

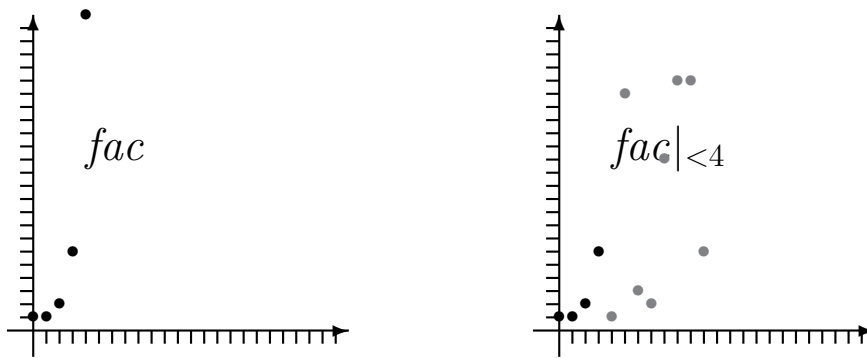
It's not that we are ultimately interested in constructing such a “bad” f , but our formalization of well-founded recursion defines coherence by the fact that one could use such a “bad” f , i.e., via (5).

“Bad” f ’s: Example

Consider fac (\rightarrow p.497). On the right-hand side, we show one possibility⁴⁷⁷ for $fac|_{<4}$:



⁴⁷⁷For 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 (\rightarrow p.381) for HOL (with the extensions we consider here). The way that arbitrary elements are formalized in `HOL.thy` (\rightarrow p.390), 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


```

cut (in Wellfounded_Recursion.thy (→ p.540))
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` (→ p.560) is defined in `HOL.thy` (→ p.390).

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 (→ p.381)).

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.

Theorems (\rightarrow p.544) 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 (➔ p.540))

Auxiliary 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”⁴⁷⁸.

Show that $(4, 24) \in (wfrec_rel\ ' < ' Fac)!$

Now let us really turn $wfrec_rel\ R\ F$ into a function ...

⁴⁷⁸When 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 $(t, s) \in r$.

wfrec (**in** Wellfounded_Recursion.thy (\rightarrow p.540))

```
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 $x.P\ x$ ⁴⁷⁹ picks the unique a such that $P\ a$ holds, if it exists. We don't care what it does otherwise (see **HOL.thy** (\rightarrow p.390)).

⁴⁷⁹The operator **THE** is similar to the Hilbert operator (\rightarrow p.370), 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.

wfrec Explained

$wfrec\ R\ F \equiv$

$\lambda x.\text{THE } y.(x, y) \in wfrec_rel\ R(\lambda f x.\textcolor{red}{F}(\textcolor{red}{cut}\ f\ R\ x)\ x)$

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.

$\text{wfrec}\ R\ F$ is the recursive function defined by functional F .

The “Fixpoint” Theorem (→ p.544)

wfrec $wf\ r \implies wfrec\ r\ H\ a = H(cut(wfrec\ r\ H)\ r\ a)\ a$

Note that **wfrec** is used here both as a name of a constant (defined above (→ p.565)) and a theorem.

So if r is well-founded and H is coherent, we have (by (5))

$$wfrec\ r\ H\ a = H(wfrec\ r\ H)\ a$$

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 (→ p.497).

23.5 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 (\rightarrow p.586)).

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}"
```


Defining 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"
```

Recursive in first argument⁴⁸⁰.

```
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"
```

Recursive in second argument.

480

```
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 (\rightarrow p.573), but conceptually, the above is a definition of the **add** function.

Note that **add** is a function of type $nat \rightarrow nat \rightarrow nat$ (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)))
```

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 $-$ (\rightarrow p.569).

The functions are recursive in one argument (just like **add** (\rightarrow p.569)).

By the definitions (of *wfrec* (\rightarrow p.565) 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.

Theorems (\rightarrow p.544) 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.

23.6 Conclusion on Well-founded Recursion

Well-founded recursion allows us to define recursive functions in HOL and thus reason about computations.

