Computer-Supported Modeling and Reasoning

Jochen Hoenicke

WS13/14

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

Making logic come to life by making it run on a computer, using the tool Isabelle. Applications in

- Mathematics[1] (Hilbert's program)

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

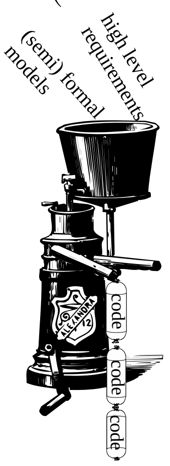
1. Is mathematics complete in the sense that every statement can be proved or disproved?
2. Is mathematics consistent in the sense that no statement can be proved both true and false?
3. Is mathematics decidable in the sense that there exists a definite method to determine the truth or falsity of any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

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

Thanks to the the incompleteness theorem of Gödel (1931) and the undecidability of first-order logic shown by Church and Turing (1936–37) we know now that his dream will never be realized completely. This makes it a never-ending task to find partial answers to Hilbert's questions.

- program and hardware verification[2]

(For the impacient: some Isabelle/HOL applications (➜ p.654))



For more details:

– Panel talk by Moshe Vardi

– Lecture by Michael J. O'Donnell

– Article by Stephen G. Simpson

– Original works Über das Unendliche and Die Grundlagen der Mathematik [vH67]

– Some quotations shedding light on Gödel's incompleteness theorem

– Eric Weisstein's world of mathematics explaining Gödel's incompleteness theorem

[2] Verification is the process of formally proving that a program has the desired properties. To this end, it is necessary to define a specification language in which the desired properties can be formulated, i.e. specified. One must define a semantics for this language as well as for the program. These semantics must be linked in such a way that it is meaningful

What this Course is Useful for

After attending this course, you might . . .

- pursue an academic career focused on the topic of this course or some other topic in formal methods;
- apply formal methods in a company[3] like Intel or Gemplus;
- work in a different area in academia or industry; even then, understanding mathematical and logical reasoning improves understanding of how to build correct systems and do more rigorous proofs.

to say: "Program X makes formula Φ true".

[3] The last 20 years have seen spectacular hardware and software failures (e.g. the Pentium bug) and the birth of a new discipline: the verification engineer.

Overview: Four Parts

1. Logics[4] (propositional, first-order, higher-order): appr. 6 units
2. Metalogics[5] (Isabelle): appr. 2 units
3. Modeling mathematics and computer science (programming languages) in higher-order logic: appr. 6 units
4. Two case studies in formalizing a theory[6] (functional and imperative programming): appr. 2 units

Presentation roughly follows this structure.

[4] The word logic is used in a wider and a narrower sense.

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

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

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

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

Theories will be considered in more detail later (➜ p.112).

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

Requirements

- Some knowledge of logic[7] is useful for this course, but 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! Also, you must follow the course each week, or you will quickly get lost. It is hard in the beginning but the rewards are large.
- Being familiar with basic Linux commands is very helpful.

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

– semantics (interpretations) of logics; and

– correctness and completeness of calculi.

As an introduction we recommend [vD80].

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.
- Validity[8] means: no counterexample. Validity independent of content. Depends on form of the expressions ⇒ can make patterns explicit by replacing words by symbols

From if A then B and A it follows that B.➠ $$\frac{A \to B \quad A}{B}$$

[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.

- What about[9]

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

[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”.

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 formalized[12] as derivation rules, e.g.

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

Rules are grafted together to build trees called derivations.

This defines a proof system[13] in the style of natural deduction.

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

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

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

We study formalization in more detail later (➜ p.228).

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

We call the rules in that particular set basic rules. Later we will see one can also derive (➜ p.44) rules.

2.2 Formalizing Propositional Logic

- We must formalize
  1. Language[14] and semantics (➜ p.20)
  2. Deductive system
- Here we will focus on formalizing the deductive machinery and say little about metatheorems[15] (soundness and completeness[16]).
- For labs we will carry out proofs using the Isabelle System.

[14] By language we mean the language of formulae. We can also say that we define the (object) logic. Here "logic" is used in the narrower sense (➜ p.7).

[15] A metatheorem is a theorem about a proof system, as opposed to a theorem derived within the proof system. The statement "proof system XYZ is sound" is a metatheorem.

[16] A proof system is sound if only valid (➜ p.10) propositions can be derived in it.

A proof system is complete if all valid (➜ p.10) propositions can be derived in it.

2.3 Propositional Logic: Language

Let a set V of (propositional) variables[17] be given. L_P, the[18] language of propositional logic, is defined by the following grammar[19] ($X \in V$):

$$P \ ::= \ X \ \mid \bot\,^{20} \mid (P \wedge{}^{21} P) \mid \\ (P \vee P) \mid (P \to P) \mid ((\neg P)^{22})$$

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

This will be different in logics that we will learn about later (➜ p.62).

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

Later (➜ p.67), we will be more fussy about this point.

[19] A notation like

$$P \ ::= \ X \ \mid \bot \mid \ (P \wedge P) \ \mid \ (P \vee P) \ \mid \ (P \to P) \ \mid \ (\neg P))$$

$$T \ ::= \ x \ \mid \ f^n(\underbrace{T, \ldots, T}_{n \text{ times}})$$

$$F ::= \ldots \ \mid \ p^n(\underbrace{T, \ldots, T}_{n \text{ times}}) \ \mid \ \forall x.\, F \ \mid \ \exists x.\, F$$

$$e \ ::= \ x \ \mid \ c \ \mid \ (ee) \ \mid \ (\lambda x.\, e)$$

$$\tau ::= T \ \mid \ \tau \to \tau$$

$$e \ ::= \ x \ \mid \ c \ \mid \ (ee) \ \mid \ (\lambda x^\tau.\, e)$$

$$P ::= x \ \mid \ \neg P \ \mid \ P \wedge P \ \mid \ P \to P \ldots$$

for specifying syntax is called Backus-Naur form (BNF) for expressing grammars. For example, the first BNF-clause reads: a propositional formula can be
a variable, or
$\perp$, or
$P_1 \wedge P_2$, where P_1 and P_2 are propositional formulae, or
$P_1 \vee P_2$, where P_1 and P_2 are propositional formulae, or
$P_1 \to P_2$, where P_1 and P_2 are propositional formulae, or
$\neg P_1$, where P_1 is a propositional formula.

The symbol P is called a non-terminal, and when we apply the rules starting from P until we reach an expression without non-terminal we say that this expression is a production of P or it is in the language generated by P.

The BNF is a very common formalism for specifying syntax, e.g., of programming languages. See `http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html` or `http://en.wikipedia.org/wiki/Backus-Naur_form`.

The elements of L_P are called (propositional) formulas[23].

[20] The symbol $\perp$ stands for “false”.

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

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

[22] “Officially”, negation does not exist in our language and proof system. Negation is only used as a shorthand, or syntactic sugar (➜ p.37), 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.

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

We omit unnecessary brackets[24].

We will later (➜ p.67) learn about another syntactic category, that of terms.

I propositional logic, a formula may also be called a proposition.

[24] To save brackets, we use standard associativity and precedences. All binary connectives (➜ p.16) are right-associative:

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

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

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

Propositional Logic: Semantics

An assignment is a function $\mathcal{A} : V \to \{0, 1\}$. 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 ...

Propositional Logic: Semantics (2)

$$\begin{aligned}
\mathcal{A}(\bot) &= 0 \\
\mathcal{A}(\neg\phi) &= \begin{cases} 1 & \text{if } \mathcal{A}(\phi) = 0 \\ 0 & \text{otherwise} \end{cases} \\
\mathcal{A}(\phi \wedge \psi) &= \begin{cases} 1 & \text{if } \mathcal{A}(\phi) = 1 \text{ and } \mathcal{A}(\psi) = 1 \\ 0 & \text{otherwise} \end{cases} \\
\mathcal{A}(\phi \vee \psi) &= \begin{cases} 1 & \text{if } \mathcal{A}(\phi) = 1 \text{ or } \mathcal{A}(\psi) = 1 \\ 0 & \text{otherwise} \end{cases} \\
\mathcal{A}(\phi \to \psi) &= \begin{cases} 1 & \text{if } \mathcal{A}(\phi) = 0 \text{ or}^{25} \mathcal{A}(\psi) = 1 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

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

Propositional Logic: Semantics (3)

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

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

The semantics will be generalised later (➜ p.70).

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

$$\dfrac{\dfrac{A \to (B \to C) \quad A}{B \to C}\ \to\text{-}E \qquad B}{C}\ \to\text{-}E$$

where the leaves are called assumptions.

We write $A_1, ..., A_n \vdash A$ if there exists a derivation of A with assumptions $A_1, ..., A_n$, e.g. $A \to (B \to C), A, B \vdash C$[26].

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

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

We will say more about the $\vdash$ notation later (➜ p.47).

Natural Deduction: an Abstract Example[27]

- Language $\mathcal{L} = \{♥, ♣, ♠, ♦\}$.
- Deductive system given by rules of proof:

$$\frac{♦}{♣}\,\alpha \qquad \frac{♦}{♠}\,\beta \qquad \frac{♣ \quad ♠}{♥}\,\gamma \qquad \frac{\begin{matrix}[♦]\\ \vdots \\ ♥\end{matrix}}{♥}\,\delta$$

How do you read these rules?[28]

How about this one?[29]

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

[27] Natural deduction is not just about propositional logic! We explain here the general principles (➜ p.14) 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.

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

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

[29] The last rule reads: if at some root of a tree in the forest

Proof of ♥

The rules:

$$\frac{\diamondsuit}{\clubsuit}\,\alpha \qquad \frac{\diamondsuit}{\spadesuit}\,\beta \qquad \frac{\clubsuit \quad \spadesuit}{\heartsuit}\,\gamma \qquad \frac{\begin{matrix}[\diamondsuit]\\ \vdots \\ \heartsuit\end{matrix}}{\heartsuit}\,\delta$$

The proof:

$$\dfrac{\dfrac{\dfrac{[\diamondsuit]^1}{\clubsuit}\,\alpha \qquad \dfrac{[\diamondsuit]^1}{\spadesuit}\,\beta}{\heartsuit}\,\gamma}{\heartsuit}\,\delta^1$$

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 make[30] an assumption. The assumption is now open[31].

We apply α.

Similarly with β.

We apply γ.

We apply δ, discharging two occurrences of ♦. We mark the brackets and the rule with a label so that it is clear which assumption is discharged in which step. The derivation is now a proof: it has no open assumptions (➔ p.27) (all discharged).

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

When interpreting a derivation tree, we must always consider the open assumptions. We must say: under the assumptions . . . , we derived . . . .

It is thus unproblematic to “make” assumptions.

[31] For example, all assumptions in

$$\dfrac{\dfrac{A \to (B \to C) \quad A}{B \to C}\,\to\text{-}E \qquad B}{C}\,\to\text{-}E$$

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 introduce[32], others eliminate connectives.

[32] It is typical that the basic (➜ p.14) 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 (➜ p.14) rules, we will later see that there are also derived rules.

Rules of Propositional Logic (➜ p.14): Conjunction

- Rules of two kinds: introduce (➜ p.28) and eliminate (➜ p.28) connectives

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

- Rules are schematic[33].

- Why valid[34]? If all assumptions are true, then so is conclusion

$$\mathcal{A} \models A \wedge B \text{ (➜ p.22) iff } \mathcal{A} \models A \text{ and } \mathcal{A} \models B$$

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

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

We will see more about the use of metavariables later (➜ p.50).

[34] A rule is valid if for any assignment (➜ p.20) under which the assumptions of the formula are true, the conclusion is true as well.

This is consistent with the earlier intuitive explanation (➜ p.10) of validity of a formula. Details can be found in any textbook on logic [vD80].

Note that while the notation $\mathcal{A} \models \ldots$ will be used again

Example Derivation with Conjunction

The rules:

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

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

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

$$\cfrac{\cfrac{A \wedge (B \wedge C)}{A}\ \wedge\text{-}EL \qquad \cfrac{\cfrac{A \wedge (B \wedge C)}{B \wedge C}\ \wedge\text{-}ER}{C}\ \wedge\text{-}ER}{A \wedge C}\ \wedge\text{-}I$$

Can we prove anything with just these three rules?[35]

later (➜ p.75), 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 $\mathcal{A} \models \ldots$ will still mean the same thing.

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

None of the rules involves discharging an assumption.

We have said earlier (➜ p.24) that a proof is a derivation with no open assumptions.

Consequently, the answer is no. We cannot prove anything with just these three rules.

Rules of Propositional Logic: Implication

- Rules

$$\dfrac{\begin{matrix}[A]\\ \vdots \\ B\end{matrix}}{A \to B}\,\to\text{-}I \qquad \dfrac{A \to B \quad A}{B}\,\to\text{-}E$$

- $\to$-E is also called modus ponens.

- $\to$-I formalizes strategy:
  To derive $A \to B$, derive B under the additional assumption A.

A very Simple Proof

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

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

Do you find this strange?[36]

[36] When we make the assumption P, we obtain a forest (➜ p.25) 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 $\to$-I, similarly as in our abstract example (➜ p.25).

Examples with Conjunction and Implication

1. $A \to B \to A$[37]
2. $A \land (B \land C) \to A \land C$[38]

[37] The rule(s):

$$\dfrac{\begin{matrix}[A]\\ \vdots\\ B\end{matrix}}{A \to B}\ \to\text{-}I$$

The proof:

$$\dfrac{\dfrac{[A]^1}{B \to A}\ \to\text{-}I}{A \to B \to A}\ \to\text{-}I^1$$

[38] The rules:

$$\dfrac{A \quad B}{A \land B}\ \land\text{-}I$$

$$\dfrac{A \land B}{A}\ \land\text{-}EL$$

$$\dfrac{A \land B}{B}\ \land\text{-}ER$$

$$\dfrac{\begin{matrix}[A]\\ \vdots\\ B\end{matrix}}{A \to B}\ \to\text{-}I$$

The proof:

$$\dfrac{\dfrac{\dfrac{[A \land (B \land C)]^2}{A}\ \land\text{-}EL \qquad \dfrac{\dfrac{[A \land (B \land C)]^2}{B \land C}\ \land\text{-}ER}{C}\ \land\text{-}ER}{A \land C}\ \land\text{-}I}{(A \land (B \land C)) \to (A \land C)}\ \to\text{-}I^2$$

3. $(A \to B \to C) \to (A \to B) \to A \to C$[39]

Are these object or metavariables here?[40]

[39] The rules:

$$\dfrac{\begin{matrix}[A]\\ \vdots \\ B\end{matrix}}{A \to B}\to\text{-}I$$

$$\dfrac{A \to B \quad A}{B}\to\text{-}E$$

The proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[(A \to B \to C)]^3 \quad [A]^5}{B \to C}\to\text{-}E \qquad \dfrac{[(A \to B)]^4 \quad [A]^5}{B}\to\text{-}E}{C}\to\text{-}E}{A \to C}\to\text{-}I^5}{(A \to B) \to A \to C}\to\text{-}I^4}{(A \to B \to C) \to (A \to B) \to A \to C}\to\text{-}I^3$$

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

Disjunction

- Rules

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

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

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

$$\frac{\bot \quad (\text{➜ p.16})}{A} \; \bot\text{-}E$$

  No introduction rule![41]

- Negation: define (➜ p.16) $\neg A$ as syntactic sugar[42] for $A \rightarrow \bot$. Rules for $\neg$ just special

[41] The symbol $\bot$ 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 $\bot$ then? We will see this soon. The main role is linked to negation. We quote from [And02, p. 152]:

> $\bot$ plays the role of a contradiction in indirect proofs.

[42] For any formal language (programming language, logic, etc.), the term syntactic sugar refers to syntax that is provided for the sake of readability and brevity, but which does not affect the expressiveness of the language.

It is usually a good idea to consider the language without the syntactic sugar for any theoretical considerations about the language, since it makes the language simpler and the considerations less error-prone. However, the correspondence between the syntactic sugar and the basic syntax should be stated formally.

cases[43] of rules for →. Convenient to have

$$\frac{\neg A \quad A}{B}\,\neg\text{-}E\text{[44]} \qquad \text{derived by} \qquad \cfrac{\cfrac{\neg A \quad A}{\bot}\,\to\text{-}E}{B}\,\bot\text{-}E$$

[43] The rule

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

is simply an instance of →-E (➜ p.31) (since $\neg A$ is shorthand for $A \to \bot$).

Likewise, the rule

$$\cfrac{\begin{matrix}[A]\\ \vdots\\ \bot\end{matrix}}{\neg A}$$

is simply an instance of →-I (➜ p.31). Therefore, we will not introduce these as special rules. But there is a special rule ¬-E (➜ p.38).

[44] For negation, it is common to have a rule

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

We have seen how this rule can be derived. The concept of deriving rules will be explained more systematically

Intuitionistic versus Classical Logic

- Peirce's Law: $((A \to B) \to A) \to A$.
  Is this valid[45]? Provable[46]?

later (➜ p.44).

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

[45] Yes, simply check the truth table:

A	B	$((A \to B) \to A) \to A$
True	*True*	*True*
True	*False*	*True*
False	*True*	*True*
False	*False*	*True*

[46] In the proof system given so far (➜ p.40), 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 logic[47], obtained by adding

$$A \vee \neg A^{48} \text{ or } \frac{\begin{matrix}[\neg A]\\ \vdots \\ \bot\end{matrix}}{A}\, RAA_{49} \text{ or } \frac{\begin{matrix}[\neg A]\\ \vdots \\ A\end{matrix}}{A}\, classical_{50}.$$

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

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

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

We quote the first sentence from [Min00]:

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

The difference between intuitionistic and classical logic has been the topic of a fundamental discourse in the literature on logic [PM68] [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

Recall the story of Oedipus from Greek mythology:

- Iokaste is the mother of Oedipus.
- Iokaste and Oedipus are the parents of Polyneikes.
- Polyneikes is the father of Thersandros.
- Oedipus is a patricide.
- Thersandros is not a patricide.

in intuitionistic logic.

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

[49] The rule

$$\dfrac{\begin{array}{c}[\neg A]\\ \vdots \\ \bot\end{array}}{A}\,RAA$$

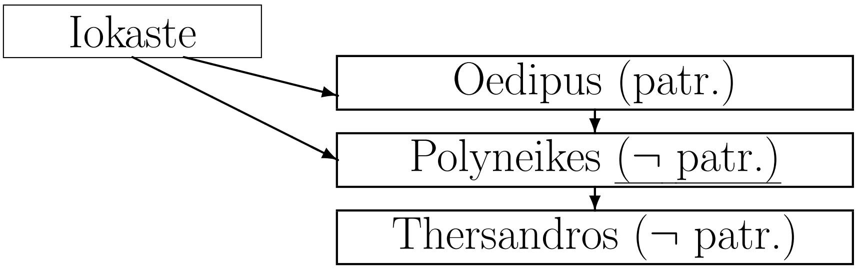
is called reductio ad absurdum.

[50] The rule

$$\dfrac{\begin{array}{c}[\neg A]\\ \vdots \\ A\end{array}}{A}\,\mathit{classical}$$

corresponds to the formulation in Isabelle.

Example of Classical Reasoning (cont.)



Iokaste

Oedipus (patr.)