We can derive recursive theorems (\rightarrow p.474) that can be used for rewriting just like in a functional programming language.

Isabelle Package for Primitive Recursion

For primitive recursion⁴⁸¹, finding a well-founded ordering is simple enough for automation⁴⁸²!

Examples (use **nat** (→ p.586) and **case** (→ p.592)-syntax):

...

⁴⁸¹A 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].

⁴⁸²The **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.

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)"

23.7 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 (\rightarrow p.502), but we have, by conservative extension (\rightarrow p.440):

- Least fixpoints for defining sets (\rightarrow p.503);
- well-founded orders for defining functions (\rightarrow p.533).

Both concepts come with induction schemes (lfp induction (\rightarrow p.519) and definition of well-foundedness (\rightarrow p.540)) 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.

24 Arithmetic

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library (➔ p.463).

- Orders (➔ p.467)
- Sets (➔ p.476)
- Functions (➔ p.487)
- (Least) fixpoints and induction (➔ p.503)
- (Well-founded) recursion (➔ p.533)
- Arithmetic
- Datatypes (➔ p.600)

Motivation

Current stage of our course:

- On the basis of conservative embeddings, set theory (→ p.476) can be built safely.
- Inductive sets (→ p.524) can be defined using least fixpoints (→ p.503) and suitably supported by Isabelle (→ p.525).
- Well-founded orderings (→ p.533) can be defined without referring to infinity (→ p.542). Recursive functions can be based on these. Needs inductive sets (→ p.563) though. Support by Isabelle (→ p.573) provided.

Next important topic: arithmetic.

Which Approach to Take?

- Purely definitional (\rightarrow p.434)?

Not possible with eight basic rules (\rightarrow p.387) (cannot enforce infinity⁴⁸³ of HOL model)!

- Heavily axiomatic? I.e., we state natural numbers by Peano axioms⁴⁸⁴ and claim analogous axioms for any other number type?

Insecure!

- Minimally axiomatic? We construct an infinite set, and define numbers etc. as inductive subset (\rightarrow p.503)?

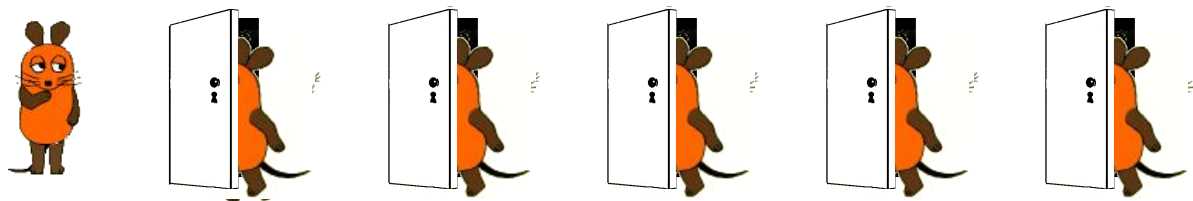
Yes. Finally use infinity (\rightarrow p.372) axiom.

24.1 What is Infinity? Cantor's Hotel

⁴⁸³Our 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.

⁴⁸⁴The Peano axioms include

- $0 \in nat$
- $\forall x. x \in nat \rightarrow Suc(x) \in nat$
- $\forall x. Suc(x) \neq 0$
- $\forall P. (P(0) \wedge \forall n. (P(n) \rightarrow P(Suc(n)))) \rightarrow \forall n. P(n)$



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 forward⁴⁸⁵, the new guest walks towards the first room, they turn around, enter their new rooms. The doors close, all guests are accomodated.

⁴⁸⁵This means, there must be a successor function on rooms. To each room, it assigns the “next” room.

Axiom of Infinity

The axiomatic core⁴⁸⁶ of datatypes (and hence, numbers⁴⁸⁷):

$$\exists f :: (ind \rightarrow ind). \textit{injective } f \wedge \neg \textit{surjective } f \quad \textit{infty} \quad (\rightarrow \text{p.388})$$

where

$$\begin{aligned} \textit{injective}^{488} f &= \forall xy. f\ x = f\ y \rightarrow x = y \\ \textit{surjective } f &= \forall y. \exists x. y = f\ x \end{aligned}$$

Forces *ind* to be “infinite type” (\rightarrow p.372) (called “*I*” in [Chu40]).

We will see soon (\rightarrow p.586) how this is done in Isabelle.

⁴⁸⁶Note that theoretically, it is not needed to add the infinity axiom (or some equivalent formulation (\rightarrow p.586)) 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.

⁴⁸⁷The natural numbers can be built as an algebraic datatype by having a constant 0 and a term constructor *Suc* (for successor).

⁴⁸⁸These constants (actually called *inj* and *sur* (\rightarrow p.584)) are defined in **Fun.thy** (\rightarrow p.489).

24.2 Type-Closed Conservative Extensions

Why must conservative extensions be type-closed? (the standard example)

Consider $H \equiv \exists f :: \alpha \Rightarrow \alpha. \textit{injective } f \wedge \neg \textit{surjective } f$

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)

$(H \equiv \exists f :: \alpha \Rightarrow \alpha.injective\ f \wedge \neg surjective\ f)$

$$\begin{aligned}
 & H = H \quad \text{holds by } refl \ (\blackrightarrow \text{ p.387}) \\
 \Rightarrow & \exists f :: bool \Rightarrow bool.inj^{489} f \wedge \neg sur\ f = \\
 & \quad \exists f :: ind \Rightarrow ind.inj\ f \wedge \neg sur\ f \\
 \Rightarrow & False = True \\
 \Rightarrow & False
 \end{aligned}$$

(unfolding H using two different type instantiations, and then using axiom of infinity (\blackrightarrow p.388) and the fact that there are only finitely many functions on $bool$).

⁴⁸⁹We use *inj* and *sur* as abbreviations for *injective* and *surjective*.

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 (➔ p.442).

24.3 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 (\rightarrow p.388) is formulated by defining a constant *Suc_Rep* having the two required properties.

inj (\rightarrow p.584) and *sur* (\rightarrow p.584) are defined in `Fun.thy` (\rightarrow p.489).

Think of `Zero_Rep`, `Suc_Rep` as provisional 0, successor.

Defining the Set *Nat*

Want to define new type *nat*. How?

Must define a set isomorphic (\rightarrow p.445) to the natural numbers. How?

By induction using the **inductive** syntax (\rightarrow p.525):

```
inductive Nat
```

```
intros
```

```
Zero_RepI: "Zero_Rep : Nat"
```

```
Suc_RepI: "i : Nat ==> Suc_Rep i : Nat"
```

Translated by Isabelle to:

$$Nat = lfp (\lambda X. \{Zero_Rep\} \cup (Suc_Rep \, X))$$

Defining the Type *nat*

Now we have the set *Nat*. What next?

Define the type *nat*, isomorphic to *Nat*, using the **typedef** (\rightarrow p.458) syntax:

```
typedef (open Nat)
  nat = "Nat" by (rule exI, rule Nat.Zero_RepI)
```

After these two steps⁴⁹⁰ we have the type *nat*.

490

Note the two ingredients for defining the type **nat**:

- An inductively defined set (\rightarrow p.519) **Nat**, i.e., a set defined as fixpoint of a monotone function. In Isabelle (**Nat.thy** (\rightarrow p.590)), the **inductive** syntax (\rightarrow p.525) is used for this purpose. This automatically generates an induction rule (\rightarrow p.529) for the set.
- A type definition (\rightarrow p.445) based on this set, defined using the **typedef** syntax (\rightarrow p.458).

Recall (\rightarrow p.458) that this process automatically generates the two constants **Abs_Nat** (\rightarrow p.457) and **Rep_Nat** (\rightarrow p.457).

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

Constants in *nat*

Moreover, define⁴⁹¹:

```
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}"
```

`Nat.thy` (→ p.590).

⁴⁹¹Based on the generic constants **Abs_Nat** (→ p.457) and **Rep_Nat** (→ p.457), we define all the constants that we need to work conveniently with **nat**, most importantly, 0 and **Suc**.