Polyneikes (¬ patr.)

Thersandros (¬ patr.)

Does Iokaste have a child that is a patricide and that itself has a child that is not a patricide?

Case 1: If Polyneikes is a patricide, then Iokaste has a child (Polyneikes) that is a patricide and that itself has a child (Thersandros) that is not a patricide.

Case 2: If Polyneikes is not a patricide, then Iokaste has a child (Oedipus) that is a patricide and that itself has a child (Polyneikes) that is not a patricide.

Here[51] is another example.

[51] 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 if non-constructive (➜ p.40).

Overview of Rules

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

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

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

2.6 Deductive System: Derived Rules

Using the basic (➜ p.14) rules, we can derive new rules.

Example: Resolution rule.

$$\frac{R \vee S \quad \neg S}{R} \qquad\qquad \frac{R \vee S \quad [R]^1 \qquad \dfrac{\dfrac{\neg S \quad [S]^1}{\bot}\,{\to}\text{-}E}{R}\,\bot\text{-}E}{R}\,\vee\text{-}E^1$$

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 $\to$-E (➜ p.16).

To apply $\vee$-E in the end, we need to derive R. But that's easy using $\bot$-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 judgement[52], i.e., reason about $\Gamma \vdash A$ where $\Gamma \equiv A_1, \ldots, A_n$ instead of individual formulae.

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 (➜ p.??)). The sequent calculus is based on sequents, which are syntactic entities of the form $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$, where the $A_1, \ldots, A_n, B_1, \ldots, B_m$ 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

[52] An object like $A \rightarrow (B \rightarrow C), A, B \vdash C$ is called a derivability judgement. We explained it earlier (➜ p.24) as simply asserting the fact that there exists a derivation tree with C at its root and open assumptions $A \rightarrow (B \rightarrow C), A, B$.

Sequent Rules (for → /∧ Fragment)

Rules for assumptions[53] and weakening[54]:

$$\Gamma \vdash A^{55} \quad (\text{where } A \in \Gamma) \qquad \frac{\Gamma \vdash B}{A, \Gamma \vdash B}\,\textit{weaken}$$

Rules for ∧ and →:

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

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

---

call natural deduction using a sequent notation.

[53] The special rule for assumptions takes the role in this sequent style (➜ p.47) notation that the process of making and discharging assumptions had in natural deduction based on trees (➜ p.23).

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.

[54] The rule *weaken* is

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

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 (➜ p.**??**).

[55] An axiom is a rule without premises. We call a rule with premises proper.

More rules can be derived[56].

One can write an axiom A as

$$\frac{}{A}$$

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

Note that the natural deduction rules (➜ p.43) for propositional logic contain no axioms. In the sequent style (➜ p.47) 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.

Note also that even a proper rule in the object logic (➜ p.15) is just an axiom at the level of Isabelle's meta-logic (➜ p.7). This will be explained later (➜ p.247).

[56] 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

$$\dfrac{\dfrac{A \land (B \land C) \vdash A \land {\color{red}(\rlap{?X}B \land C)}}{A \land (B \land C) \vdash A}\;\land\text{-}EL \qquad \dfrac{\dfrac{A \land (B \land C) \vdash {\color{red}\rlap{?Z}A} \land ({\color{red}\rlap{?Y}B} \land C)}{A \land (B \land C) \vdash ({\color{red}\rlap{?Y}B} \land C)}\;\land\text{-}ER}{A \land (B \land C) \vdash C}\;\land\text{-}ER}{\dfrac{A \land (B \land C) \vdash A \land C}{\vdash A \land (B \land C) \to A \land C}\;\to\text{-}I}\;\land\text{-}I$$

This rule can be derived as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{A, B, \Gamma \vdash C}{A, \Gamma \vdash B \to C}\;\to\text{-}I}{\Gamma \vdash A \to B \to C}\;\to\text{-}I \qquad \dfrac{\Gamma \vdash A \land B}{\Gamma \vdash A}\;\land\text{-}EL}{\Gamma \vdash B \to C}\;\to\text{-}E \qquad \dfrac{\Gamma \vdash A \land B}{\Gamma \vdash B}\;\land\text{-}ER}{\Gamma \vdash C}\;\to\text{-}E$$

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

The topmost connective of the formula is $\to$, so the best rule[57] to choose is $\to$-*I*.

The topmost connective of the formula is $\wedge$, so the best rule (➜ p.51) to choose is $\wedge$-*I*.

Things are becoming less obvious. To know that $\wedge$-*EL* is the best rule for the l.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 r.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 $?X = (B \wedge C)$.

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

The question will be addressed more systematically later (➜ p.275).

Comments about Refinement

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

- Refinement style means we work from goals to axioms[58].
- Metavariables are used to delay commitments.

Isabelle allows other refinements[59]/alternatives too (see labs).

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

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

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

But that is just one aspect. We will give a more theoretic account of the way Isabelle constructs proofs later (➜ p.263).

3 Natural Deduction: Review

Overview

- Short review: ND Systems and proofs (➜ p.53)
- First-Order Logic (➜ p.60)
  - Overview (➜ p.60)
  - Syntax (➜ p.67)
  - Semantics (➜ p.70)
  - Deduction (➜ p.79), some derived rules, and examples

How Are ND Proofs Built?

ND proofs[60] build derivations under (possibly temporary) assumptions.

[60] ND stands for Natural Deduction. It was explained in the previous lecture (➜ p.23).

ND: Example for $\rightarrow / \wedge$ Fragment

Rules:

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

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

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

Proof:

$$\dfrac{\dfrac{\dfrac{[A \wedge B]^1}{B}\wedge\text{-}ER \qquad \dfrac{[A \wedge B]^1}{A}\wedge\text{-}EL}{B \wedge A}\wedge\text{-}I}{A \wedge B \rightarrow B \wedge A}\rightarrow\text{-}I^1$$

Alternative Formalization Using Sequents[61]

Rules (for → /∧ fragment). Here, Γ is a set of formulae.

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

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

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

Two representations (➜ p.56) equivalent. Sequent notation seems simpler in practice[62].

[61] The judgement ($\Gamma \vdash \phi$) means that we can derive ϕ from the assumptions in Γ using certain rules. As explained in the previous lecture (➜ p.47), one can make such judgements the central objects of the deductive system.

[62] In 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

$$\dfrac{\dfrac{\dfrac{A \wedge (B \wedge C) \vdash A \wedge ?X}{A \wedge (B \wedge C) \vdash A} \qquad \dfrac{\dfrac{A \wedge (B \wedge C) \vdash ?Z \wedge (?Y \wedge C)}{A \wedge (B \wedge C) \vdash (?Y \wedge C)}}{A \wedge (B \wedge C) \vdash C}}{A \wedge (B \wedge C) \vdash A \wedge C}}{\vdash A \wedge (B \wedge C) \to A \wedge C}$$

Solution for $?Z = A$, $?Y = B$ and $?X = (B \wedge C)$.
We went through this example in detail last lecture (➜ p.50).

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.52)
- Metavariables used to delay commitments

Isabelle allows other refinements (➜ p.52)/alternatives too (see labs).

4 First-Order Logic

4.1 First-Order Logic: Overview

In propositional logic, formulae are Boolean[63] combinations of propositions. This will remain important for modeling simple patterns of reasoning (➜ p.10).

An atomic (➜ p.10) proposition is just a letter (variable (➜ p.16)). 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.

[63] The set (or "type") *bool* contains the two truth values *True*, *False*. A propositional formula containing n variables can be viewed as a function $\mathit{bool}^n \to \mathit{bool}$. 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"[64]. 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].

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

In particular, no statement about all elements of a possibly infinite domain can be expressed in propositional logic, since each formula involves only finitely many different variables, and up to equivalence (➜ p.22) 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 (➜ p.16) 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 $\quad d(x, y) \equiv x$ is divisible by y

Propositional connectives (➜ p.16) 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"[65] in a domain[66].
- More generally, a function of arity (➜ p.67) n expresses an n-ary operation over some domain, e.g.

Function	arity	expresses ...
0	nullary	number "0"
s	unary	successor in $\mathbb{N}$[67]
$+$	binary	function plus in $\mathbb{N}$

The generic notation for function application is $f(t_1, \ldots, t_n)$, but note special notations[68]: infix, prefix, etc.

[65] As opposed to a variable which also stands for a "thing".

This distinction will become clear soon (➜ p.66).

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

Any set of "things" that we want to reason about.

[67] $\mathbb{N}$ denotes the natural numbers.

[68] So a function symbol f denotes an operation that takes n "things" and returns a "thing". $f(t_1, \ldots, t_n)$ is a "thing" that depends on "things" $t_1, \ldots, t_n$.

The generic notation for function application is like this: $f(t_1, \ldots, t_n)$, 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 "some[69] 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?[70]

$\forall x.\, \exists y.\, y * 2 = x$	true for rationals
$x < y \rightarrow \exists z.\, x < z \wedge z < y$	true for any dense (➜ p.115) order
$\exists x.\, x \neq 0$	true for domains with more than one element
$(\forall x.\, p(x, x)) \rightarrow p(a, a)$	valid

[69] Just like a constant, a variable stands for a "thing".

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

[70] Intuitively, satisfiable means "can be made true" and valid means "always true".

More formally, this will be defined later (➜ p.75).

4.2 First-Order Logic: Syntax

- Two syntactic categories: terms[71] and formulae
- A first-order language[72] 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 $i \in \mathbb{N}$ (➜ p.65).
- One often calls the pair $\langle \mathcal{F}, \mathcal{P} \rangle$ a signature.

[71] We have already learned about the syntactic category of formulae last lecture (➜ p.18).

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!

[72] There isn't simply the language of first-order logic! Rather, the definition of a first-order language is parametrised by giving a $\mathcal{F}$ and a $\mathcal{P}$. Each symbol in $\mathcal{F}$ and $\mathcal{P}$ must have an associated arity, i.e., the number of arguments the function or predicate takes. This could be formalized by saying that the elements of $\mathcal{F}$ are pairs of the form f/n, where f is the symbol itself and n is the arity, and likewise for $\mathcal{P}$. All that matters is that it is specified in some unambiguous way what the arity of each symbol is.

Terms and Formulae in First-Order Logic

Consider the following grammar (➜ p.16) ($x \in \mathit{Var}$, $f^n \in \mathcal{F}$, $p^n \in \mathcal{P}$):

$$T \ ::= \ x \ \mid \ f^n(\underbrace{T, \ldots, T}_{n \text{ times}^{73}})$$

$$F \ ::= \ \ldots \ (\text{➜ p.16}) \ \mid \ p^n(\underbrace{T, \ldots, T}_{n \text{ times}}) \ \mid \ \forall x. F \ \mid \ \exists x. F$$

The productions (➜ p.16) of T are called terms (set *Term*[74]).

The productions of F are called formulae (set *Form*).

Formulae of the form $p^n(\ldots)$ are called atoms.

Note quantifier scoping[75].

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.

[73] Note in particular the case $n = 0$. Then $1 \leq j \leq 0$ means that there exists no such j, and so $t_j \in \mathit{Term}$ for all j is vacuously true. We then speak of f as a constant (➜ p.65).

[74] *Term* and *Form* together make up a first-order language. Note that strictly speaking, *Term* and *Form* depend on the signature (➜ p.67), but we always assume that the signature is clear from the context.

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

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

Variable Occurrences

- All occurrences of a variable in a formula[76] are bound or free or binding.
- Example:

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

  Which are bound? Which are free? Which are binding?
- A formula with no free variable occurrences is called closed.
- There will be an exercise.

---

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 (➜ p.19) and associativity in propositional logic.

[76] All occurrences of a variable in a term or formula are bound or free or binding. These notions are defined by induction on the structure of terms/formulae. This is why the following definition is along the lines of our definition of terms (➜ p.68) and formulae (➜ p.68).

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 $f(t_1, \ldots, t_n)$ are the free occurrences of x in $t_1, \ldots, t_n$;

3. there are no free occurrences of x in $\bot$;
4. the free occurrences of x in $p(t_1, \ldots, t_n)$ are the free occurrences of x in $t_1, \ldots, t_n$;
5. the free occurrences of x in $\neg\phi$ are the free occurrences of x in ϕ;
6. the free occurrences of x in $\psi \circ \phi$ are the free occurrences of x in ψ and the free occurrences of x in ϕ ($\circ \in \{\wedge, \vee, \rightarrow\}$);
7. the free occurrences of x in $\forall y.\, \psi$, where $y \neq x$, are the free occurrences of x in ψ; likewise for $\exists$;
8. x has no free occurrences in $\forall x.\, \psi$; in $\forall x.\, \psi$, the (outermost) $\forall$ binds all free occurrences of x in ψ; the occurrence of x next to $\forall$ is a binding occurrence of x; likewise for $\exists$.

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

A structure[77] is a pair $\mathcal{A} = \langle U_{\mathcal{A}}, I_{\mathcal{A}} \rangle$ where $U_{\mathcal{A}}$ is an 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 $x \in \mathit{Var}$.

We also define

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

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

A universe is sometimes also called domain (➜ p.65).

As you saw, a structure (➜ p.71) gives a meaning to functions, predicates, and variables.

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

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

As shorthand, write $p^{\mathcal{A}}$[78] for $I_{\mathcal{A}}(p^n)$, etc.

---

the predicate symbols;

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

As before (➜ p.68), we assume that the signature (➜ p.67) 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].

[78] In the notation $p^{\mathcal{A}}$, the superscript has nothing to do with the superscript we sometimes use (➜ p.67) to indicate the arity.

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

The Value of Formulae

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

$$\begin{aligned}
\mathcal{A}(p(t_1,\dots,t_n)) &= \begin{cases} 1 & \text{if } (\mathcal{A}(t_1),\dots,\mathcal{A}(t_n)) \in p^{\mathcal{A}} \\ 0 & \text{otherwise} \end{cases} \\
\mathcal{A}(\forall x.\,\phi) &= \begin{cases} 1 & \text{if for all } u \in U_{\mathcal{A}}, \mathcal{A}_{[x/u]}{}^{79}(\phi) = 1 \\ 0 & \text{otherwise} \end{cases} \\
\mathcal{A}(\exists x.\,\phi) &= \begin{cases} 1 & \text{if for some } u \in U_{\mathcal{A}}, \mathcal{A}_{[x/u]}(\phi) = 1 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

Rest as for propositional logic (➜ p.21).

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

Models

- If $\mathcal{A}(\phi) = 1$, we write $\mathcal{A} \models \phi$ and say ϕ is true in $\mathcal{A}$ or $\mathcal{A}$ is a model of ϕ.
- If every suitable structure[80] 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.[81]

[80] A structure (➜ p.71) is suitable for ϕ if it defines meanings for the signature (➜ p.67) 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 symbol must be interpreted as an n-ary function. Without explicitly mentioning it (➜ p.72), we always assume that structures are suitable.

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

As explained before (➜ p.71), 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 $\mathcal{A}(.)$ for the meaning of a term (➜ p.73) or formula (➜ p.74). In the alternative terminology, we write $\mathcal{I}(A)(.)$ instead. This makes sense since

An Example

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

We now show a model and a non-model ...

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 model[82]: Not a model[84]:

$$\begin{array}{rcl@{\qquad}rcl}
U_{\mathcal{A}} & = & \mathbb{N}\ (\text{➜ p.65}) & U_{\mathcal{A}} & = & \{\mathtt{a},\mathtt{b},\mathtt{c}\} \\
p^{\mathcal{A}} & = & \{(m,n) \mid m <\text{[83]} n\} & p^{\mathcal{A}} & = & \{(\mathtt{a},\mathtt{b}),(\mathtt{a},\mathtt{c})\} \\
s^{\mathcal{A}}(x) & = & x+1\ (\text{➜ p.77}) & s^{\mathcal{A}} & = & \text{“the identity function”}
\end{array}$$

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

[83] In logic, we insist on the distinction between syntax and semantics. In particular, we set up the formalism so that the syntax is fixed first (➜ p.67) and then the semantics (➜ p.70), and so there could be different semantics for the same syntax.

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

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

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

4.4 Towards a Deductive System

In natural language, quantifiers are often implicit[85]: all males don't cry.

Some phrases in natural language proofs have the flavor of introduction rules (➜ p.28).

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 (➜ p.86).

Existential statements are proven by giving a witness.

we write $m < n$, we assume that $m, n \in \mathbb{N}$ (➜ p.75), in this context.

[84] The identity function maps every object to itself.

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

[85] In the statement

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

the ∀-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 (➜ p.28) are “inherited”[86].

But we must introduce rules for the quantifiers.

[86] First-order logic inherits all the rules of propositional logic (➜ p.28). Note however that the metavariables (➜ p.29) in the rules now range over first-order formulae.

Universal Quantification (∀): Rules

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

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

Note that rules are schematic[87]: $P(x)$ stands for any formula, and $P(t)$ stands for the formula obtained by substituting t for x (➜ p.83).

[87] Similarly as in the previous lecture (➜ p.29), one should note that P is not a predicate, but rather $P(x)$ is a schematic expression: $P(x)$ stands for any formula, possibly containing occurrences of x.

In the context of $\forall$-E, $P(t)$ stands for the formula obtained from $P(x)$ by replacing all occurrences of x by t (➜ p.83).

Universal Quantification: Side Condition

What does arbitrary mean? Consider the following “proof”

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[x=0]^1}{\forall x.\, x=0}\,\forall\text{-}I}{x=0\to\forall x.\,x=0}\,\to\text{-}I^1}{\forall x.\,(x=0\to\forall x.\,x=0)}\,\forall\text{-}I}{0=0\to\forall x.\,x=0}\,\forall\text{-}E \qquad \dfrac{}{0=0}\,\mathit{refl}^{88}}{\forall x.\,x=0}\,\to\text{-}E$$

Formal meaning of side condition (➜ p.80): x not free in any open assumption on which $P(x)$ depends. Violated![89]

[88] When one has a predicate symbol $=$, it is usual to have a rule that says that $=$ is reflexive (➜ p.104).

Don't worry about it at this stage, just take it that we have such a rule. We will look at this later (➜ p.100).

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

Note that saying “x must not be free in any open assumption on which $P(x)$ depends” means in particular that $P(x)$ itself must not be an assumption. This is the case we have here!

So whenever $\forall$-I (➜ p.80), the $P(x)$ above the line will be the root of a derivation tree constructed so far, and this tree cannot be the trivial tree just consisting of the assumption $P(x)$.

Another Proof? (1)

Is the following a proof? Is the conclusion valid?

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

Conclusion is not valid.
The formula is false when $U_{\mathcal{A}}$ has at least 2 elements.[90]

Proof is incorrect.

[90] Here we assume that the predicate symbol $=$ is interpreted by $\mathcal{A}$ (➜ p.71) 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 $\mathcal{A}_{[x/u]}$ (➜ p.74)$(\neg\forall y.\, x = y) = 1$ 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, $\mathcal{A}((\forall x.\, \neg\forall y.\, x = y) \to \neg\forall y.\, y = y) = 0$.

Reason: Substitution[91] must avoid capturing[92] variables.