Some Theorems (→ p.474) in `Nat.thy`⁴⁹²

`nat_induct` $\llbracket P\ 0; \bigwedge n. P\ n \implies P\ (Suc\ n) \rrbracket \implies P\ n$

`diff_induct` $\llbracket \bigwedge x. P\ x\ 0; \bigwedge y. P\ 0\ (Suc\ y);$
 $\bigwedge xy. P\ x\ y \implies P\ (Suc\ x)\ (Suc\ y) \rrbracket$
 $\implies P\ m\ n$

We can now exploit that *nat* is defined based on a set (→ p.522) defined using least fixpoints (→ p.504). In particular, `nat_induct` follows (but not “automatically”! (→ p.588)) from the `induct` (→ p.519) theorem (→ p.474) of `Lfp` (→ p.506).

⁴⁹²This file should be contained in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

Nat (\rightarrow p.590) and Well-Founded Orders

Examples of theorems (\rightarrow p.474) involving well-founded orders (\rightarrow p.533):

`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` (→ p.590) defines rich theory on *nat*. Uses `primrec` (→ p.573) syntax for defining recursive functions (→ p.533), and `case`⁴⁹³ 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)"
```

⁴⁹³The `case` statement for `nat` is a function of type $nat \Rightarrow (nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat$. `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” (→ p.399)) of the original (“abstract”) syntax. In the original syntax it would read `case 0 ($\lambda x.x$) (n - m)`.

Some Theorems (\rightarrow p.474) in Nat (\rightarrow p.590)

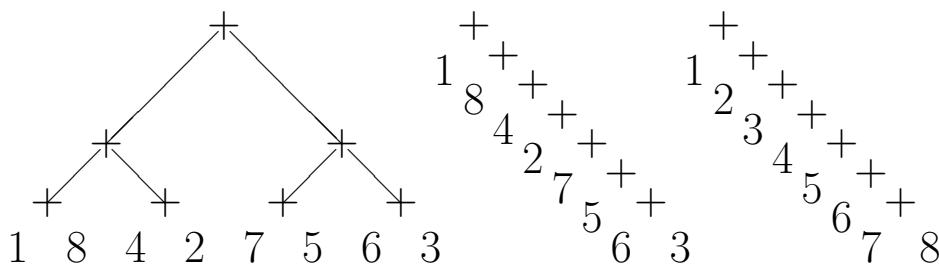
`add_0_right` $m + 0 = m$
`add_ac` $m + n + k = m + (n + k)$
 $m + n = n + m$
 $x + (y + z) = y + (x + z)$
`mult_ac` $m * n * k = m * (n * k)$
 $m * n = n * m$
 $x * (y * z) = y * (x * z)$

Note third part⁴⁹⁴ of `add_ac`, `mult_ac`, respectively.

Technically, `add_ac` and `mult_ac` are lists of `thm` (\rightarrow p.252)'s.

⁴⁹⁴The theorems $x + (y + z) = y + (x + z)$ and $x * (y * z) = y * (x * z)$ are called left-commutation laws and are crucial for (ordered (\rightarrow p.285)) rewriting (\rightarrow p.276).

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 (\rightarrow p.282), for an appropriately chosen term ordering (\rightarrow p.285).



Proof of add_0_right

$$\begin{array}{c}
 \text{add_0} \\
 \hline
 0 + 0 = 0
 \end{array}
 \quad
 \begin{array}{c}
 \text{add_Suc} \\
 \hline
 \text{Suc } n + 0 = \text{Suc}(n + 0)
 \end{array}
 \quad
 \text{sym}
 \quad
 \begin{array}{c}
 [n + 0 = n]^1 \\
 \hline
 \text{Suc}(n + 0) = \text{Suc } n
 \end{array}
 \quad
 \text{arg_cong}
 \quad
 \begin{array}{c}
 \text{subst} \\
 \hline
 \text{Suc } n + 0 = \text{Suc } n
 \end{array}
 \quad
 \begin{array}{c}
 \text{add_0_rightnat_induct}^1 \\
 \hline
 m + 0 = m
 \end{array}$$

Note that $\text{Suc } n + 0 = \text{Suc}(n + 0)$ is an instance of $\text{Suc } m + n = \text{Suc}(m + n)$.

24.4 Integers

The integers are implemented⁴⁹⁵ as equivalence classes⁴⁹⁶ over $\text{nat} \times \text{nat}$.

```
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)
```

⁴⁹⁵The file should be contained in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

⁴⁹⁶Recall the general concept of an equivalence relation (\rightarrow p.107). 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

Some Theorems (→ p.474) in IntArith

<code>zminus_zadd_distrib</code>	$-(z + w) = -z + -w$
<code>zminus_zminus</code>	$-(-z) = z$
<code>zadd_ac</code>	$z1 + z2 + z3 = z1 + (z2 + z3)$
	$z + w = w + z$
	$x + (y + z) = y + (x + z)$
<code>zmult_ac</code>	$z1 * z2 * z3 = z1 * (z2 * z3)$
	$z * w = w * z$
	$z1 * (z2 * z3) = z2 * (z1 * z3)$

Compare to *nat* theorems (→ p.593).

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 (→ p.445) the set of all pairs of naturals (which corresponds to the **UNIV** (→ p.481) constant on the type $nat \times nat$ (→ p.455)) 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**.

24.5 Further Number Theories

- Binary Integers (`Integ/Bin.thy`, for fast computation)
- Rational Numbers (`Real/PRat.thy`)
- Reals⁴⁹⁷ (`Real/PReal.thy`: based on Dedekind-cuts of rationals [Ded])
- Hyperreals (`Real/RealDef.thy` for non-standard analysis)⁴⁹⁸
- Machine numbers (floats); see work for Intel’s Pentium-IV; built in HOL-light [Har98, Har00]

⁴⁹⁷The 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.

⁴⁹⁸In 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 [FP98]. We just mention this here to say that Isabelle/HOL is used for “cutting-edge” mathematics and not just toy examples.

24.6 Conclusion on Arithmetic

Using conservative extensions (\rightarrow p.434) in HOL, we can build

- the naturals (\rightarrow p.586) (as type definition (\rightarrow p.445) based on *ind*), and
- higher number theories (\rightarrow p.597) (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 systems⁴⁹⁹.

⁴⁹⁹Hybrid systems is a field in software engineering concerned with using finite automata for controlling physical systems such as ABS in cars etc.

The methodological overhead can be tackled by powerful mechanical support, since many proof-tasks are routine.

25 Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library (➔ p.463).

- Orders (➔ p.467)
- Sets (➔ p.476)
- Functions (➔ p.487)
- (Least) fixpoints and induction (➔ p.503)
- (Well-founded) recursion (➔ p.533)
- Arithmetic (➔ p.577)
- Datatypes

What Are Datatypes?

We have seen types, but what are data⁵⁰⁰types?

- Order 0 (\rightarrow p.228) (no \rightarrow in type).
- Terms defined by finite set of term constructors (\rightarrow p.278).
- Typically inductive definition.
- Term constructed by syntactic rule is unique.

Counterexample: α *set* (\rightarrow p.476).

⁵⁰⁰We have seen types, but what are datatypes?

First of all, a datatype must be of order 0 (\rightarrow p.228), 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} (\rightarrow p.170). But if we have polymorphism, it just means that the type must not contain \rightarrow . E.g., α *list* could be a datatype. However, when one describes a datatype, one would usually speak about generic instances such as α *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 (\rightarrow p.278) that have τ as result type. At least one term constructor should just have type τ . E.g., $Nil : \alpha \text{ list}$ and $Cons : \alpha \rightarrow (\alpha \text{ list}) \rightarrow \alpha \text{ list}$ 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.

Datatypes: Motivation

Last lecture (\rightarrow p.577): Construction of natural numbers (\rightarrow p.586).