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

1. $x[x \leftarrow t] = t$;
2. $y[x \leftarrow t] = y$ if y is a variable other than x;
3. $f(t_1, \ldots, t_n)[x \leftarrow t] = f(t_1[x \leftarrow t], \ldots, t_n[x \leftarrow t])$ (where f is a function symbol, $n \geq 0$);
4. $p(t_1, \ldots, t_n)[x \leftarrow t] = p(t_1[x \leftarrow t], \ldots, t_n[x \leftarrow t])$ (where p is a predicate symbol, possibly $\bot$);
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. $(\mathsf{Q}x.\psi)[x \leftarrow t] = \mathsf{Q}x.\psi$ (where $\mathsf{Q} \in \{\forall, \exists\}$);

Replacing x with y in $\forall$-E is illegal because y is bound (➜ p.69) in $\neg\forall y.\, y = y$. This detail concerns substitution (and renaming of bound (➜ p.69) variables), not $\forall$-E. Exercise

---

8. $(\mathsf{Q}y.\psi)[x \leftarrow t] = \mathsf{Q}y.(\psi[x \leftarrow t])$ (where $\mathsf{Q} \in \{\forall, \exists\}$) if $y \neq x$ and $y \notin FV(t)$;

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

[92] A substitution (➜ p.83) (replacement of a variable by a term) must not replace bound (➜ p.69) occurrences of variables, and if we replace x with t in an expression ϕ, then this replacement should not turn free (➜ p.69) occurrences of variables in t into bound (➜ p.69) occurrences in ϕ. It is possible to avoid this by renaming variables.

This is part of the standard definition of a substitution (➜ p.83). The problem is not related to $\forall$-E in particular.

Another Proof? (2)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\forall x.\, A(x) \wedge B(x)]^1}{A(x) \wedge B(x)}\,\forall\text{-}E}{A(x)}\,\wedge\text{-}EL}{\forall x.\, A(x)}\,\forall\text{-}I \qquad \dfrac{\dfrac{\dfrac{[\forall x.\, A(x) \wedge B(x)]^1}{A(x) \wedge B(x)}\,\forall\text{-}E}{B(x)}\,\wedge\text{-}ER}{\forall x.\, B(x)}\,\forall\text{-}I}{(\forall x.\, A(x)) \wedge (\forall x.\, B(x))}\,\wedge\text{-}I}{(\forall x.\, A(x) \wedge B(x)) \to (\forall x.\, A(x)) \wedge (\forall x.\, B(x))}\,\to\text{-}I^1$$

Yes (check side conditions[93] of $\forall$-I).

[93] In both cases, x does not occur free (➜ p.69) in $\forall x.\, A(x) \wedge B(x)$, which is the open assumption (➜ p.81) on which $A(x)$, respectively $B(x)$, depends.

Boys Don't Cry

Let $\phi \equiv (\forall x.\, b(x) \to m(x)) \land (\forall x.\, m(x) \to \neg c(x))$.

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{
          \dfrac{[\phi]^1}{\forall x.\, m(x) \to \neg c(x)}\,\land\text{-}ER
        }{m(x) \to \neg c(x)}\,\forall\text{-}E
        \qquad
        \dfrac{
          \dfrac{
            \dfrac{[\phi]^1}{\forall x.\, b(x) \to m(x)}\,\land\text{-}EL
          }{b(x) \to m(x)}\,\forall\text{-}E
          \qquad [b(x)]^2
        }{m(x)}\,\to\text{-}E
      }{\neg c(x)}\,\to\text{-}E
    }{b(x) \to \neg c(x)}\,\to\text{-}I^2
  }{\forall x.\, b(x) \to \neg c(x)}\,\forall\text{-}I
}{\phi \to (\forall x.\, b(x) \to \neg c(x))}\,\to\text{-}I^1
$$

Aside: $A \leftrightarrow B$

Define[94] $A \leftrightarrow B$ as $A \rightarrow B \wedge B \rightarrow A$.

The following rule can be derived (➜ p.44) (in propositional logic, actually):

$$\dfrac{\begin{matrix}[A] & [B]\\ \vdots & \vdots \\ B & A\end{matrix}}{A \leftrightarrow B}\,\leftrightarrow\text{-}I$$

You could do this as an exercise!

[94] By defining we mean, use $A \leftrightarrow B$ as shorthand for $A \rightarrow B \wedge B \rightarrow A$, in the same way as we regard negation as a shorthand (➜ p.16).

Proof?

$$\cfrac{\cfrac{[A]^1}{\forall x.\, A}\,\forall\text{-}I \qquad \cfrac{[\forall x.\, A]^1}{A}\,\forall\text{-}E}{A \leftrightarrow \forall x.\, A}\,\leftrightarrow\text{-}I^1$$

Yes, but only if x not free (➜ p.69) in A.

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

Side Conditions and Proof Boxes

We mentioned previously (➜ p.46) 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 define[95] $\exists x.\, A$ as $\neg\forall x.\, \neg A$.
- Equivalence follows from our definition of semantics (➜ p.74).

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

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

[95] By <u>defining</u> 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 (➜ p.16).

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

Where do the Rules for $\exists$ Come from?

- We can[96] use definition $\exists x.\, A \equiv \neg\forall x.\, \neg A$ and the given rules for $\forall$ to derive (➜ p.44) ND (➜ p.55) 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 formalization follows the second approach.

[96]

- We can use definition $\exists x.\, A \equiv \neg\forall x.\, \neg A$ and the given rules for $\forall$ to derive (➜ p.44) ND (➜ p.55) proof rules. In this case, the soundness (➜ p.15) 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 (➜ p.15) must be proven by hand (however, proving rules sound is an aspect we neglect (➜ p.9) 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 (➜ p.15).

$\exists$-I as a Derived Rule

The rule:

$$\dfrac{P(t)}{\exists x.\, P(x)}\;\exists\text{-}I$$

$$\dfrac{\dfrac{\dfrac{[\forall x.\,\neg P(x)]^1}{\neg P(t)}\;\forall\text{-}E \qquad P(t)}{\bot}\;\rightarrow\text{-}E}{\exists x.\, P(x)\neg\forall x.\,\neg P(x)}\;\rightarrow\text{-}I^1$$

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

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

We aim for applying $\rightarrow$-I in the last step (recall $\neg$-definition (➜ p.16)

We apply $\forall$-E.

Making assumption $P(t)$ allows us to use $\rightarrow$-E (recall $\neg$-definition (➜ p.16)).

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

$\exists$-E as a Derived Rule

The rule:

$$\dfrac{\exists x.\,P(x) \qquad \begin{array}{c}[P(x)]\\ \vdots \\ R\end{array}}{R}\ \exists\text{-}E$$

$$\dfrac{\exists x.\,P(x)\neg\forall x.\,\neg P(x) \qquad \dfrac{\dfrac{\dfrac{[\neg R]^1 \qquad \begin{array}{c}[P(x)]^2\\ \vdots\\ R\end{array}}{\bot}\ \to\text{-}E}{\neg P(x)}\ \to\text{-}I^2}{\forall x.\,\neg P(x)}\ \forall\text{-}I}{\dfrac{\bot}{R}\ RAA^1}\ \to\text{-}E$$

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

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

We assume a hypothetical derivation[97].

We make an additional assumption and apply $\to$-E (recall $\neg$-definition (➜ p.16))

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

[97] We are constructing here a "schematic fragment" of a derivation tree. Within this construction, we assume a hypothetical derivation of R from assumption $P(x)$. 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 R from assumption $P(x)$. Somewhere in the middle of the constructed fragment, we will discharge the assumption $P(x)$. In the final rule $\exists$-E, this means an application of $\exists$-E involves discharging $P(x)$. Therefore $\exists$-E has brackets around the $P(x)$.

- The hypothetical derivation of R may contain other assumptions than $P(x)$. These are not discharged in the constructed fragment, and so in the final rule $\exists$-E, we

At this step, the side condition from $\forall$-I applies. $\exists$-E will inherit it![98]

We apply $\to$-E.

We are done. Note that this proof uses classical[99] reasoning.

---

must also read the notation

$$\begin{array}{c} P(x) \\ \vdots \\ R \end{array}$$

as a derivation of R where one of the assumptions is $P(x)$. There may be other assumptions, but these are not discharged. This is no different from previous rules (➜ p.25) involving discharging.

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

x must not be free (➜ p.69) in R or in hypotheses of the subderivation of R other than $P(x)$ (occurrences in $(P(x)$ are allowed (➜ p.95) because the assumption $P(x)$ was discharged before the application of $\forall$-I). Contrast this with $\forall$-I (➜ p.81).

[99] Defining (➜ p.90) $\exists x.\, A$ as $\neg\forall x.\, \neg A$ is only sensible in

Example Derivation Using $\exists$-E

We want to prove $(\forall x.\,A(x) \to B) \to ((\exists x.\ A(x)) \to B)$, where x does not occur free in B (➜ p.88).

$$\dfrac{\dfrac{\dfrac{[\exists x.\ A(x)]^2 \qquad \dfrac{\dfrac{[\forall x.\,A(x) \to B]^1}{A(x) \to B}\,\forall\text{-}E \qquad [A(x)]^3}{B}\,{\to}\text{-}E}{B}\,\exists\text{-}E^3}{(\exists x.\ A(x)) \to B}\,{\to}\text{-}I^2}{(\forall x.\,A(x) \to B) \to ((\exists x.\ A(x)) \to B)}\,{\to}\text{-}I^1$$

classical reasoning (➜ p.40), since the derivation of the rule $\exists$-E requires the RAA (➜ p.40) rule.

4.6 Conclusion on FOL

- Propositional logic is good for modeling simple patterns of reasoning (➜ p.10) like "if . . . then . . . else".
- In first-order logic, one has "things" and relations on / properties of "things". Quantify over "things". Powerful[100]!

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

For example, one can reason:

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

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

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

While some applications in the fields mentioned above require other logics, e.g. modal logics[101], 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-

- Limitation: cannot quantify over predicates[102].
- "A" world or "the" world is modeled in first-order logic using so-called first-order theories. This will be studied next lecture (➜ p.112).

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.

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

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

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

since we would be quantifying over p.

5 First-Order Logic with Equality

Overview

Last lecture: first-order logic (➜ p.60).

This lecture:

- first-order logic with equality (➜ p.100) and first-order theories (➜ p.112);
- set-theoretic reasoning (➜ p.124).

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

FOL with Equality

Equality is a logical symbol rather than a mathematical one[103].

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

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

Recall (➜ p.67) 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 (➜ p.67), and whose meaning must be defined “by the user” by giving a structure (➜ p.71).

Above we say “mathematical” instead of “non-logical” because we assume that mathematics is our domain of discourse, so that the signature (➜ p.67) 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.

$t_1 = t_2 \in \mathit{Form}$ (➜ p.68) if $t_1, t_2 \in \mathit{Term}$ (➜ p.68).

Semantics: recall a structure (➜ p.71) 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 "="[104] 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.

[104]

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

The first $\underline{=}$ is a predicate symbol.

The second $\underline{=}$ 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 $\underline{=}$ is semantic equality, i.e., the identity relation on the domain (➜ p.70).

Rules[105]

- Equality is an equivalence relation[106]

$$\frac{}{t=t}\,\mathit{refl}\quad \frac{s=t}{t=s}\,\mathit{sym}\quad \frac{r=s\quad s=t}{r=t}\,\mathit{trans}$$

- Equality is also a congruence[107] on terms and all rela-

[105] Since $=$ is a logical symbol in the formalism of first-order logic with equality, there should be derivation rules (➜ p.79) for $=$ to derive which formulas $a=b$ are true.

[106] In general mathematical terminology, a relation $\equiv$ is an equivalence relation if the following three properties hold:

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

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

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

Example: being equal modulo 6.
"a is equal b modulo 6" is often written $a \equiv b \bmod 6$.

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

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

$14 \equiv 8 \bmod 6$ and $15 \equiv 9 \bmod 6$, hence $14 \cdot 15 \equiv 8 \cdot 9 \bmod 6$.

tions[108]

$$\frac{r = s}{T(r) = T(s)}\; cong_1$$

$$\frac{r = s \quad P(r)}{P(s)}\; cong_2$$

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.

[108] Why did we use letters T and P here?

Recall the rules for building terms (➜ p.68) and atoms (➜ p.68).

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

In general, no! The notations $T(r)$ and $P(r)$ are metanotations (➜ p.29). $T(r)$ stands for any term in which r occurs, and $P(r)$ stands for any formula in which r occurs.

Soundness of Rules

For any $U_{\mathcal{A}}$, equality in $U_{\mathcal{A}}$ is an equivalence relation[109] and functions/predicates/logical-operators are "truth-functional"[110].

And in this context, the notation $T(s)$ stands for the term obtained from $T(r)$ by replacing all occurrences of r with s. In analogy the notation $P(s)$ is defined.

Note that r and s are 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 (➜ p.216).

[109] On the semantic level, two things are equal if they are identical. Semantic equality is an equivalence relation (➜ p.104). This semantic fact is so fundamental that we cannot explain it any further.

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

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

Congruence: Alternative Formulation

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

$$\frac{r = s}{T[z \leftarrow r] = T[z \leftarrow s]}\ \mathit{cong}_1$$

$$\frac{r = s \quad P[z \leftarrow r]}{P[z \leftarrow s]}\ \mathit{cong}_2$$

One time z is replaced with r and one time with s.[111]

$I_{\mathcal{A}}(T(x)) = I_{\mathcal{A}}(T(y))$.

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.

[111] The notation $T[z \leftarrow r]$ stands for the term obtained from T by replacing z with r. $[z \leftarrow r]$ is called a substitution (➜ p.83).

To have an unambiguous notation for "replacing some occurrences of r", we start from a term T containing occurrences of a variable z. On the LHS, z is replaced with r, on the RHS z is replaced with s. So on the RHS we have a term obtained from the term on the LHS by replacing some occurrences of r with s.

One can say that z is introduced to mark the occurrences of r that should be replaced by s.

Note that r and s can be arbitrary terms, whereas z is a variable (substitutions replace variables, not arbitrary

Congruence: Example

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

$$\begin{aligned} &A = x^2 + w^2 > 12 \cdot x, \quad A = z^2 + w^2 > 12 \cdot x, \\ &A = x^2 + w^2 > 12 \cdot z, \quad A = z^2 + w^2 > 12 \cdot z. \end{aligned}$$ [112]

We show two ways:

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

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

terms).

[112] The atom $x^2 + w^2 > 12 \cdot x$ contains two occurrences of x. There are four ways to choose some occurrences of x in $x^2 + w^2 > 12 \cdot x$.

Each of those ways corresponds to an atom obtained from $x^2 + w^2 > 12 \cdot x$ by replacing some occurrences of x with z. That is, there are four different A's such that $A[x/z] = x^2 + w^2 > 12 \cdot x$. 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 3 for z.

Generalized Congruence

The congruence rules can be generalized to n equalities instead of just 1 equality. The generalized rules are derivable from the simple ones by n-fold application.

$$\frac{r_1 = s_1 \;\; \cdots \;\; r_n = s_n}{T[z_1 \leftarrow r_1, \ldots, z_n \leftarrow r_n] = T[z_1 \leftarrow s_1, \ldots, z_n \leftarrow s_n]}\;cong_1$$

$$\frac{r_1 = s_1 \;\; \cdots \;\; r_n = s_n \quad P[z_1 \leftarrow r_1, \ldots, z_n \leftarrow r_n]}{P[z_1 \leftarrow s_1, \ldots, z_n \leftarrow s_n]}\;cong_2$$

Isabelle Rule

The Isabelle FOL rule is simply[113] (using a tree syntax)

$$\frac{r = s \quad P(r)}{P(s)}\, subst$$

or literally

$$[\![a = b; P(a)]\!] \Longrightarrow P(b)$$

[113] The Isabelle FOL rule is:

$$\frac{r = s \quad P(r)}{P(s)}\, subst$$

In this rule, P is an Isabelle metavariable (➜ p.29).

Why doesn't the Isabelle rule contain a z to mark (➜ p.107) which occurrences should be replaced?

We cannot understand this yet (➜ p.216), but think of P as a formula where some positions are marked in such a way that once we apply P to r (we write $P(r)$), r will be inserted into all those positions. This is why $P(r)$ is a formula and $P(s)$ is a formula obtained by replacing some occurrences of r with s.

Proving $\exists x.\, t = x$

$$\dfrac{\dfrac{}{t = t}\ \textit{refl}\ (\text{➜ p.104})}{\exists x.\, t = x}\ \exists\text{-}I\ (\text{➜ p.92})$$

In the rule $\dfrac{P(t)}{\exists x.\, P(x)}\ \exists\text{-}I\ (\text{➜ p.92})$, "$P(x)$" is metanotation (➜ p.29). In the example, $P(x) = (t = x)$.

Notational confusion avoided by a precise metalanguage (➜ p.196).

6 First-Order Theories

What Is a Theory?

Recall our intuitive explanation of theories (➜ p.7).

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 (➜ p.14).

A structure (➜ p.71) in which these rules are true is then called a model (➜ p.75) of the theory.

6.1 Example 1: Partial Orders

- The language of the theory of partial orders[114]: $\leq$[115]

[114] A partial order is a binary relation that is reflexive, transitive (➜ p.104), and anti-symmetric: $a \leq b$ and $b \leq a$ implies $a = b$.

[115] $\leq$ 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 (➜ p.48)

$$\forall x, y, z.\, x \leq y \land y \leq z \to x \leq z^{116}$$
$$\forall x, y.\, x \leq y \land y \leq x \leftrightarrow x = y^{117}$$

- Alternative to axioms is to use rules
$\dfrac{x \leq y \quad y \leq z}{x \leq z}\,\mathit{trans}$ $\dfrac{x \leq y \quad y \leq x}{x = y}\,\mathit{antisym}$ $\dfrac{x = y}{x \leq y}\,{\leq}\text{-}\mathit{refl}$

  Such a conversion is possible since implication is the main connective.[118]

[116] The axiom $\forall x, y, z.\, x \leq y \land y \leq z \to x \leq z$ encodes transitivity.

[117] Note that $\forall x, y.\, x \leq y \land y \leq x \leftrightarrow x = y$ encodes both antisymmetry ($\to$) and reflexivity ($\leftarrow$). Recall (➜ p.87) that $A \leftrightarrow B$ is a shorthand for $A \to B \land B \to A$.

[118] One can see that using $\to$-I and $\to$-E (➜ p.31), one can always convert a proof using the axioms to one using the proper (➜ p.48) rules.

More generally, an axiom of the form $\forall x_1, \ldots, x_n.\ A_1 \land \ldots \land A_n \to B$ can be converted to a rule

$$\frac{A_1 \quad \ldots A_n}{B}\ .$$

Do it in Isabelle!

More on Orders

- A partial order (➜ p.113) $\leq$ is a linear or total order[119] when

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

  Note: no "pure" rule formulation[120] of this disjunction.

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

  $$\forall x, y.\, x < \text{[121]} y \to \exists z.(x < z \wedge z < y)$$

  What does $<$ mean?

[119] We define these notions according to usual mathematical terminology.

A partial order (➜ p.113) $\leq$ is a linear or total order if for all a, b, either $a \leq b$ or $b \leq a$.

A partial order (➜ p.113) $\leq$ is dense if for all a, b where $a < b$, there exists a c such that $a < c$ and $c < b$.