How can we build “datatypes (\rightarrow p.602)” (as in ML [Pau96])?

Can we base the construction of arbitrary datatypes on the natural numbers?

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 (\rightarrow p.582), 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:

α *set* (\rightarrow p.476). Sets are not a datatype:

1. While the type α *set* does not contain an \rightarrow , it is isomorphic (\rightarrow p.450) 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 $\text{Abs}_{\text{set}} f$ (\rightarrow p.451) (alternatively: $\text{Collect } f$ (\rightarrow p.452)) 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 $\text{Insert}(h, S)$ is a set. But then $\text{Insert}(1, \{\})$ would be different from $\text{Insert}(1, \text{Insert}(1, \{\}))$, 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 con-

- Well, yes — by Gödelization⁵⁰¹. But that wouldn't be automatable easily ...
- Instead: We use S-expressions rather than *nat* (→ p.586).

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`⁵⁰² and [Wen99].

tradition to the inductive definition (→ p.522) 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.

⁵⁰¹In 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 (→ p.185).

⁵⁰²This file should be contained in your Isabelle distribution.

S-Expressions as Basis

In the end we want to have datatypes such as lists (\rightarrow p.278) 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.

25.1 S-Expressions

LISP-like S-expressions⁵⁰³ are a kind of of binary trees. We call the type α *dtree*. This uses $\alpha + nat$ (\rightarrow p.460).

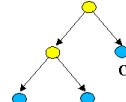
Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

⁵⁰³The 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*.



This is encoded as a set of “nodes”⁵⁰⁴ (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 (\rightarrow p.445) of α *dtree* uses such an encoding.

As always with parametric polymorphism (\rightarrow p.185), 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 (\rightarrow p.185) 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 (\rightarrow p.460). So a leaf label has type $\alpha + \text{nat}$ (written (α, nat) *sum* (\rightarrow p.460) before), and it has the form either *Inl* (\rightarrow p.460)(*a*) for some $a :: \alpha$, or *Inr* (\rightarrow p.460)(*n*) for some $n :: \text{nat}$.

⁵⁰⁴The set

$$\{(\langle 0, 0 \rangle, a), (\langle 0, 1 \rangle, b), (\langle 1 \rangle, c)\}$$

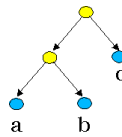
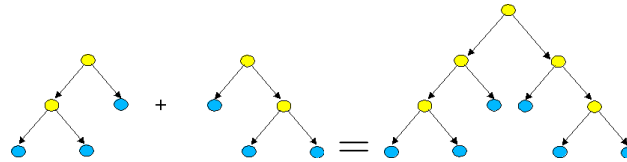
represents the tree

Building Trees

- $Atom(n)$ ⁵⁰⁵



- $Scons\ X\ Y$ ⁵⁰⁶



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 $\langle p_0, \dots, p_n \rangle$.

⁵⁰⁵ $Atom$ takes a leaf label (\rightarrow p.607) and turns it into a (simplest possible) S-expression (\rightarrow p.606) (tree).

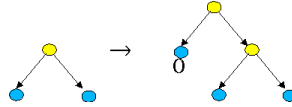
So it has type $\alpha + nat \Rightarrow \alpha\ dtree$.

⁵⁰⁶ $Scons$ takes two S-expressions (\rightarrow p.606) and creates a new S-expression as illustrated below:

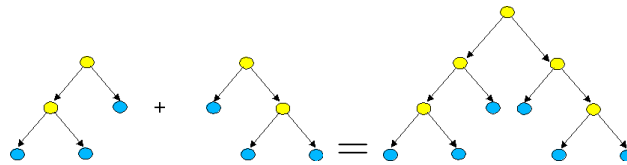
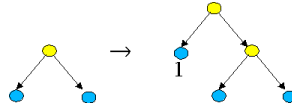
Tagging Trees

We want to tag an S-expression by either 0 or 1. This can be done by “*Scons*” (\rightarrow p.608)-ing it with an S-expression consisting of an administration label (\rightarrow p.607). By convention, the tag is to the left.

- **In0_def** $In0(X) \equiv Scons\ Atom(Inr\ (\rightarrow\text{ p.460})(0))(X)$



- **In1_def** $In1(X) \equiv Scons\ Atom(Inr\ (\rightarrow\text{ p.460})(1))(X)$



So it has type $[\alpha\ dtree, \alpha\ dtree]$ (\rightarrow p.193) $\Rightarrow \alpha\ dtree$.

Products and Sums on Sets of S-Expressions

Product of two sets A and B of S-expressions: All *Scons* (\rightarrow p.608)-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 (\rightarrow p.609) and S-expressions in B have been tagged 1 (\rightarrow p.609), so that one can tell where they come from.

$$\text{usum_def} \quad \text{usum } A \ B \equiv In0 \text{ ' }^{507} A \cup In1 \text{ ' } B$$

⁵⁰⁷Recall that ' denotes the image (\rightarrow p.483) of a function applied to a set.

Some Properties of Trees and Tree Sets

- *Atom*, *In0*, *In1*, *Scons* are⁵⁰⁸ injective (\rightarrow p.388).
- *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 $[(\alpha \text{ dtree}) \text{ set}, (\alpha \text{ dtree}) \text{ set}] (\rightarrow \text{p.1 } (\alpha \text{ dtree}) \text{ set})$.
- *uprod* and *usum* are monotone (\rightarrow p.505).
- Tree sets represent a universe that is closed under products and sums⁵⁰⁹ combined with arbitrary applications of *lfp* (\rightarrow p.504).

Reminder: we simplified!

⁵⁰⁸This 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 (\rightarrow p.602). A datatype consists of terms constructed using term constructors (\rightarrow p.278) and is uniquely defined by what it is syntactically (one also says that terms are generated freely using the constructors). For example, (\rightarrow p.582) 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.

⁵⁰⁹Given 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,

25.2 Lists in Isabelle

Similar to the construction of *nat* (\rightarrow p.588), we first construct a set of S-expressions having the “structure of lists”. We start by defining “provisional” (\rightarrow p.586) 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 expect⁵¹⁰ *Cons* to have, and how does *CONS* compare? Must wrap list elements by *Atom* \circ *Inl*.

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 (\rightarrow p.503), 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 (\rightarrow p.505), and note as well that slight complications arise from the fact that *usum* and *uprod* have type $[(\alpha \text{ dtree}) \text{ set}, (\alpha \text{ dtree}) \text{ set}] (\rightarrow$ p.193) $\Rightarrow (\alpha \text{ dtree}) \text{ set}$ rather than $(\alpha \text{ dtree}) \text{ set} \Rightarrow (\alpha \text{ dtree}) \text{ set}$.

⁵¹⁰*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-

Lists as S-Expressions: Intuition

Examples of how lists would be represented as S-expressions:

$$\begin{array}{rcl}
 Nil^{511} & & [] \\
 In0(Atom(Inr\ 0)) & & \\
 \hline
 Cons(7, Nil) & & [7] \\
 CONS\ (Atom(Inl\ 7))\ In0(Atom(Inr\ 0)) & & \\
 \hline
 Cons(5, Cons(7, Nil)) & & [5, 7] \\
 CONS\ (Atom(Inl\ 5)) & & \\
 (CONS\ (Atom(Inl\ 7))\ In0(Atom(Inr\ 0))) & &
 \end{array}$$

Now let's construct the S-expressions having this form.

expression) and a “list element”, we must first wrap the list element by $Atom \circ Inl$, so that it becomes an S-expression.

⁵¹¹ 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 $[]$, $[7]$, $[5, 7]$.

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`⁵¹² for how it’s really done!

⁵¹²This file should be contained in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

<http://isabelle.in.tum.de/library/>

Defining the “Real” List Type

We now apply the type definition mechanism (\rightarrow p.445) using the `typedef` (\rightarrow p.458) syntax. How do we define A formally?