[120] The axiom $\forall x, y.\, x \leq y \vee y \leq x$ cannot be phrased as a proper (➜ p.48) rule in the style of, for example, the transitivity axiom (➜ p.114).

[121] We use $s < t$ as shorthand for $s \leq t \wedge \neg s = t$.

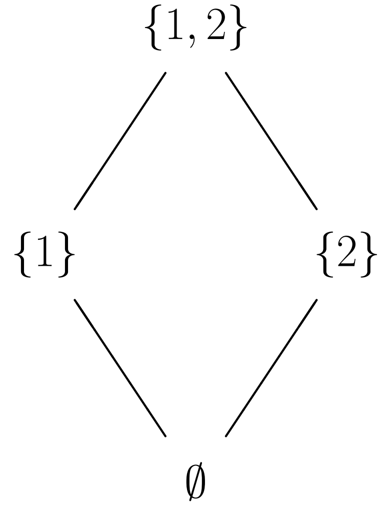
We say that $<$ is the strict part of the partial order (➜ p.113) $\leq$.

Structures for Orders ...

Give structures (➜ p.71) for orders that are ...

1. not total: $\subseteq$-relation[122];
2. total but not dense: integers with $\leq$;
3. dense: reals with $\leq$.

[122] The $\subseteq$-relation is partial but not total. As an example, consider the $\subseteq$-relation on the set of subsets of $\{1, 2\}$.



Depicting partial orders (➜ p.113) by such a graph is quite common. Here, node a is below node b and connected by an arc if and only if $a <$ (➜ p.115)b and there exists no c with $a < c < b$.

6.2 Example 2: Groups

- Language: Function symbols $\_ \cdot \_$, $\_^{-1}$, e[123]

In this example, we have the partial order (➜ p.113)

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

[123] $\underline{\_ \cdot \_}$ 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).

$\underline{\_^{-1}}$ 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) (➜ p.65).

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 is[124] a model[125] of

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

  It is an example of an equational theory[126].

Two theorems: (1) $x^{-1} \cdot x = e$ and (2) $e \cdot x = x$
We will now prove them.

[124] In general mathematical terminology, a group consists of three function symbols $\_ \cdot \_$, $\_^{-1}$, e, obeying the following laws:

Associativity $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all a, b, c,

Right neutral $a \cdot e = a$ for all a,

Right inverse $a \cdot a^{-1} = e$ for all a.

[125] A model (➔ p.75) of the group axioms is a structure (➔ p.71) 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".

[126] An equational theory is a set of equations. Each equation is an axiom.

Equational Proofs

A typical proof in an equational theory looks very different from the natural deduction style (➜ p.23), but it looks very much like the proofs you know from school mathematics.

An equational proof consists simply of a sequence of equations, written as $t_1 = t_2 = \ldots = t_n$, 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.

More on the justification later (➜ p.122).

Sometimes, each equation is surrounded by several ∀-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 (➜ p.48) rules where the premises are called conditions [Höl90].

Note also that sometimes, one also considers the basic rules of equality (➜ p.104) 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 1

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

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

$x^{-1} \cdot x = x^{-1} \cdot (x \cdot e) = x^{-1} \cdot (x \cdot (x^{-1} \cdot x^{-1^{-1}})) =$
$x^{-1} \cdot ((x \cdot x^{-1}) \cdot x^{-1^{-1}}) = x^{-1} \cdot (e \cdot x^{-1^{-1}}) =$
$(x^{-1} \cdot e) \cdot x^{-1^{-1}} = x^{-1} \cdot x^{-1^{-1}} = e.$

Theorem 2

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

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

$$e \cdot x = (x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x) \overset{\text{Thm. 1}}{=} x \cdot e = x$$

Equational Proofs Justified

Translated to natural deduction style (➜ p.23), an equational proof looks like this:

$$\dfrac{\dfrac{\dfrac{\mathsf{Ax}_{n-1}}{\cdots}\;\forall\text{-}E}{s_{n-1}=s'_{n-1}}\;(\mathit{sym}) \qquad \begin{array}{c}\dfrac{\dfrac{\dfrac{\mathsf{Ax}_2}{\cdots}\;\forall\text{-}E}{s_2 = s'_2}\;(\mathit{sym}) \quad \dfrac{\dfrac{\dfrac{\mathsf{Ax}_1}{\cdots}\;\forall\text{-}E}{s_1=s'_1}\;(\mathit{sym}) \quad \dfrac{}{t_1=t_1}\;\mathit{refl}}{t_1=t_2}\;\mathit{cong}_2}{}\;\mathit{cong}_2 \\ \vdots \\ t_1 = t_{n-1}\end{array}}{t_1=t_n}\;\mathit{cong}_2\ (\text{➜ p.104})$$

where each Ax_i is an axiom of the equational theory[127].

[127] The double line marked with $\forall$-E stands for 0 or more applications of the $\forall$-E (➜ p.80) rule. Moreover, there might be an application of *sym* (➜ p.104).

Lessons Learned from this Example

- Equational proofs are often tricky! Equalities are used in different directions, “eureka”[128] terms are needed, etc.
- In some cases (the word problem[129] is) decidable.
- In Isabelle, equational proofs are accomplished by term rewriting (➜ p.299).
- Explicit natural deduction (➜ p.23) proofs are tedious in practice. Try it on above examples![130]

[128] By “eureka” terms we mean terms that have to be guessed in order to find a proof. At least at first sight, it seems like these terms simply fall from the sky.

The Greek ευρεκα (heureka) is 1st person singular perfect of ευρισκειν (heuriskein), “to find”. It was exclaimed by Archimedes upon discovering how to test the purity of Hiero’s crown.

[129] The word problem w.r.t. an equational theory (here: the group axioms) is the problem of deciding whether two terms s and t are equal in the theory, that is to say, whether the formula $s = t$ is true in any model (➜ p.75) of the theory.

[130]

$$\dfrac{\dfrac{\boxed{\text{r-neutr}}}{x\cdot e=x} \quad \dfrac{\dfrac{\text{Theorem 1}}{x^{-1}\cdot x=e} \quad \dfrac{\dfrac{\boxed{\text{assoc}}}{(x\cdot x^{-1})\cdot x=x\cdot(x^{-1}\cdot x)} \quad \dfrac{\dfrac{\dfrac{\boxed{\text{r-inv}}}{x\cdot x^{-1}=e}}{e=x\cdot x^{-1}}\,\mathit{sym} \quad \dfrac{}{e\cdot x=e\cdot x}\,\mathit{refl}}{e\cdot x=(x\cdot x^{-1})\cdot x}}{e\cdot x=x\cdot(x^{-1}\cdot x)}}{e\cdot x=x\cdot e}}{e\cdot x=x}$$

This is an example of the general scheme (➜ p.122).

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].

- 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 (➜ p.320), we will be completely formal.

Most steps use the congruence rule $cong_2$ (➜ p.104).

Each framed box in the derivation tree stands for a sub-tree consisting of a group axiom (➜ p.117) and possibly several applications of $\forall$-E (➜ p.80).

Sets: Language

Assuming any first-order language with equality (➜ p.127), we add:

- set-comprehension $\{x \mid P(x)\}$[131] and a binary membership predicate $\in$.
- Term/formula distinction inadequate[132]: need a syntactic category for sets.
- Comprehension is a binding operator: x bound in $\{x \mid P(x)\}$ (➜ p.69).

[131] Set comprehension is a way of defining sets. $\{x \mid P(x)\}$ stands for the set of elements of the universe for which $P(x)$ (some formula usually containing x) holds.

[132] It is more adequate to regard a set as a term than as a formula. A set is a "thing", not a statement about "things". (➜ p.67)

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. (➜ p.68)

However, the syntax used in set comprehensions is not legal syntax for terms (➜ p.68), since $P(x)$ is a formula.

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

Examples

- What does the following say?

$$x \in \{y \mid y \bmod 6 = 0\}$$

  $x \bmod 6 = 0$.

- What about this?

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

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

Proof Rules for Sets

Introduction, elimination, extensional equality[133]

$$\frac{P(t)}{t \in \{x \mid P(x)\}}\,\textit{compr-I} \qquad \frac{t \in \{x \mid P(x)\}}{P(t)}\,\textit{compr-E}$$

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

The following equivalence is derivable[134]:

$\forall x.\, P(x) \leftrightarrow$ (➜ p.87)$x \in \{y \mid P(y)\}$

[133] Two things are extensionally equal if they are "equal in their effects". Thus two sets are equal if they have the same members, regardless of what syntactic expressions are used to define those sets.

Note that extensional equality may be undecidable.

[134]

$$\cfrac{\cfrac{\cfrac{[P(x)]^1}{x \in \{y \mid P(y)\}}\,\textit{compr-I} \qquad \cfrac{[x \in \{y \mid P(y)\}]^1}{P(x)}\,\textit{compr-E}}{P(x) \leftrightarrow x \in \{y \mid P(y)\}}\,\leftrightarrow\textit{-I}\ (\text{➜ p.87})^1}{\forall x.\, P(x) \leftrightarrow x \in \{y \mid P(y)\}}\,\forall\textit{-I}$$

Rule ∀-*I* (➜ p.80) was defined in a previous lecture.

Digression: Sorts

- The following notations are common in mathematics and logic:

$$\begin{aligned}
\{x \underline{\in U} \mid P(x)\} &\equiv \{x \mid x \in U \wedge P(x)\}\\
\forall x \underline{\in U}.\, P(x) &\equiv \forall x.\, x \in U \to P(x)\\
\exists x \underline{\in U}.\, P(x) &\equiv \exists x.\, x \in U \wedge P(x)
\end{aligned}$$

  These are syntactic sugar (➜ p.37). One uses them when U denotes an "important" sub-universe[135] such as $\mathbb{R}$ or $\mathbb{N}$. Such a U is sometimes called sort.

- There is also sorted first-order logic[136].

[135] We already know what a universe (➜ p.71) or domain (➜ p.71) is. To interpret a particular language, we have a structure (➜ p.71) 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 $\mathbb{N}$.

[136] In sorted logic, sorts are part of the syntax. So the signature (➜ p.67) 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

7.2 Operations on Sets

- Functions on sets

$$\begin{aligned} A \cap {}^{137} B &\equiv \{x \mid x \in A \wedge x \in B\} \\ A \cup B &\equiv \{x \mid x \in A \vee x \in B\} \\ A \setminus B &\equiv \{x \mid x \in A \wedge x \notin B\} \end{aligned}$$

- Predicates on sets

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

well-formed, and so they are not assigned a meaning.

In contrast, our logic is unsorted. The special syntax we provide for sorted reasoning is just syntactic sugar (➜ p.37), 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.

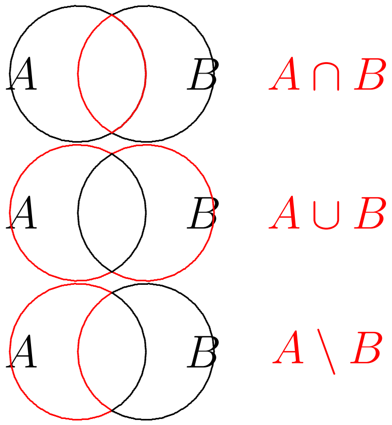
[137]

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



Correspondence between Set-Theoretic and Logical Operators

$$\begin{aligned} x \in A \cap B &\leftrightarrow x \in A \wedge x \in B \\ x \in A \cup B &\leftrightarrow x \in A \vee x \in B \\ x \in A \setminus B &\leftrightarrow x \in A \wedge x \notin B \end{aligned}$$

These correspondences follow from the definitions of the set-theoretic operators (➜ p.129) and $\forall x.\, P(x) \leftrightarrow x \in \{y \mid P(y)\}$ (➜ p.127).

Example: what is the logical form[138] of $x \in ((A \cap B) \cup (A \cap C))$? $(x \in A \wedge x \in B) \vee (x \in A \wedge x \in C)$

[138] When we transform an expression containing set operators $\cap, \cup, \setminus, \subseteq$ into an expression using $\wedge, \vee, \neg, \rightarrow$, we call the latter the logical form of the expression.

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

Venn diagram (Is this a proof?)[139]

[139] A Venn diagram draws sets as bubbles. Intersecting sets are drawn as overlapping bubbles, and the overlapping area is meant to depict the intersection of the sets.

A Venn diagram is not a proof in the sense defined earlier (➜ p.14).

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 distinction (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 language[140])

By extensionality (➜ p.127), 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

$$\begin{aligned}&(x \in A \wedge (x \in B \vee x \in C)) \leftrightarrow \\ &\qquad (x \in A \wedge x \in B) \vee (x \in A \wedge x \in C)\end{aligned}$$

But that is a propositional tautology.

Do it in Isabelle!

[140] We intersperse formal notation with natural language here in order to give an intuitive and short proof.

We can also do this more formally in Isabelle.

7.3 Extending Set Comprehensions

Recall set comprehensions (➜ p.125) $\{x \mid P(x)\}$.
Now what do you think this is?

$$\{f(x) \mid P(x)\} \equiv \{y \mid \exists x.\, P(x) \land y = f(x)\}$$

Example: $t \in \{x^2 \mid x > 5\}$ equivalent to (➜ p.127) $\exists x.\, x > 5 \land t = x^2$.
True for $t \in \{36, 49, \ldots\}$

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.

$$\begin{aligned} x \in \{m_s \mid s \in S\} &\leftrightarrow (\text{➜ p.134})\ x \in \{y \mid \exists s.\, s \in S \wedge y = m_s\} \\ &\leftrightarrow (\text{➜ p.127})\ \exists s.\, s \in S \wedge x = m_s \\ &\leftrightarrow (\text{➜ p.128})\ \exists s \in S.\, x = m_s \end{aligned}$$

Uses extended comprehensions (➜ p.134), indexing syntax, and sorted quantification (➜ p.128).

Logical Forms of the New Notation

What is the logical form (➜ p.131) of $\{x_i \mid i \in I\} \subseteq A$?

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

[141]

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

follows from the definition of $\subseteq$ (➜ p.131).

[142] We want to show

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

$$\begin{array}{lcr} x \in \{x_i \mid i \in I\} & \equiv & \text{(def. of notation) (➜ p.134)} \\ x \in \{y \mid \exists i.\, i \in I \wedge y = x_i\} & \equiv & \textit{compr-I} \text{ (➜ p.127)} \\ \exists i.\, i \in I \wedge x = x_i & \equiv & \text{(Sorted quantification) (➜ p.128)} \\ \exists i \in I.\, x = x_i & & \end{array}$$

Powersets

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

What is the logical form (➜ p.131) of:

1. $x \in \wp(A)$?

   $x \subseteq A$, i.e., $\forall y.\, (y \in x \to y \in A)$

2. $\wp(A) \subseteq \wp(B)$?

   $\forall x.\, x \in \wp(A) \to x \in \wp(B)$, i.e.,
   $\forall x.\, x \subseteq A \to x \subseteq B$, i.e.,
   $\forall x.\, (\forall y.\, y \in x \to y \in A) \to (\forall y.\, y \in x \to y \in B)$

Exercise: prove that the last answer is equivalent to $A \subseteq B$, i.e., $\forall x.\, x \in A \to x \in B$.

7.4 Outlook

Sets can have other sets as elements.

Implicitly assume that universe of discourse is collection[143] of all sets.

[143] We speak of collection of all sets rather than set of all sets in order to pretend that we are being careful since we are not sure if there is such a thing as a set of all sets. Therefore we use the "neutral" word collection whose meaning is obvious...

Is it?

Recall that we have defined set as collection of objects (➜ p.124) 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 $U := \{x \mid \top$[144]$\}$. Then[145] $U \in U$.

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

Assume $R \in R$. By the definition of R, this means $R \in \{A \mid A \notin A\}$. Using *compr-E* (➜ p.127), this implies $R \notin R$.

Now assume $R \notin R$. Using *compr-I* (➜ p.127), 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?[146]

[144] Assume that $\top$ is syntactic sugar (➜ p.37) for a proposition that is always true, say $\top \equiv \bot \to \bot$. We have not introduced this, but it is convenient.

So semantically (➜ p.74), we have $I_{\mathcal{A}}(\top) = 1$ for all $I_{\mathcal{A}}$.

[145] Recall that a set comprehension (➜ p.125) 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 (➜ p.127), this means that $x \in U$ for any x. Thus in particular, $U \in U$.

[146] It tells us that there can be no such thing as the set of all sets.

The fundamental flaw of naïve set theory is in saying that a set is a collection of "objects" (➜ p.124) 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

Where Do We Go from here?

- The λ-calculus (➜ p.141) as basis for a metalanguage (➜ p.196) to avoid notational confusion (➜ p.111)
- Resolution (➜ p.263) and other deduction techniques (➜ p.274): understanding Isabelle better and achieving a higher level of automation
- Higher-order logic (➜ p.320): a formalism for (among other things) non-naïve set theory[147]

---

existence of sets, so the definition is circular.

Note that while the proof of the contradiction looks classical (it seems that we make the assumption $R \in R \vee R \notin R$, it is in fact not classical. There will be an exercise on this.

The intuition for the solution to this dilemma is not difficult: A set is a collection of objects of which we are already sure that they exist. In particular, since we are only just about to define sets, these objects may not themselves be sets.

Once we have such sets, we can introduce "sets of second order", that is, sets that contain sets of the first kind. This process can be continued ad infinitum.

The formal details will come later (➜ p.320).

[147] Higher-order logic (➜ p.320) is a solution to the dilemma posed by Russell's paradox. (➜ p.139)

It is a surprisingly simple formalism which can be extended (➜ p.398) conservatively: this means that it can be ensured that the extensions cannot compromise the truth or falsity of statements that were already expressible before the extension.

The λ-Calculus: 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 (➜ p.162) ($\lambda^{\rightarrow}$)
- An extension of the typed λ-calculus (➜ p.178)
- Higher-order unification (➜ p.192)

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 8[148].

[148] As you might guess, the formalism of the λ-calculus is not directly related to usual arithmetic and so it is not built into this formalism that $3+5$ should evaluate to 8. However, it may be a reasonable choice, depending on the context, to extend the λ-calculus in this way, but this is not our concern at the moment.

Syntax

($x \in Var$, $c \in Const$[149])

$$e \ ::= \ x \ | \ c \ | \ (ee) \ | \ (\lambda x.\, e)$$[150]

The objects generated by this grammar (➜ p.16) are called λ-terms or simply terms.

[149] Similarly as for first-order logic (➜ p.67), 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 (➜ p.65). 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 (➜ p.144) 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.

[150] A λ-term can either be

Conventions: iterated λ & left-associated application[151]

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

Is $\lambda x.\,x + 5$ a λ-term?[152]

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

[151] We write $\lambda x_1 x_2 \dots x_n.e$ instead of $\lambda x_1.(\lambda x_2.(\dots e)\dots)$.

$e_1\ \ e_2 \dots e_n$ is equivalent to $(\dots(e_1\ \ e_2)\dots e_n)\dots$, not $(e_1(e_2 \dots e_n)\dots)$. Note that this is in contrast to the associativity of logical operators (➜ p.19). There are some good reasons for these conventions.

[152] Strictly speaking, $\lambda x.\,x+5$ does not adhere to the definition of syntax of λ-terms, at least if we parse it in the usual way: $+$ is an infix constant applied to arguments x and 5.