```
typedef (List)
  'a list =
    "list(range (Atom o Inl)) :: 'a dtree set"
  by ...
```

Choosing A as $\text{range}(Atom \circ Inl)$ together with the explicit type declaration forces A to be the set containing all $Atom(Inl\ t)$, for each $t :: \alpha$.

Example of a definition of a polymorphic (\rightarrow p.185) 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 (\rightarrow p.458), Isabelle provides the **datatype** syntax to support the construction (\rightarrow p.445) 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*⁵¹³?

⁵¹³The **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 (\rightarrow p.588).

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* (\rightarrow p.607) 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*.

26 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 (➔ p.463):

- Orders (➔ p.467)
- Sets (➔ p.476)
- Functions (➔ p.487)
- (Least) fixpoints and induction (➔ p.503)
- (Well-founded) recursion (➔ p.533)
- Arithmetic (➔ p.577)
- Datatypes (➔ p.600)

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 (\rightarrow p.??)
- Denotational semantics and functional languages
- Object-oriented languages (Java-Light ...)

27 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 (\rightarrow p.627);
- red-black trees (\rightarrow p.632).

This is just to stimulate you to look for more applications on your own!

27.1 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 illegal⁵¹⁴ 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.

⁵¹⁴By “illegal memory access”, we mean access to regions not assigned to the program.

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 limited⁵¹⁵.
- 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.

⁵¹⁵The 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.

27.2 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 [IEE85]. 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:

$$\begin{aligned} \text{format } (E, p, N) = \\ \{x \mid \exists s \, e \, k. \, s < 2 \wedge e < E \wedge k < 2^p \wedge x = (-1)^s \cdot 2^e \cdot k / 2^N\} \end{aligned}$$

Rounding

Rounding takes a real to a representable real nearby. E.g. rounding up:

$$\begin{aligned} \text{round } \text{fmt } x = \epsilon a. \quad & a \in \text{format } \text{fmt} \wedge a \leq x \wedge \\ & \forall b \in \text{format } \text{fmt}. b \leq x \rightarrow b \leq a \end{aligned}$$

Formalization of the Standard [IEE85].

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.

27.3 Red-Black Trees

Red-black trees are trees that can be used for implementing sets/dictionaries, just like AVL trees. To formulate “balancedness” 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 Kimmig⁵¹⁶ 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.

⁵¹⁶Angelika 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).

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]!

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.

- [Ded] Richard Dedekind. Cuts of Rationals. ??, ??
- [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.
- [FP98] Jacques D. Fleuriot and Lawrence C. Paulson. A combination of nonstandard analysis and ge-

ometry theorem proving, with application to newton's principia. In Claude Kirchner and Hélène Kirchner, editors, Proceedings of the 15th CADE, volume 1421 of LNCS, pages 3–16. Springer-Verlag, 1998.

- [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 logische 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 Sys-

teme. 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 Environments. Cambridge University Press, 1993.
- [HR04] Michael Huth and Mark Ryan. Logic in Computer Science. Modelling and Reasoning about Systems. Cambridge University Press, 2nd edition edition, 2004.
- [HS86] J. Roger Hindley and Jonathan P. Seldin. Introduction to Combinators and λ -Calculus.

Cambridge University Press, 1986. Republished in [HS90].

- [HS90] J. Roger Hindley and Jonathan P. Seldin. Introduction to Combinators and λ -Calculus. Cambridge University Press, 1990.
- [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 Universit  t 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 byte-code verifiers. Theoretical Computer Science, 3(298):583–626, 2003.
- [LP81] Harry R. Lewis and Christos H. Papadimitriou. Elements 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].
- [Nip03] Tobias Nipkow. Java bytecode verification. Journal 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

Reasoning and its Applications: Essays in Honor of Larry Wos, chapter 3. MIT Press, 1997.

- [Pau05] Lawrence C. Paulson. The Isabelle Reference Manual. Computer Laboratory, University of Cambridge, October 2005.
- [Pea18] Guisepe Peano. ?? ??, 18??
- [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.
- [Tho91] Simon Thompson. Type Theory and Functional Programming. Addison-Wesley, 1991.
- [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 Functional 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. Con-

tains 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.

- [WR25] Alfred N. Whitehead and Bertrand Russell.
Principia Mathematica, volume 1. Cambridge
University Press, 1925. 2nd edition.
- [Zer19] Ernst Zermelo. ?? ??, 19??