If we parse $x+5$ as $((x+)5)$, i.e., x applied to (the constant) $+$, and the resulting term applied to (the constant) 5, then $\lambda x.\,x + 5$ would indeed adhere to the definition of syntax of

Substitution

- Will see shortly that "computations" are based on substitutions, defined similarly as in FOL (➜ p.83).

$$(g\,x\,3)[x \leftarrow 5]^{153} = g\,5\,3$$

- Must respect free (➜ p.148) and bound (➜ p.148) variables,

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

- Same problems as with quantifiers (➜ p.83)

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

λ-terms, but of course, this is pathological and not intended here.

It is convenient to allow for extensions of the syntax of λ-terms, allowing for:

- application to several arguments rather than just one;
- infix notation (➜ p.65).

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 $\lambda x.\, ((\mathit{plus}\, x)\, 5)$ (i.e., write + in a Curryed (➜ p.156) way).

[153] Here we use the notation $e[x \leftarrow t]$ for the term obtained from e by replacing x with t. There is also the notation $e[t/x]$, and confusingly, also $e[x/t]$. We will attempt to be consistent within this course, but be aware that you may find such different notations in the literature.

Bound, Free, Binding Occurrences

Recall the notions of bound, free, and binding (➜ p.69) occurrences of variables in a term. Same here:

λ-calculus	FOL
$FV(x) := \{x\}$	$= FV(x)$
$FV(c) := \emptyset$	$= FV(c)$
$FV(MN) := FV(M) \cup FV(N)$	$= FV(M \wedge N)$
$FV(\lambda x.\, M) := FV(M) \setminus \{x\}$	$= FV(\forall x.\, M)$

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

A term with no free variable occurrences is called closed (➜ p.69).

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 $y \in FV(N)$, and z is fresh (➜ p.192): $z \notin FV(N) \cup FV(P)$

Cases similar to those for quantifiers: λ binding is ‘generic’[154].

[154] Recall the definition (➜ p.83) 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 (➜ p.125). 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 (➜ p.222).

Substitution: Example

$$\begin{aligned}(x(\lambda x.\, xy))[x \leftarrow \lambda z.\, z] &\overset{3}{=} x[x \leftarrow \lambda z.\, z](\lambda x.\, xy)[x \leftarrow \lambda z.\, z]\\ &\overset{1,4}{=} (\lambda z.\, z)\lambda x.\, xy\end{aligned}$$

$$\begin{aligned}(\lambda x.\, xy)[y \leftarrow x] &\overset{6}{=} \lambda z.\, ((xy)[x \leftarrow z][y \leftarrow x])\\ &\overset{3,1,2}{=} \lambda z.\, ((zy)[y \leftarrow x])\\ &\overset{3,2,1}{=} \lambda z.\, zx\end{aligned}$$

In the last example, clause 6 avoids capture, i.e., $\lambda x.\, xx$[155].

[155] If it wasn't for clause 6, i.e., if we applied clause 5 ignoring the requirement on freeness, then $(\lambda x.\, xy)[y \leftarrow x]$ would be $\lambda x.\, xx$.

Reduction: Intuition

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

$f\ x = x + 5$ (➜ p.143) $\rightsquigarrow$ $f = \lambda x.\, x + 5$

$f\ 3 = 3 + 5$ $\rightsquigarrow$

$(\lambda x.\, x + 5)(3) \rightarrow_\beta (x + 5)[x \leftarrow 3] = 3 + 5$ (➜ p.144)

β-reduction replaces (➜ p.144) a parameter by an argument[156].

This should propagate into contexts[157], e.g.

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

[156] In the λ-term $(\lambda x.M)N$, we say that N is an argument (and the function $\lambda x.M$ is applied to this argument), and every occurrence of x in M is a parameter (we say this because x is bound by the λ).

This terminology may be familiar to you if you have experience in functional programming, but actually, it is also used in the context of function and procedure declarations in imperative programming.

[157] In

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

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

Reduction: Definition

- Axiom for β-reduction: $(\lambda x.M)N \rightarrow_\beta M[x \leftarrow N]$[158]
- Rules for β-reduction of redexes[159] in contexts:

$$\frac{M \rightarrow_\beta M'}{NM \rightarrow_\beta NM'} \qquad \frac{M \rightarrow_\beta M'}{MN \rightarrow_\beta M'N} \qquad \frac{M \rightarrow_\beta M'}{\lambda z.M \rightarrow_\beta \lambda z.M'} *^{160}$$

- Reduction is reflexive-transitive (➜ p.104) closure

$$\frac{M \rightarrow_\beta N}{M \rightarrow_\beta^* N} \qquad \frac{}{M \rightarrow_\beta^* M} \qquad \frac{M \rightarrow_\beta^* N \quad N \rightarrow_\beta^* P}{M \rightarrow_\beta^* P}$$

- A term without redexes is in β-normal form.

[158] As you see, β-reduction is defined using rules (two of them being axioms (➜ p.48), the rest proper rules (➜ p.48)) in the same way that we have defined proof systems for logic (➜ p.14) before. Note that we wrote the first axiom (➜ p.48) defining β-reduction without a horizontal bar.

[159] In a λ-term, a subterm of the form $(\lambda x.\, M)N$ is called a redex. It is a subterm to which β-reduction can be applied.

[160] The rule for propagating $\rightarrow_\beta$ to an abstraction, let us call it λ-*abstr*,

$$\frac{M \rightarrow_\beta M'}{\lambda z.M \rightarrow_\beta \lambda z.M'} \; \lambda\text{-}abstr$$

actually has a vacuous side condition:

> z is not free in any open assumption on which $M \rightarrow_\beta M'$ depends.

The side condition is just like for ∀ (➜ p.80).

The side condition is vacuous because in the derivation system for $\rightarrow_\beta$ (or $\rightarrow_\beta^*$) we present here, there is no rule

Reduction: Examples

$$\underline{(\lambda x.\, \lambda y.\, g\, x\, y)a}\, b \rightarrow_\beta \underline{(\lambda y.\, g\, a\, y)b} \rightarrow_\beta g\, a\, b$$

$$\text{So } (\lambda x.\, \lambda y.\, g\, x\, y)a\, b \rightarrow_\beta^* g\, a\, b$$

---

involving discharging open assumptions, and thus there is no point in <u>making</u> assumptions. The root of a derivation tree for $\rightarrow_\beta$ is always an application of the axiom for β-reduction. When we consider $\rightarrow_\beta^*$, we may in addition have applications of the reflexivity axiom.

However, we will have exercises on $\rightarrow_\beta$ using an Isabelle theory called `RED`, and in this theory, the above rule is called `epsi` and looks as follows:

```
"[|!!x. M(x) --> N(x)|] ==>
(lam x. M(x)) --> (lam x. N(x))"
```

Observe that there is a meta-level universal quantifier in this rule. From the exercises, you know that the meta-level universal quantifier corresponds to a side condition in paper-and-pencil proofs.

Moreover, when we later look at the meta-logic (➜ p.228), there will be a rule (➜ p.**??**)

$$\frac{a \equiv b}{(\lambda x.a) \equiv (\lambda x.b)} \equiv\text{-}abstr^*$$

looking very similar to the λ-*abstr* rule and having a side condition.

To illustrate why the side condition is needed in general, consider a derivation system where in addition to the rules for $\rightarrow_\beta$ and $\rightarrow_\beta^*$, we also allow applications rules for $\rightarrow$ (➔ p.31) (implication) and $\forall$ (➔ p.80) of first-order logic.

For the example we give, suppose that we have an encoding of the number 0 and the + function in the untyped λ-calculus, and that these behave as expected (in fact we will have an exercise showing this; in the following we use "0" and "+" just for simplicity and clarity; + is written infix).

Under these assumptions, we will now derive $\lambda xy.\,y{+}x \rightarrow_\beta \lambda xy.\,y$. Before looking at the derivation tree, think about what this says intuitively: it says that + is a function that takes two arguments, ignores the first argument and returns the second argument. Clearly, this does not correspond to the usual definition of +! The trick in the following derivation is to smuggle in an instantiation of x, namely to force

Shows Currying[161]

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

---

x to be 0. The derivation looks as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[y+x \to_\beta y]^1}{\lambda y.\, y+x \to_\beta \lambda y.\, y}\;\lambda\text{-}abstr}{\lambda xy.\, y+x \to_\beta \lambda xy.\, y}\;\lambda\text{-}abstr}{(y+x \to_\beta y) \to \lambda xy.\, y+x \to_\beta \lambda xy.\, y}\;{\to}\text{-}I^1}{\forall x.(y+x \to_\beta y) \to \lambda xy.\, y+x \to_\beta \lambda xy.\, y}\;\forall\text{-}I}{(y+0 \to_\beta y) \to \lambda xy.\, y+x \to_\beta \lambda xy.\, y}\;\forall\text{-}E \qquad \dfrac{\text{(routine)}}{y+0 \to_\beta y}}{\lambda xy.\, y+x \to_\beta \lambda xy.\, y}\;{\to}\text{-}E$$

In the above derivation, the side condition for λ-*abstr* is violated.

In Isabelle, such a “smuggling in” of an instantiation can be achieved using `instantiate_tac`, see RED_wrongepsi.thy and wrongepsi.ML.

[161] You may be familiar with functions taking several arguments, or equivalently, a tuple of arguments, rather than just one argument.

Shows divergence[162]

But (➜ p.157) $\underline{(\lambda x.\, \lambda y.\, y)((\lambda x.\, xx)(\lambda x.\, xx))} \to_\beta \lambda y.\, y$

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 (➜ p.146), so $g\,a\,b$ is read $(g\,a)\,b$.

Currying will become even clearer once we introduce the typed λ-calculus (➜ p.163).

[162] We say that a β-reduction sequence diverges if it is infinite.

Note that for $(\lambda xy.\, y)((\lambda x.\, xx)(\lambda x.\, xx))$, there is a finite β-reduction sequence

$$(\lambda xy.\, y)((\lambda x.\, xx)(\lambda x.\, xx)) \to_\beta \lambda y.\, y$$

but there is also a diverging sequence

$$(\lambda xy.\, y)((\lambda x.\, xx)(\lambda x.\, xx)) \to_\beta (\lambda xy.\, y)((\lambda x.\, xx)(\lambda x.\, xx)) \to_\beta \ldots$$

Conversion

- $\underline{\beta\text{-conversion}}$: "symmetric closure" (➜ p.104) of β-reduction

$$\frac{M \to_\beta^* N}{M =_\beta N} \qquad \frac{M =_\beta N}{N =_\beta M}$$

- $\underline{\alpha\text{-conversion}}$: bound variable renaming (usually implicit[163])

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

- $\underline{\eta\text{-conversion}}$: for normal-form analysis[164]

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

---

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

$$\lambda z.\, z =_\beta \lambda x.\, x$$

[164] η-conversion is defined as

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

It is needed for reasoning about normal forms.

$$g\,x =_\eta \lambda y.\, g\,x\,y \quad \underline{\text{reflects}} \quad g\,x\,b =_\beta (\lambda y.\, g\,x\,y)b$$

More specifically: if we did not have the η-conversion rule, then $g\,x$ and $\lambda y.\, g\,x\,y$ would not be "equivalent" up to conversion. But that seems unreasonable, because they behave the same way when applied to b. Applied to b, both terms can be converted to $g\,x\,b$. This is why it is reasonable to introduce a rule such that $g\,x$ and $\lambda y.\, g\,x\,y$ are "equivalent" up to conversion.

λ-Calculus Meta-Properties[165]

Confluence (equivalently[166], 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 $N_2 \rightarrow_\beta^* P$.

---

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 $N_1 \overset{*}{\leftrightarrow} N_2$, 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 $\overset{*}{\leftrightarrow} := (\leftrightarrow)^*$ 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 $M_1 \overset{*}{\leftrightarrow} M_9$.

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

One also says that the η-conversion expresses the idea of extensionality (➜ p.127) [HS90, chapter 7].

Note that with the help of β-reduction and

$$\begin{tikzcd}
 & M \arrow[dl, "*"'] \arrow[dr, "*"] & \\
N_1 \arrow[dr, "*"'] & & N_2 \arrow[dl, "*"] \\
 & P &
\end{tikzcd}$$

Uniqueness of Normal Forms

Corollary of the Church-Rosser property:

If $M \rightarrow_\beta^* N_1$ and $M \rightarrow_\beta^* N_2$ where N_1 and N_2 in normal form, then $N_1 =_\alpha N_2$.

Example:

$$(\lambda xy.\, y)(\underline{(\lambda x.\, xx)a}) \rightarrow_\beta \underline{(\lambda xy.\, y)(aa)} \rightarrow_\beta \lambda y.\, y$$

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

---

transitivity (➜ p.153), η-conversion can be generalized to more than one variable, i.e. $M =_{\beta\eta} \lambda x_1 \ldots x_n.\, M\, x_1 \ldots x_n$. E.g. we can derive $\lambda xyz.\, M\, x\, y\, z =_{\beta\eta} M$:

$$\frac{\dfrac{\dfrac{\dfrac{\lambda z.\, M\, x\, y\, z =_\eta M\, x\, y}{\lambda yz.\, M\, x\, y\, z =_{\beta\eta} \lambda y.\, M\, x\, y} \qquad \lambda y.\, M\, x\, y =_\eta M\, x}{\lambda yz.\, M\, x\, y\, z =_{\beta\eta} M\, x}}{\lambda xyz.\, M\, x\, y\, z =_{\beta\eta} \lambda x.\, M\, x} \qquad \lambda x.\, M\, x =_\eta M}{\lambda xyz.\, M\, x\, y\, z =_{\beta\eta} M}$$

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

165

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

166 A reduction $\rightarrow$ is called confluent if

for all M, N_1, N_2, if $M \rightarrow^* N_1$ and $M \rightarrow^* N_2$, then

Turing Completeness

The λ-calculus can represent all computable functions.[167]

[167] The untyped λ-calculus is Turing complete. This is usually shown not by mimicking a Turing machine in the λ-calculus, but rather by exploiting the fact that the Turing computable functions are the same class as the μ-recursive functions [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 $\lambda^{\to}$

Motivation: Suppose you have constants 1, 2 with usual meaning. Is it sensible to write 1 2 (1 applied to 2)?

$\lambda^{\to}$ (simply typed λ-calculus, simple type theory) restricts syntax to "meaningful expressions".

In untyped λ-calculus, we have syntactic objects[168] called terms (➜ p.145).

We now introduce syntactic objects called types[169].

We will say "a term has a type" or "a term is of a type".

[168] We also say that we have defined a term language (➜ p.145). A particular language is given by a signature, although for the untyped λ-calculus this is simply the set of constants $Const$.

[169] We can say that we define a type language, i.e., a language consisting of types. A particular type language is characterized by 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 $\{\mathbb{N}, \mathit{bool}\}$, where $\mathbb{N}$ represents the natural numbers and *bool* the Boolean values $\perp$ (➜ p.16) 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 (➜ p.162), $T \in \mathcal{B}$)

  $$\tau ::= T \mid (\tau \to \tau) \text{ (➜ p.16)}$$

  Examples: $\mathbb{N}$, $\mathbb{N} \to$ [170]$\mathbb{N}$, $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$, $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$[171]

- Syntax for (raw[172]) terms: λ-calculus (➜ p.145) augmented with types[173]

  $$e \; ::= \text{ (➜ p.16) } x \mid c \mid (ee) \mid (\lambda x^{\tau}.\, e)$$

[170] The type $\mathbb{N} \to \mathbb{N}$ is the type of a function that takes a natural number and returns a natural number.

The type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ 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.

[171] To save parentheses, we use the following convention: types associate to the right, so $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ stands for $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$.

Recall that application associates to the left (➜ p.146). This may seem confusing at first, but actually, it turns out that the two conventions concerning associativity fit together very neatly.

[172] In the context of typed versions of the λ-calculus, raw terms are terms built ignoring any typing conditions (➜ p.168). So raw terms are simply terms as defined for the untyped λ-calculus (➜ p.145), possibly augmented with type superscripts.

[173] So far, this is just syntax!

$(x \in \mathit{Var},\, c \in \mathit{Const}^{174})$

The notation $(\lambda x^{\tau}.\, e)$ simply specifies that binding (➜ p.148) 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.

[174] *Var* and *Const* are the sets of variables and constants, respectively, as for the untyped λ-calculus (➜ p.145).

Signatures and Contexts

Generally (in various logic-related formalisms[175]) a signature defines the "fixed" symbols of a language, and a context defines the "variable" symbols of a language. In $\lambda^{\rightarrow}$,

[175] For propositional logic (➜ p.16), 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 (➜ p.16).

In first-order logic (➜ p.67), 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 (➜ p.128), the signature must also specify the sorts of all symbols. But we did not study sorted logic in any detail.

In the untyped λ-calculus, the signature is simply the set of constants (➜ p.145).

Summarizing, we have not been very precise about the

- a signature Σ is a sequence ($c \in \mathit{Const}$ (➜ p.164))

$$\Sigma ::= \langle\,\rangle \mid \Sigma, c : \tau^{176}$$

- a context Γ is a sequence ($x \in \mathit{Var}$)

$$\Gamma ::= \langle\,\rangle \mid \Gamma, x : \tau$$

notion of a signature so far.

For $\lambda^{\rightarrow}$, the rules for "legal" terms become more tricky, and it is important to be formal about signatures.

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

[176] We call an expression of the form $x : \tau$ (➜ p.166) or $c : \tau$ (➜ p.166) 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.

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

$$\Gamma \vdash_\Sigma c\, x : \sigma$$[177]

where $\Sigma = c : \tau \to \sigma$ and $\Gamma = x : \tau$.

We usually leave Σ implicit (➜ p.166) and write $\vdash$ instead of $\vdash_\Sigma$.

If Γ is empty it is omitted.

[177] The expression

$$\Gamma \vdash_\Sigma c\, x : \sigma$$

is called a type judgement. It says that given the signature $\Sigma = c : \tau \to \sigma$ and the context $\Gamma = x : \tau$, 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 (➜ p.47) before.

Type Assignment Calculus: Rules[178]

$$\frac{c:\tau\in\text{[179]}\Sigma}{\Gamma\vdash c:\tau}\,\mathit{assum}\qquad \Gamma,x:\tau,\Delta\vdash x:\tau\quad \mathit{hyp}\text{[180]}$$

$$\frac{\Gamma\vdash e:\sigma\to\tau\quad \Gamma\vdash e':\sigma}{\Gamma\vdash ee':\tau}\,\mathit{app}\qquad \frac{\Gamma,x:\sigma\text{[181]}\vdash e:\tau}{\Gamma\vdash\lambda x^{\sigma}.\,e\;:\sigma\to\tau}\,\mathrm{abs}$$

Note that due to requiring $x:\sigma$ to occur at the end, rule

[178] Type assignment is defined as a system of rules for deriving type judgements (➜ p.167), in the same way that we have defined derivability judgements (➜ p.47) for logics (➜ p.14), and β-reduction (➜ p.153) for the untyped λ-calculus.

[179] Recall that Σ is a sequence. By abuse of notation, we sometimes identify this sequence with a set and allow ourselves to write $c:\tau\in\Sigma$.

We may also write $\Sigma\subseteq\Sigma'$ meaning that $c:\tau\in\Sigma$ implies $c:\tau\in\Sigma'$.

[180] One could also formulate *hyp* as follows:

$$\frac{x:\tau\in\Gamma}{\Gamma\vdash x:\tau}\,\mathit{hyp}$$

That would be in close analogy to LF, a system not treated here.

[181] A sequence is a collection of objects which differs from sets in that a sequence contains the objects in a certain order, and there can be multiple occurrences of an object.

abs is deterministic[182] when applied bottom-up.

We write a sequence containing the objects $o_1, \ldots, o_n$ as $\langle o_1, \ldots, o_n \rangle$, or sometimes simply $o_1, \ldots, o_n$.

If Ω is the sequence $o_1, \ldots, o_n$, then we write Ω, o for the sequence $\langle o_1, \ldots, o_n, o \rangle$ and o, Ω for the sequence $\langle o, o_1, \ldots, o_n \rangle$.

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

[182] Signatures (➜ p.165) and contexts are sequences, and intuitively, the order in which the type bindings (➜ p.166) occur in these sequences does not matter.

Now, the way we have set up the type assignment calculus, it would seem that the order does matter, namely since in rule abs, the binding $x : \sigma$ above the horizontal line must be the last binding in the context. An alternative formulation would be

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

However, the original formulation is more straightforward in light of the fact that type derivations are usually constructed

Also note the analogy to minimal logic over →[183].

bottom-up. The bottom-up application of the original abs is deterministic, whereas the alternative formulation would confront us with the choice of how to split up the context.

For example, we could start a derivation of $y : \rho, z : \omega \vdash \lambda x^\sigma .\, c \, : \sigma \to \tau$ in three ways:

$$\frac{\underline{x : \sigma}, y : \rho, z : \omega \vdash c : \tau}{y : \rho, z : \omega \vdash \lambda x^\sigma .\, c \, : \sigma \to \tau}\,\text{abs}$$

or

$$\frac{y : \rho, \underline{x : \sigma}, z : \omega \vdash c : \tau}{y : \rho, z : \omega \vdash \lambda x^\sigma .\, c \, : \sigma \to \tau}\,\text{abs}$$

or

$$\frac{y : \rho, z : \omega, \underline{x : \sigma} \vdash c : \tau}{y : \rho, z : \omega \vdash \lambda x^\sigma .\, c \, : \sigma \to \tau}\,\text{abs}$$

[183] Recall the sequent rules (➜ p.48) of the “→ /∧” fragment of propositional logic. Consider now only the “→” fragment. We call this fragment minimal logic over →.

β-Reduction in $\lambda^{\rightarrow}$

β-reduction defined as before (➜ p.153), has subject reduc-

If you take the rule

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

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

$$\Gamma \vdash A \quad \text{(where } A \in \Gamma\text{)} \text{ (➜ p.48)}$$

of propositional logic.

Likewise, if you do the same with the rule

$$\frac{\Gamma \vdash e : \sigma \to \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash ee' : \tau} \mathit{app}$$

of $\lambda^{\rightarrow}$, you obtain essentially the rule

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

(➜ p.48)of propositional logic.

Finally, if you do the same with the rule

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

of $\lambda^{\rightarrow}$, you obtain essentially the rule

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

(➜ p.48)of propositional logic.

Note that in this setting, there is no analogous propositional logic rule for

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau} \; assum$$

So for the moment, we can observe a close analogy between $\lambda^{\rightarrow}$, for Σ being empty, and the → fragment of propositional logic, which is also called minimal logic over → (➜ p.170).

Such an analogy between a type theory (of which $\lambda^{\rightarrow}$ 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:

tion property[184] and is strongly normalizing[185].

$\lambda^{\rightarrow}$ is the type theory of minimal logic over $\rightarrow$.

Note that for this analogy, it is quite crucial that we have no constants (Σ is empty). Namely, this condition implies that for some types, we cannot give a closed (➜ p.148) term that has this type. For example, we can give a closed term of type $\tau \rightarrow \sigma \rightarrow \tau$, namely $\lambda xy.\, x$, while we cannot give a closed term of type $(\tau \rightarrow \tau) \rightarrow \tau$. We say that $\tau \rightarrow \sigma \rightarrow \tau$ is inhabited (➜ p.337) while $(\tau \rightarrow \tau) \rightarrow \tau$ is not inhabited.

The inhabited types correspond exactly to the formulas that are derivable in minimal logic over $\rightarrow$, and the inhabiting term is regarded as a proof.

[184] Subject reduction is the following property: reduction (➜ p.153) does not change the type of a term, so if $\vdash_{\Sigma} M : \tau$ and $M \rightarrow_{\beta} N$, then $\vdash_{\Sigma} N : \tau$.

[185] The simply-typed λ-calculus, unlike the untyped λ-calculus (➜ p.143), 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 reduc-

Example 1

$$\dfrac{\dfrac{\dfrac{}{x : \sigma,\ y : \tau \vdash x : \sigma}\,\mathit{hyp}}{x : \sigma \vdash \lambda y^{\tau} .\, x : \tau \to \sigma}\,\text{abs}}{\vdash \lambda x^{\sigma} .\, \lambda y^{\tau} .\, x : \sigma \to (\tau \to \sigma)}\,\text{abs}$$

Note the use of schematic types[186]!

For simplicity, applications of *hyp* (➜ p.168) are usually not explicitly marked in proof.

tion order.

[186] In this example, you may regard σ and τ as base types (this would require that $\sigma, \tau \in \mathcal{B}$), but in fact, it is more natural to regard them as metavariables standing for arbitrary types. Whatever types you substitute for σ and τ, you obtain a derivation of a type judgement.

This is in analogy to schematic derivations in a logic (➜ p.33).

Note also that Σ (➜ p.166) is irrelevant for the example and hence arbitrary.

Example 2

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash f : \sigma \to \sigma \to \tau \quad \Gamma \vdash x : \sigma}{\Gamma \vdash f\,x : \sigma \to \tau}\,app \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash f\,x\,x : \tau}\,app}{f : \sigma \to \sigma \to \tau \vdash \lambda x^{\sigma}.\, f\,x\,x : \sigma \to \tau}\,abs}{\vdash \lambda f^{\sigma \to \sigma \to \tau}.\, \lambda x^{\sigma}.\, f\,x\,x : (\sigma \to \sigma \to \tau) \to \sigma \to \tau}\,abs$$

Example 3

$$\begin{aligned}\Sigma &= f : \sigma \to \sigma \to \tau \\ \Gamma &= x : \sigma\end{aligned}$$

$$\dfrac{\dfrac{\dfrac{f : \sigma \to \sigma \to \tau \in \Sigma}{\Gamma \vdash f : \sigma \to \sigma \to \tau}\,\mathit{assum} \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash f\,x : \sigma \to \tau}\,\mathit{app} \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash f\,x\,x : \tau}\,\mathit{app}$$

Note that this time, f is a constant[187].

We will often suppress applications of *assum* (➜ p.168).

[187] In Example 3, we have $f : \sigma \to \sigma \to \tau \in \Sigma$, and so f is a constant (➜ p.166).

In Example 2, we have $f : \sigma \to \sigma \to \tau \in \Gamma$, and so f is a variable (➜ p.166).

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 $\lambda^{\to}$ language, then it is intuitive that there should be constants $1, 2, \ldots$ denoting the numbers. If

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

Type construction:

- Type construction[188] is decidable.
- There is a practically useful implementation for type-construction (Hindley-Milner algorithm $\mathcal{W}$ [Mil78, NN99]).

Term congruence[189] ($e =_{\alpha\beta\eta} e'$? (➜ p.158)) is decidable.

$1, 2, \ldots$ were variables, then we could write strange expressions like $\lambda 2^{\mathbb{N}\to\mathbb{N}}.\, y$, so we could use 2 as a variable of type $\mathbb{N} \to \mathbb{N}$.

[188] Type construction is the problem of given a Σ, Γ and e, finding a τ such that $\Gamma \vdash_{\Sigma} e : \tau$.

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

[189] $\alpha\beta\eta$-conversion is defined as for $\lambda^{\to}$ (➜ p.158). 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 (➜ p.179) and type classes (➜ p.182).

As we will see later (➜ p.196), this is the universal representation for object logics in Isabelle.

Polymorphism: Intuition

In functional programming, the function *append* for concatenating two lists works the same way on integer lists and on character lists: *append* is polymorphic[190].

Type language (➜ p.162) must be generalized to include type variables (denoted by $\alpha, \beta \ldots$) and type constructors.

Example: *append* has type α *list* $\rightarrow$ α *list* $\rightarrow$ α *list*, and by type instantiation, it can also have type, say, *int list* $\rightarrow$ *int list* $\rightarrow$ *int list*.

[190] In functional programming, you will come across functions that operate uniformly on many different types. For example, a function *append* for concatenating two lists works the same way on integer lists and on character lists. Such functions are called polymorphic.

More precisely, this kind of polymorphism, where a function does exactly the same thing regardless of the type instance, is called parametric polymorphism, as opposed to ad-hoc polymorphism (➜ p.183).

In a type system with polymorphism, the notion of base type (➜ p.163) (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 (➜ p.65). However, other conven-

Polymorphism: Two Syntaxes

- Syntax for polymorphic types ($\mathcal{B}$ a set of type constructors[191] including $\rightarrow$), $T \in \mathcal{B}$, α is a type variable)

  $$\tau ::= \alpha \mid (\tau, \ldots, \tau)\, T \text{ (➜ p.16)}$$

  Examples: $\mathbb{N}, \mathbb{N} \rightarrow$ (➜ p.163)$\mathbb{N}$, $\alpha\ \mathit{list}$, $\mathbb{N}\ \mathit{list}$, $(\mathbb{N}, \mathit{bool})\ \mathit{pair}$.

- Syntax for (raw (➜ p.163)) terms as before (➜ p.163):

  $$e \ ::= \text{ (➜ p.16) } x \mid c \mid (ee) \mid (\lambda x^{\tau} \text{ (➜ p.163)}. e)$$

  $(x \in \mathit{Var}, c \in \mathit{Const}$ (➜ p.164))

---

tions exist, even within Isabelle (➜ p.188).

A type constructor of arity > 0 is called type operator by some authors [GM93, page 196], but we do not follow this terminology. Also, those authors say type constant for what we call "type constructor" (i.e., of arity 0 as well as > 0), but again, we do not follow this terminology: for us a type constant has arity 0.

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

[191] As before (➜ p.162), we define a type language, i.e., a language consisting of types, and a particular type language is characterized by 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 (➜ p.67). The intention is that a type constructor may be applied to types.

Following the conventions of ML [Pau96], we write types in postfix notation (➜ p.65), something we have not seen

Polymorphic Type Assignment Calculus

Type substitutions (denoted Θ) defined in analogy to substitutions in FOL[192]. Apart from application of Θ in rule *assum*, type assignment is as for $\lambda^{\rightarrow}$ (➜ p.168):

$$\frac{c:\tau\in\ \text{(➜ p.168)}\Sigma}{\Gamma\vdash c:\tau\Theta}\,\mathit{assum}^{*}\qquad \Gamma, x:\tau,\Delta\vdash x:\tau\quad \mathit{hyp}\ \text{(➜ p.168)}$$

$$\frac{\Gamma\vdash e:\sigma\rightarrow\tau\quad \Gamma\vdash e':\sigma}{\Gamma\vdash ee':\tau}\,\mathit{app}\qquad \frac{\Gamma, x:\sigma\ \text{(➜ p.168)}\vdash e:\tau}{\Gamma\vdash\lambda x^{\sigma}.\,e\ :\sigma\rightarrow\tau}\,\text{abs}$$

*: Θ is any type substitution.

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 (➜ p.188), however written infix rather than postfix.

So the $\mathcal{B}$ is some fixed set "defined by the user", but it should definitely always include $\rightarrow$.

[192] A type substitution replaces a type variable by a type, just like in first-order logic (➜ p.83), a substitution replaces a variable by a term.

Type Classes: Intuition

Type classes[193] are a way of . . .

[193] Type classes are a way of "making ad-hoc polymorphism (➜ p.183) 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 (➜ p.65)) 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 poymorphism (➜ p.179), “making ad-hoc polymorphism[194] 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 formalization here, and refer to [HHPW96, Nip93, NP93] for details.

but also somewhat different from ad-hoc polymorphism (➜ p.183): The types of the symbols need not be declared separately. E.g., one has to declare only once that $\leq$ is of type $(\alpha :: \mathit{ord}$ (➜ p.184)$, \alpha)$.

[194] Ad-hoc polymorphism, also called overloading, refers to functions that do different (although usually similar) things on different types. For example, a function $\leq$ may be defined as ’a’ $\leq$ ’b’ ... on characters and $1 \leq 2 \ldots$ on integers. In this case, the symbol $\leq$ must be declared and defined separately for each type.

This is in contrast to parametric pomorphism (➜ p.179), but also somewhat different from type classes.

Type classes are a way of “making ad-hoc polymorphism less ad-hoc” [HHPW96, WB89].

Type Classes in Isabelle

- Syntactic classes[195] (similarly as in Haskell): E.g., declare that there exists a class *ord* which is a subclass of class *term*, and that for any $\tau :: ord$, the constant $\leq$ is defined and has type $\tau \rightarrow \tau \rightarrow bool$. Isabelle has syntax for this.

[195] A syntactic class is a class of types for which certain symbols are declared to exist. Isabelle has a syntax for such declarations. E.g., the declaration

```
sort ord < term
const <= : ['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" $ord \prec term$. The class *term* must be defined elsewhere.

The second line declares a symbol `<=`. Such a declaration is preceded by the keyword `const`. The notation $\alpha :: ord$ stands for a type variable constrained to be in class *ord*. So `<=` is declared to be of type $[\alpha :: ord, \alpha] \Rightarrow bool$, meaning that it takes two arguments of a type in the class *ord* and returns a term of type *bool*. The symbol $\Rightarrow$(=>) is the function type arrow (➜ p.163) in Isabelle. Note that the

- Axiomatic classes[196]: 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 (➜ p.185).

---

second occurrence of α is written without :: *ord*. This is because it is enough to state the class constraint once.

Note also that $[\alpha :: \mathit{ord}, \alpha] => \mathit{bool}$ is in fact just another way of writing $\alpha :: \mathit{ord} => \alpha => \mathit{bool}$, similarly as for goals (➜ p.270).

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

[196] In addition to declaring the syntax of a type class, one can axiomatize the semantics of the symbols. Again, Isabelle has a syntax for such declarations. E.g., the declaration

```
axclass order < ord
  order_refl: ''x <= x ''
  order_trans: ''[| x <= y; y <= z |] ==> x <= z''
  ...
```

may form part of an Isabelle theory file. It declares an axiomatic type class *order* which is a subclass of *ord* defined above.

The next two lines are the axioms. Here, `order_refl` and `order_trans` are the names of the axioms. Recall that

To use a class, we can declare members[197] of it, e.g., $\mathbb{N}$ is a member of *ord*.

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

Whenever an Isabelle theory declares (➜ p.186) 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.

[197] One also speaks of a type being an instance of a type class, but this is slightly confusing, since we also say that a type can be an instance of another type, e.g., $\mathbb{N} \to \mathbb{N}$ is an instance of α, since $\alpha[\alpha \leftarrow (\mathbb{N} \to \mathbb{N})] = \mathbb{N} \to \mathbb{N}$ (➜ p.181). 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 classes[198], say $\mathcal{K} = \{ord, order, lattice, \ldots\}$,
- a set of type constructors[199], say

---

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 (➜ p.184) 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" $\tau :: \kappa$.

[198] The set $\mathcal{K}$ we gave is incomplete and just exemplary.

So the set of type classes involved in an Isabelle theory is a finite set of names (written lower-case), typically including *ord*, *order*, and *lattice*.

We have seen some Isabelle syntax for declaring the type classes previously (➜ p.184).

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

[199] As before, the set $\mathcal{B}$ we gave is incomplete (there are "…") and just exemplary. We might call $\mathcal{B}$ a type

$\mathcal{B} = \{\mathit{bool}, \_ \rightarrow \_$[200]$, \mathit{ind}, \_\ \mathit{list}, \_\ \mathit{set} \ldots\}$,

- a set of constants (➜ p.164) *Const* and a set of variables (➜ p.164) *Var*,

we define

---

signature (➜ p.162).

Note also that an _ is used to denote the arity of a type constructor (➜ p.179).

– _ *list* means that *list* is unary type constructor;

– _ → _ means that → 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*, →, and *list*. Note however that *bool* is fundamental (since object level predicates are modeled as functions taking terms to a Boolean), and so is →, the constructor (➜ p.188) of the function space between two types (➜ p.163).

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

[200] In $\lambda^{\rightarrow}$, types were built from base types using a "special

- Polymorphic types[201]:
  τ ::= (➜ p.16) α | α:: κ | $(\tau, \ldots, \tau)\ T$
- Raw (➜ p.163) terms (as before (➜ p.163)):
  e ::= (➜ p.16) x | c | (ee) | $(\lambda x^{\tau}$ (➜ p.163)$. e)$

(α is type variable, $T \in \mathcal{B}$ (➜ p.187), $\kappa \in \mathcal{K}$ (➜ p.187), $x \in \mathit{Var}$, $c \in \mathit{Const}$ (➜ p.164))

symbol" → (➜ p.163).

When we generalize $\lambda^{\rightarrow}$ to a λ-calculus with polymorphism, this "special symbol" becomes a type constructor. However, the syntax is still special, and it is interpreted in a particular way (➜ p.163).

[201] τ ::= (➜ p.16) α | α:: κ | $(\tau, \ldots, \tau)\ T$
(α is type variable)

is a grammar defining what polymorphic types are (syntactically). As before (➜ p.166), τ is the non-terminal (➜ p.16) we use for (now: polymorphic) types.

This grammar is not exemplary but generic, and it deserves a closer look.

A type variable is a variable that stands for a type, as opposed to a term. We have not given a grammar for type variables, but assume that there is a countable set of type variables disjoint from the set of term variables. We use α as the non-terminal for a type variable (abusing notation, we

Type Assignment Calculus with Type Classes

Assume some syntax for declaring $\tau :: \kappa$ (➜ p.186) and $\kappa \prec \kappa'$ (➜ p.184). In addition introduce the rule

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

Type assignment rules as before (➜ p.181), but type substitution Θ in

$$\frac{c : \tau \in\ (\text{➜ p.168})\Sigma}{\Gamma \vdash c : \tau\Theta}\,\mathit{assum}$$

must respect class constraints (➜ p.184): for each $\alpha :: \kappa$ occurring in τ where $\alpha\Theta = \sigma$, judgement (➜ p.187) $\sigma :: \kappa$ must hold.

---

often also use α to denote an actual type variable).

First, note that a type variable may be followed by a class constraint (➜ p.184) $:: \kappa$ (recall (➜ p.187) that κ is the nonterminal 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 (➜ p.184).

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.

Example

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

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

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

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

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

8.4 Higher-Order Unification

The λ-calculus is “the” (➜ p.142) metalogic. Hence we now (sometimes) call its variables “metavariables” (➜ p.29) for emphasis and we precede them with “?”. E.g. they can stand for object (➜ p.29)-level formulae (➜ p.18). More details later (➜ p.196).

Two issues concerning metavariables are:

- suitable renamings[202] of metavariables;
- unification[203] before rule application.

[202] Whenever a rule is applied, the metavariables occurring in it must be renamed to fresh variables to ensure that no metavariable in the rule has been used in the proof before.

The notion fresh is often casually used in logic, and it means: this variable has never been used before. To be more precise, one should say: never been used before in the relevant context.

[203] The mechanism to instantiate metavariables as needed is called (higher-order) unification. Unification is the process of finding a substitution (➜ p.149) that makes two terms equal.

We will now see more formally what it is and later also where it is used (➜ p.263).

What Is Higher-Order Unification?

Unification of terms e, e': find substitution (➜ p.147) θ for metavariables such that $e\theta =_{\alpha\beta\eta} e'\theta$.

Examples[204]:

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

Why higher-order (➜ p.219)? Metavariables may be instantiated to functions, e.g. $[?P \leftarrow \lambda y.y + y]$.

[204] A solution for $?X + ?Y =_{\alpha\beta\eta} x + x$ is $[?X \leftarrow x, ?Y \leftarrow x]$.

A solution for $?P(x) =_{\alpha\beta\eta} x + x$ is $[?P \leftarrow (\lambda y.y + y)]$.

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

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

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

Higher-Order Unification: Facts

- Unification modulo[205] $\alpha\beta$ (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 (➜ p.309)) are decidable.
- HO-unification has possibly infinitely many solutions.

We will look at some of these issues again later (➜ p.299).

[205] Unification of terms e, e' modulo $\alpha\beta$ means finding a substitution θ for metavariables such that $\theta(e) =_{\alpha\beta} \theta(e')$.

Likewise, unification of terms e, e' modulo $\alpha\beta\eta$ 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 (➜ p.143).
- β-reduction (➜ p.153) is the notion of "computing" in λ-calculus.
- λ-calculus is Turing-complete (➜ p.161).
- $\lambda^{\rightarrow}$ (➜ p.162) restricts syntax to "meaningful" λ-terms.
- Add-on features: Polymorphism and type classes (➜ p.178).
- The λ-calculus will be used to represent syntax of object logics. λ-terms[206] stand for object terms/formulae. This will be explained next lecture (➜ p.196).
- HO-unification (➜ p.192) is important in applying proof rules.

[206] So just like first-order logic (➜ p.67), the λ-calculus has a syntactic category called terms. But the word "term" has a meaning for the λ-calculus that differs from the meaning it has for first-order logic, and so one can say λ-term for emphasis.

Note that at this stage (➜ p.236), we have no syntactic category called "formula" for the λ-calculus.

9 Encoding Syntax

Metatheory: Motivation

Previously (➜ p.141), we have seen the (polymorphically (➜ p.179)) typed λ-calculus (➜ p.162) (with type classes (➜ p.182)).

Now, we will see how the typed λ-calculus can be used as a metalanguage (➜ p.111) ("metalogic") for representing[207] the syntax of an object logic, e.g. first-order logic (➜ p.60).

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 $\ulcorner\phi\urcorner : o$[208].

Later (➜ p.228): representing proofs/provability. Then we will really have a metalogic, not just metalanguage.

[207] In the following, we will distinguish between the object logic and the metalogic. We have already seen this kind of distinction before (➜ p.29).

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 (➜ p.228) are about.

[208]

$\phi \in$ *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'[209] only once

+ Shared support for automation[210]

+ Conceptual framework[211] for exploring what a logic is

But

+/− Metalayer[212] between user and logic

− Makes assumptions[213] about structure of logic

9.1 $\lambda^{\rightarrow}$: Review

[209] By the core we mean the syntax and proof rules of the metalogic. These should be simple, so that one can be reasonably confident that the implementation is correct.

[210] There are some general techniques involved in automating the search for a proof that work for various object logics. It is therefore useful to implement these techniques on a higher level, rather than considering each object logic individually.

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

[212] Having a logic and a metalogic can be very mind-boggling. We already experienced that when working with Isabelle, it is sometimes confusing to know whether we are at the level of a particular theory, or at the level of general Isabelle syntax, or at the level of ML, the programming language that Isabelle is implemented in.

[213] Designing a metalogic is a bold endeavor.

How are we supposed to know that the metalogic is expressive enough to encode any object logic someone might

$\lambda^{\rightarrow}$ is sufficient for presentation here (no polymorphism (➜ p.179), type classes (➜ p.182)).

- Syntax for types ($\mathcal{B}$ a set of base types (➜ p.162), $T \in \mathcal{B}$)

  $$\tau ::= T \mid \tau \rightarrow \tau \text{ (➜ p.16)}$$

  Examples: $\mathbb{N}$, $\mathbb{N} \rightarrow$ (➜ p.163)$\mathbb{N}$, $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$, $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ (➜ p.163)

- Syntax for terms: λ-calculus (➜ p.145) augmented with types (➜ p.163)

  $$e \ ::= \text{ (➜ p.16) } x \mid c \mid (ee) \mid (\lambda x^{\tau}.\, e)$$

  ($x \in \mathit{Var}$, $c \in \mathit{Const}$ (➜ p.164))

---

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 (➜ p.96)) for which the theorem does not hold [BM00].

Type Assignment

- Signature (➜ p.165) $\Sigma ::= \langle\,\rangle \mid \Sigma, c:\tau$ (➜ p.166).
- Context (➜ p.165) $\Gamma ::= \langle\,\rangle \mid \Gamma, x:\tau$ (➜ p.166).
- Type assignment rules (➜ p.168)

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

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

9.2 Representing Syntax of Propositional Logic

Let *Prop*[214] be our object logic (➜ p.197):

$$P \ ::= \ (\text{➜ p.16})\ x \ \mid \ \neg P \ \mid \ P \wedge P \ \mid \ P \rightarrow P$$

Let $\lambda^{\rightarrow}$ be our metalogic (➜ p.197). Declare

- $\mathcal{B} = \{o\}$ (➜ p.197).
- Signature (➜ p.166) assigns types to constants[215]:

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

[214] We consider here the fragment of propositional logic containing the logical symbols (➜ p.102) $\neg, \wedge, \rightarrow$, and we call it *Prop*. We chose this small fragment because it is sufficient for our purposes, namely to demonstrate how encoding syntax in $\lambda^{\rightarrow}$ works. It would be trivial to adapt everything in the sequel to include $\vee$ or $\perp$ (➜ p.16).

[215] Now the object/meta distinction starts becoming mindboggling!

We declare

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

and so on the level of our metalogic (➜ p.197) $\lambda^{\rightarrow}$, *not*, *and*, and *imp* are constants (➜ p.166). However, these constants represent the logical symbols (➜ p.102) of the object logic.

Note the types of the constants:

not has type $o \rightarrow o$, so it takes a proposition and returns a proposition.

and and *imp* have type $o \rightarrow o \rightarrow o$, so each takes two (➜ p.156) propositions and returns a proposition.

- Context (➜ p.166) assigns types to variables[216].

This approach is called first-order syntax (see later (➜ p.220)).

[216] We identify metalevel variables and object level propositional variables. Hence Γ should contain expressions of the form $a : o$, where a is a $\lambda^{\rightarrow}$ variable, representing a propositional variable. Note that under this agreement, Γ should not contain expressions like, e.g., $a : o \rightarrow o$.

Example of First-Order Syntax

$a : o \vdash \mathit{imp}\ (\mathit{not}\ a)\ a : o$[217]

$$\dfrac{\dfrac{a : o \vdash \mathit{imp} : o \to o \to o \qquad \dfrac{a : o \vdash \mathit{not} : o \to o \quad a : o \vdash a : o}{a : o \vdash \mathit{not}\, a : o}}{a : o \vdash \mathit{imp}\,(\mathit{not}\, a) : o \to o} \qquad a : o \vdash a : o}{a : o \vdash \mathit{imp}\,(\mathit{not}\, a)\, a : o}$$

Applications of *hyp* (➜ p.174) and *assum* (➜ p.176) suppressed. Otherwise always rule *app* (➜ p.168).

[217] $a : o \vdash \mathit{imp}\ (\mathit{not}\ a)\ a : o$ is a judgement (➜ p.167) in $\lambda^{\to}$, which may or may not be provable.

If we set up everything correctly and if $a : o \vdash \mathit{imp}\ (\mathit{not}\ a)\ a : o$ is provable, then the judgement represents the fact $\neg a \to a$ is a proposition.

In this sense, we could then say that derivability in $\lambda^{\to}$ 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

$a : o \vdash \mathit{not}\ (\mathit{imp}\ a)\ a : o$[218]

$$\dfrac{a : o \vdash \mathit{not} : o \to o \qquad \dfrac{a : o \vdash \mathit{imp} : o \to o \to o \quad a : o \vdash a : o}{a : o \vdash \mathit{imp}\, a : o \to o}}{???}$$

No proof possible! (Requires analysis of normal forms[219].)

[218] $a : o \vdash \mathit{not}\ (\mathit{imp}\ a)\ a : o$ is a judgement (➜ p.167) in $\lambda^{\rightarrow}$ which may or may not be provable.

If we set up everything correctly and if $a : o \vdash \mathit{not}\ (\mathit{imp}\ a)\ a : o$ 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 $\mathit{not}\ (\mathit{imp}\ a)\ a$ corresponds to $(\rightarrow\ a)\neg a$.

We will see that $a : o \vdash \mathit{not}\ (\mathit{imp}\ a)\ a : o$ isn't provable, and this reflects the fact that there is no proposition represented by $\mathit{not}\ (\mathit{imp}\ a)\ a$.

[219] Generally, it is difficult to prove that a proof of a given judgement within a given proof system (➜ p.14) 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 bijection[220] $\ulcorner\cdot\urcorner : \mathit{Prop} \to o$ that is

- adequate: each proposition in *Prop* can be represented by a $\lambda^{\to}$-term of type o:

$$\text{If } P \in \mathit{Prop} \text{ then } \Gamma \vdash \ulcorner P \urcorner : o$$

- faithful: each $\lambda^{\to}$-term of type o represents a proposition in *Prop*:

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

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].

[220] In general mathematical terminology, a bijection between A and B is a mapping $f : A \to B$ 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: $(\neg a) \to b \in \mathit{Prop}$ therefore $\mathit{imp}\ (\mathit{not}\ a)\ b : o$

Formalize mapping $\ulcorner\cdot\urcorner$:

$$\begin{array}{rcll} \ulcorner x \urcorner & = & x & \text{for } x \text{ a variable} \\ \ulcorner \neg P \urcorner & = & \mathit{not}\, \ulcorner P \urcorner & \\ \ulcorner P \wedge Q \urcorner & = & \mathit{and}\, \ulcorner P \urcorner\, \ulcorner Q \urcorner & \\ \ulcorner P \to Q \urcorner & = & \mathit{imp}\, \ulcorner P \urcorner\, \ulcorner Q \urcorner & \end{array}$$

Formal statement accounts for variables:

If $P \in \mathit{Prop}$, and if for each propositional variable x in P, we have $x : o \in \Gamma$, then $\Gamma \vdash \ulcorner P \urcorner : o$. Proof by induction[221].

[221] If $P \in \mathit{Prop}$, 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* (➜ p.168).

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

Consider the propositional formula $\neg P$. We have $\ulcorner \neg P \urcorner = \mathit{not}\ \ulcorner P \urcorner$. 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 $\Gamma \vdash \mathit{not} : o \to o$ by rule *assum* (➜ p.168), and so $\Gamma \vdash \mathit{not}\ \ulcorner P \urcorner : o$ by rule *app* (➜ p.168).

Now consider the propositional formula $P \wedge Q$. We have $\ulcorner P \wedge Q \urcorner = \mathit{and}\ \ulcorner P \urcorner\ \ulcorner Q \urcorner$. 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 $\ulcorner\cdot\urcorner^{-1}$

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

For bijection (➜ p.205), should have $\ulcorner\ulcorner P\urcorner\urcorner^{-1} = P$ and $\ulcorner\ulcorner t\urcorner^{-1}\urcorner = t$. Former is trivial[222], but what about latter?

over $\Gamma \vdash \mathit{and} : o \rightarrow o \rightarrow o$ by rule *assum* (➜ p.168), and so $\Gamma \vdash \mathit{and}\ \ulcorner P\urcorner\ \ulcorner Q\urcorner : o$ by two applications of rule *app* (➜ p.168).

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

[222] By the definition of *Prop* (➜ p.201) and the definition of $\ulcorner\cdot\urcorner$ (➜ p.206), it is clear that $\ulcorner P\urcorner$ is defined for all $P \in \mathit{Prop}$. It is very easy to show by induction on *Prop* that $\ulcorner\ulcorner P\urcorner\urcorner^{-1} = P$.

Here is an example of a proof by induction on *Prop*. (➜ p.206)

Obviously, everything we say here depends on the particular fragment (➜ p.201) 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 $t = \mathit{not}\,((\lambda x^o.\, x)a)$, we have $a : o \vdash t : o$

$$\dfrac{a : o \vdash \mathit{not} : o \to o \qquad \dfrac{\dfrac{a : o, x : o \vdash x : o}{a : o \vdash \lambda x^o.\, x : o \to o}\,\mathit{abs} \qquad a : o \vdash a : o}{a : o \vdash (\lambda x^o.\, x)\, a : o}\,\mathit{app}}{a : o \vdash \mathit{not}\,((\lambda x^o.\, x)\, a) : o}\,\mathit{app}$$

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

Normal Forms

If $t : o$, then there exists a t' such that $t =_{\beta\eta}$ (➜ p.158)t', where $t' : o$ and t' is in normal form[223], obtained by applying β-reduction as long as possible.

Bijection Theorem: The encoding $\ulcorner\cdot\urcorner$ is a bijection

Examples:

$$\begin{array}{lll} \mathit{not}\,((\lambda x^o.\,x)\,a) & =_{\beta\eta} & \mathit{not}\,a \\ \mathit{not} & =_{\beta\eta} & \lambda x^o.\,\mathit{not}\,x \\ \mathit{imp}\,(\mathit{not}\,((\lambda x^o.\,x)\,a)) & =_{\beta\eta} & \lambda x^o.\,\mathit{imp}\,(\mathit{not}\,a)\,x \end{array}$$

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 (➜ p.158).

You may wonder: Why is there such a thing as a maximal η-expansion? Can't I expand a λ-term to $\lambda x_1 \ldots x_n.\, M\, x_1 \ldots x_n$ 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 $\lambda x.\,\mathit{not}\,x$ since *not* is of function type: it has type $o \to o$ (➜ p.201). Therefore, $\mathit{not}\,x$ can be assigned a type (➜ p.168), which is an intermediate step in typing $\lambda x.\,\mathit{not}\,x$:

[223] To be precise, the normal form use here is the so-called canonical $\beta\eta$-long normal form.

between propositional formulae with variables in Γ[224] and canonical terms t', where $\Gamma \vdash t' : o$.

Corollary: If $t : o$[225] then $t =_{\beta\eta} t'$ and $\ulcorner t' \urcorner^{-1} \in \mathit{Prop}$ for some canonical t'.

$$\dfrac{\dfrac{\Gamma, x : o \vdash \mathit{not} : o \to o \quad \Gamma, x : o \vdash x : o}{\Gamma, x : o \vdash \mathit{not}\, x : o}\,\mathit{app}}{\Gamma \vdash \lambda x.\, \mathit{not}\, x : o \to o}\,\mathit{abs}$$

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

Effectively, when a term of type $\tau_1 \to \tau_n \to \tau$ is η-expanded, it will have the form $\lambda x_1 x_2 \ldots x_n.e$.

Normal forms are unique (➜ p.160).

[224] Saying that a propositional formula has variables in Γ is an abuse of terminology, i.e., it isn't exactly true, but it is trusted that the reader can guess the exact formulation.

What we mean is: a propositional formula such that for each propositional variable x occurring in the formula, we have $x : o \in \Gamma$.

[225] Simply writing $t : o$ is again a bit sloppy. We should write: $\Gamma \vdash t : o$ for some Γ containing only expressions of

9.3 Representing Syntax of First-Order Logic

In *Prop*, we only have the syntactic category (➜ p.18) of formulae (propositions), represented in $\lambda^{\rightarrow}$ by the type o (➜ p.197).

In first-order[226] logic, we also have the syntactic category (➜ p.67) of terms. For representation in $\lambda^{\rightarrow}$, we now introduce type i, so $\mathcal{B} = \{i, o\}$.

Just like $\Gamma \vdash a : o$ means that a represents a proposition (➜ p.203), $\Gamma \vdash t : i$ means that t represents a term.

the form $x : o$, where x is a propositional variable in *Prop*.

[226] In the previous section (➜ p.201), we have seen how we can use first-order syntax (of $\lambda^{\rightarrow}$) to represent the syntax of an object logic, then *Prop*. We haven't really understood yet why we speak of first-order syntax, but note that the notion "first-order" refers to $\lambda^{\rightarrow}$, 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 $\lambda^{\rightarrow}$ syntax (the metalanguage). To avoid confusion, it is best to imagine that it is a mere coincidence that both the object and the metalanguage (➜ p.220) 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 (➜ p.197) level language[227]:

$$\begin{array}{llcl}
\text{Terms} & T & ::= \ (\text{➜ p.16}) & x \mid 0 \mid s(T) \mid T+T \mid T\times T \\
\text{Formulae} & F & ::= & T=T \mid \neg F \mid F\wedge F \mid F\rightarrow F
\end{array}$$

In $\lambda^{\rightarrow}$ (on metalevel), define signature (➜ p.201) $\Sigma = \Sigma_{\mathcal{F}}$[228] $\cup \Sigma_{\mathcal{P}} \cup \Sigma_{\mathcal{C}}$:

$$\begin{array}{lcl}
\Sigma_{\mathcal{F}} & = & \langle \mathit{zero} : i,\ \mathit{succ} : i \rightarrow i,\ \mathit{plus} : i \rightarrow i \rightarrow i, \\
 & & \mathit{times} : i \rightarrow i \rightarrow i\rangle \\
\Sigma_{\mathcal{P}} & = & \langle \mathit{eq} :\ i \rightarrow i \rightarrow o\rangle \\
\Sigma_{\mathcal{C}} & = & \langle \mathit{not} : o \rightarrow o,\ \mathit{and} : o \rightarrow o \rightarrow o,\ \mathit{imp} : o \rightarrow o \rightarrow o\rangle
\end{array}$$

[227] With this grammar, we specify a certain language of a fragment (since quantifiers, $\vee$, and $\perp$ are missing) of first-order logic.

Alternatively, we could say that $\mathcal{F} = \{0, s, +, \times\}$ (➜ p.67) and $\mathcal{P} = \{=\}$ (➜ p.67). However, the way we defined first-order logic (➜ p.68), the language thus obtained would also include quantifiers, $\vee$, and $\perp$. For the moment we want to restrict ourselves to the fragment given by the grammar for FOA.

[228] We have defined

$$\begin{array}{lcl}
\Sigma_{\mathcal{F}} & = & \langle \mathit{zero} : i,\ \mathit{succ} : i \rightarrow i,\ \mathit{plus} : i \rightarrow i \rightarrow i,\ \mathit{times} : i \rightarrow i \rightarrow i\rangle \\
\Sigma_{\mathcal{P}} & = & \langle \mathit{eq} : i \rightarrow i \rightarrow o\rangle
\end{array}$$

$\mathit{zero} : i$ means: viewed on the object level, 0 is a term. $\mathit{plus} : i \rightarrow i \rightarrow i$ means: viewed on the object level, plus is a function that takes two (➜ p.156) terms and returns a term. $\mathit{eq} : i \rightarrow i \rightarrow o$ means: viewed on the object level, $=$ is a predicate that takes two (➜ p.156) terms and returns a proposition.

Example: $\ulcorner x + s(0) \urcorner$[229] $= \mathit{plus}\ x\ (\mathit{succ}\,\mathit{zero})$.

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

Recall that we encoded the non-logical (➜ p.102) 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 $\Gamma_{\mathcal{F}} \cup \Gamma_{\mathcal{P}}$ instead of our $\Sigma_{\mathcal{F}} \cup \Sigma_{\mathcal{P}}$. This is in line with Perlis' epigram (➜ p.176). We will sometimes take this approach in the exercises as the encoding of $\lambda^{\rightarrow}$ in Isabelle makes it more straightforward to play around with different Γ's than with different Σ's.

[229] We extend the definition of $\ulcorner \cdot \urcorner$ (➜ p.206) as follows:

$$\begin{array}{rcl} \ulcorner x \urcorner & = & x \\ \ulcorner 0 \urcorner & = & \mathit{zero} \\ \ulcorner s\ t \urcorner & = & \mathit{succ}\,\ulcorner t \urcorner \\ \ulcorner r + t \urcorner & = & \mathit{plus}\,\ulcorner r \urcorner\,\ulcorner t \urcorner \end{array}$$

Encoding FOL in General

In general, to encode some first-order language (➜ p.67), we must define $\Sigma_{\mathcal{F}}$ and $\Sigma_{\mathcal{P}}$ so that for each n-ary $f \in \mathcal{F}$, $p \in \mathcal{P}$

$$f_{enc} : \underbrace{i \to \ldots \to i}_{n \text{ times}} \to i \ \in \ \Sigma_{\mathcal{F}},$$
$$p_{enc} : \underbrace{i \to \ldots \to i}_{n \text{ times}} \to o \ \in \ \Sigma_{\mathcal{P}},$$

and then $\ulcorner f(t_1, \ldots, t_n) \urcorner = f_{enc} \ulcorner t_1 \urcorner \ldots \ulcorner t_n \urcorner$ and $\ulcorner p(t_1, \ldots, t_n) \urcorner = p_{enc} \ulcorner t_1 \urcorner \ldots \ulcorner t_n \urcorner$.

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

$$\ulcorner r \times t \urcorner \ = \ times \ulcorner r \urcorner \ulcorner t \urcorner$$

Note that here, on the object level, x is a first-order variable (a variable is a term (➜ p.62)), and hence on the metalevel, it has type i (➜ p.211).

Quantifiers in First-Order Syntax

Along the same lines (➜ p.212), one might suggest

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

But this approach has some problems:

- Variables are also terms, so "$\mathit{var} \subseteq i$"[230]? No subtyping!
- *all* is not a binding operator (➜ p.69) in $\lambda^{\to}$. E.g., $(p(x) \wedge \forall x.\, q(x))[x \leftarrow a]$ cannot be modeled[231] as $(\mathit{and}\ (p\ x)(\mathit{all}\ x\ (q\ x)))[x \leftarrow a]$.

[230] In first-order logic, variables are not a syntactic category (➜ p.67) of their own, but rather they are a "subcategory" 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 $\lambda^{\to}$.

[231] There is a notion of substitution (➜ p.149) in $\lambda^{\to}$, hence on the metalevel. But if we define $\bigwedge : \mathit{var} \to \mathit{Prop} \to \mathit{Prop}$, it is just a constant like any other on the level of $\lambda^{\to}$, and hence $(\Rightarrow\ (p\ x)(\bigwedge\ x\ (q\ x)))[x \leftarrow a] = (\Rightarrow\ (p\ a)(\bigwedge\ a\ (q\ a)))$, and not $(\Rightarrow\ (p\ a)(\bigwedge\ x\ (q\ x)))$ as one should expect (➜ p.147).

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 $\lambda^{\to}$ level,

Example, full FOA (➜ p.212): $F ::= \ldots \forall x.\, A \;\mid\; \exists x.\, A$
$\Sigma = \Sigma_{\mathcal{F}}$ (➜ p.212) $\cup\, \Sigma_{\mathcal{P}}$ (➜ p.212) $\cup\, \Sigma_{\mathcal{C}}$ (➜ p.212) $\cup\, \Sigma_{\mathcal{Q}}$:

$$\Sigma_{\mathcal{Q}} = \langle \mathit{all} : (i \to o^{232}) \to o,\ \mathit{exists} : (i \to o) \to o \rangle$$

Extend the definition of $\ulcorner . \urcorner$ (➜ p.206):

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

should the terms of type var be variables or constants?

One could imagine that they are variables. This means that the signature Σ (➜ p.166) would not contain any constants of type var or $\ldots \to \mathit{var}$. The only terms of type var would be variables. In this case, a $\lambda^{\to}$ term like $(\bigwedge\ x\ (q\ x)))$ could only be typed in a context Γ containing $x : \mathit{var}$, i.e., x would be a free variable of the term.

Alternatively, one could imagine that they are constants. The signature signature Σ (➜ p.166) would contain expressions of the form $x : \mathit{var}$, where x would be a $\lambda^{\to}$ 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 $\lambda^{\to}$.

In either case, the operation of substitution on the metalevel is of no use for implementing substitution on the object level.

[232] Some intuition: a proposition is represented by a term of type o. Now a term of type $i \to o$ represents a proposition

Adequacy and faithfulness as before[233].

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 (➜ p.222).

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

[233] Terms and formulae are represented by (canonical) members of i and o. The principle is similar as for *Prop* (➜ p.205).

Examples

$$\begin{array}{rcl}
\ulcorner\forall x.\, x = x\urcorner \ (\text{➜ p.216}) & = & \mathit{all}(\lambda x^i.\, \mathit{eq}\, x\, x\ (\text{➜ p.216})) \\
\ulcorner\forall x.\, \exists y.\, \neg(x + x = y)\ (\text{➜ p.216})\urcorner & = & \\
\multicolumn{3}{r}{\mathit{all}(\lambda x^i.\, \mathit{exists}(\lambda y^i.\, \mathit{not}\,(\mathit{eq}\,(\mathit{plus}\, x\, x)\, y)))}
\end{array}$$

Example derivation (all but one steps use rule *app* (➜ p.168)):

$$\dfrac{\vdash \mathit{all} : (i \to o) \to o \qquad \dfrac{\dfrac{\dfrac{x : i \vdash \mathit{eq} : i \to i \to o \quad x : i \vdash x : i}{x : i \vdash \mathit{eq}\, x : i \to o} \qquad x : i \vdash x : i}{x : i \vdash \mathit{eq}\, x\, x : o}}{\vdash \lambda x^i.\, \mathit{eq}\, x\, x : i \to o}\ \mathit{abs}}{\vdash \mathit{all}(\lambda x^i.\, \mathit{eq}\, x\, x) : o}$$

Order

Order of a type: For type τ written $\tau_1 \to \ldots \to \tau_n$, right associated (➜ p.163), $\tau_n \in \mathcal{B}$:

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

Intuition: "functions as arguments"[234].

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

A type of order > 1 is called higher order (although in higher-order unification (➜ p.192) or higher-order rewriting (➜ p.299), even order 1 is considered higher-order).

[234] A term of first-order type is a function taking (an arbitrary number of (➜ p.156)) arguments all of which must be of base type.

A term of second-order type is a function taking (an arbitrary number of (➜ p.156)) 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 (➜ p.156)) 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 (➜ p.201) (logical symbols) or non-logical symbols (➜ p.212) are first-order (hence first-order syntax):

$$\mathit{and} : o \to o \to o$$

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

$$\mathit{all} : (i \to o) \to o$$

Exercise: Summation Operator

What is the order of the summation operator $\sum$?

$$sum : i \to i \to (i \to i) \to i$$

$$\ulcorner \sum_{x=0}^{n} (x + s(s(0))) \urcorner =$$

$$sum\; zero\; n\, (\lambda x^{i}.\, plus\; x\; (succ\; succ\; zero))$$

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 (➜ p.150) λ-operator.

Thanks to this technique, standard operations on syntax need no special encoding (➜ p.215), but are supported implicitly by $\lambda^{\rightarrow}$.

We will now see this.

Binding

Binding (➜ p.69) 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 (➜ p.215).

Substitution

Recall rules for ∀ (➜ p.80):

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

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

Now apply substitution. . .

Now apply β-reduction. . .

We now understand "marked positions in a formula" (➜ p.110).

Equivalence under Bound Variable Renaming

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

$$(\forall x.\, P \leftrightarrow \text{ (➜ p.87)} \forall y.\, P[x \leftarrow y])$$

Likewise, on the metalevel, formulae obtained by bound variable renaming are α-equivalent (➜ p.158):

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

9.5 Summary of Encoding Syntax

Object Language	**Metalanguage**
Syntactic category (➜ p.211) *Term*, *Prop*	Type declaration (➜ p.211) $\mathcal{B} = \{i, o\}$
Variable x	Variable[235] x
Non-logical symb. (➜ p.212) $+$	1st-order constant (➜ p.212) $\mathit{plus} : i \rightarrow i \rightarrow i$
Logical symbol (➜ p.201) $\wedge$	1st-order constant (➜ p.201) $\mathit{and} : o \rightarrow o \rightarrow o$

[235] Although propositional variables (➜ p.16) and first-order variables (➜ p.62) are quite different concepts, the representation in $\lambda^{\rightarrow}$ uses $\lambda^{\rightarrow}$-variables for both. Technically however, there is a difference between the representations of propositional variables (➜ p.202) and first-order variables (➜ p.213). In particular, propositional variables are represented as $\lambda^{\rightarrow}$-variables of type o, and first-order variables are represented as $\lambda^{\rightarrow}$-variables of type i.

Object Language	**Metalanguage**
Binding operator (➜ p.216) ∀	2nd-order const. (➜ p.216) $all : (i \to o) \to o$
Meaningful expr. (➜ p.201) $a \wedge b \in Prop$	Member of type (➜ p.203) $(and\, a\, b) : o$

10 Isabelle's Metalogic

Representing Syntax and Proofs

- We will extend the λ-calculus to a logic (with formulae and inference rules): Isabelle's metalogic, which goes under the name of $\mathcal{M}$, Pure[236].
- The main purpose of the metalogic is to provide a framework for reasoning about proof rules, and for deriving new proof rules from existing ones..

This lecture is loosely based on Paulson's work [Pau89]. It is maybe the most challenging lecture of this course.

[236] In 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 formalizations, 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 we (➜ p.320) 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!

Deriving proof rules

In a previous lecture we proved the derived rule ∃-E (➜ p.93)

$$\dfrac{\exists x.\, P(x) \qquad \begin{array}{c}[P(x)]\\ \vdots \\ R\end{array}}{R}\ \exists\text{-}E$$

, where x does not occur in R and in any open assum
the proof of R

How can we prove this formally using a meta-calculus and a meta-logic?

Derived Proof Rules = Meta-Theorems

- When constructing proofs, there are
  - aspects that are specific to certain logics and its logical symbols (➜ p.102): the proof rules (➜ p.28);
  - aspects that reflect general principles (➜ p.14) of proof building: making and discharging assumptions, substitution (➜ p.83), side conditions (➜ p.80), etc.

  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.

What Is Formality anyway?

- Ultimately, logic and formal reasoning have to resort to natural language. Proofs of, say, the soundness of a derivation system employ the usual mathematical rigor, but that's all. Imagine this for the situation that we just want to do reasoning[237] in propositional logic (➜ p.10) and nothing else.

- We will now introduce a logic $\mathcal{M}$. Its proof system (➜ p.14) is <u>small</u>!

[237] We would formalize the language and the proof system as we did in the first lecture (➜ p.15). Any proofs of soundness and completeness or other meta-properties should be rigorous, but they still resort to natural language.

Expressing proof rules

The derived rule $\exists$-E (➜ p.93) is expressed in our metalogic as

$$\bigwedge P\ R.\, \exists x.\, Px \Rightarrow (\bigwedge x.\, Px \Rightarrow R) \Rightarrow R.$$

Here $\Rightarrow$ can be read as “provability” and $\bigwedge$ denotes that the proof does not depend on P, R resp. x. Thus the side condition of $\exists$-E is expressed by the nested $\bigwedge x$.

10.1 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 (➜ p.244).

Some variations are possible (mainly: polymorphism/type classes or not), but those are not so important for us.

$\mathcal{M}$ will be based on $\lambda^{\to}$. Would you call $\lambda^{\to}$ (➜ p.162) a logic?

So far, $\lambda^{\to}$ (➜ p.162) is not a logic (no connectives, no formulae). We will now define a particular language (➜ p.165) of $\lambda^{\to}$ that can be called a logic.

The Metalogic $\mathcal{M}$ Based on $\lambda^{\rightarrow}$

Isabelle's metalogic is based on $\lambda^{\rightarrow}$ over some $\mathcal{B}$ (➜ p.162) where *prop* $\in \mathcal{B}$, and some[238] signature Σ (➜ p.165) where

- $\Rightarrow$: $prop \rightarrow prop \rightarrow prop$ (➜ p.163) $\in \Sigma$,
- $\equiv_\sigma$: $\sigma \rightarrow \sigma \rightarrow prop \in \Sigma$ for all types σ, and
- $\bigwedge_\sigma : (\sigma \rightarrow prop) \rightarrow prop \in \Sigma$ for all types σ.

We usually omit type subscripts[239] and write $\equiv$, $\bigwedge$.

$\Rightarrow$, $\equiv$, and $\bigwedge$[240] are the logical symbols (➜ p.102) of $\mathcal{M}$. $\Rightarrow$ and $\equiv$ are written infix (➜ p.65).

Terms of type *prop* are called (meta-)formulae: types generalize syntactic categories (➜ p.67).

[238] Σ contains $\Rightarrow$, $\equiv$ and $\bigwedge$, but in addition, Σ may specify other symbols.

[239] Alternatively, we could define that

- $\equiv_\alpha$: $\alpha \rightarrow \alpha \rightarrow prop \in \Sigma$, and
- $\bigwedge_\alpha : (\alpha \rightarrow prop) \rightarrow prop \in \Sigma$,

where α is a type variable (➜ p.180).

[240] $\Rightarrow$ is called meta-implication, $\equiv$ is called meta-equality, and $\bigwedge$ is called meta-universal-quantification.

Proof System for $\mathcal{M}$

The proof system will be presented in the style of natural deduction (➜ p.23).

This is as formal as we get (for the metalogic): derivation trees in natural deduction style are authoritative.

The judgements[241], just like for natural deduction proofs (➜ p.23) in propositional logic or first-order logic, are formulae, i.e., terms of type *prop* (➜ p.236). This is in contrast to derivability judgements (➜ p.47) or type judgements (➜ p.167).

[241] We define our proof system for $\mathcal{M}$ using natural deduction (➜ p.23).

The judgements are formulae, i.e., terms of type *prop* (➜ p.236). This means that a node ϕ in a derivation tree, as in

$$\frac{\cdots}{\phi}\cdots$$

must be a term of type *prop*. It cannot be a derivability judgement (➜ p.47) or type judgement (➜ p.167) or a term of type, say $prop \to prop$.

Rules for ⇒

$$\dfrac{\begin{array}{c}[\phi]\\ \vdots\\ \psi\end{array}}{\phi \Rightarrow \psi}\Rightarrow\text{-}I \quad \dfrac{\phi \Rightarrow \psi \quad \phi}{\psi}\Rightarrow\text{-}E$$

Just like rules for → (➜ p.31)!

For layout reasons we sometimes swap left and right:

$$\dfrac{\phi \quad \phi \Rightarrow \psi}{\psi}\Rightarrow\text{-}E$$

Note that these rules are Meta-Rules[242].

[242] If we express the rules

$$\dfrac{\begin{array}{c}[\phi]\\ \vdots\\ \psi\end{array}}{\phi \Rightarrow \psi}\Rightarrow\text{-}I \quad \dfrac{\phi \Rightarrow \psi \quad \phi}{\psi}\Rightarrow\text{-}E$$

in our metalanguage we get

$$\bigwedge \phi\ \psi.(\phi \Rightarrow \psi) \Rightarrow \phi \Rightarrow \psi$$

in both cases. Although this metaformula is true, it is not very helpful. We cannot derive anything from it without using the metarules.

One could invent a metametalanguage for expressing derivability in our metalanguage and this process could be repeated ad infinitum. We will, however, not do much metametareasoning and therefore need no metametalanguage to express metarules.

Rules for $\bigwedge$

Meta-universal-quantification is formalized in the style of higher-order abstract syntax (➜ p.216) ($\bigwedge_\sigma : (\sigma \to \mathit{prop}) \to \mathit{prop}$ (➜ p.236)); may write $\bigwedge x.\phi$ as syntactic sugar (➜ p.37) for $\bigwedge_\sigma(\lambda x.\phi)$.

Note: quantification over terms of arbitrary type!

Rules:

$$\frac{\phi\ x}{\bigwedge x.\ \phi\ x}\ \bigwedge\text{-}I^* \qquad \frac{\bigwedge x.\ \phi\ x}{\phi\ b}\ \bigwedge\text{-}E$$

Side condition $*$: x is not free in any open assumption on which ϕx depends.

Just like rules for $\forall$ (➜ p.80).

Rules for α, β-Conversions

Isabelle always rewrites every proposition into its normal form using α, β-conversion in the background. This can be expressed by the rule

$$\frac{\phi}{\phi'}\,conv \quad \text{(where } \phi =_{\alpha\beta} \phi'\text{)}$$

Rules for $\equiv$: Equivalence Relation

Isabelle also adds a meta-equality $\equiv$, whose main purpose is to introduce new definitions. It satisfies the following axioms.

- $\equiv$-*refl*: $\bigwedge t.\ t \equiv t$
- $\equiv$-*subst*: $\bigwedge \phi.\bigwedge s.\bigwedge t.\ s \equiv t \Rightarrow \phi\ s \Rightarrow \phi\ t$
- $\equiv$-*ext*: $\bigwedge f.\bigwedge g.\ (\bigwedge x.\ f\ x \equiv g\ x) \Rightarrow f \equiv g$
- $\equiv$-*I*: $\bigwedge \phi.\bigwedge \psi.\ (\phi \Rightarrow \psi) \Rightarrow (\psi \Rightarrow \phi) \Rightarrow \phi \equiv \psi.$

Additionally a theory can introduce new axioms of the form $\mathit{def}:\ c \equiv t$ where c is a fresh constant symbol (which is then added to Σ) and t a λ-term containing no free variables.

A proof in the Metalogic

To demonstrate the metalogic we prove symmetry. We first instantiate the axiom $\equiv$-*subst*:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bigwedge \phi\ s\ t. s \equiv t \Rightarrow \phi\ s \Rightarrow \phi\ t}\ \equiv\text{-}subst}{\bigwedge s\ t.\ s \equiv t \Rightarrow (\lambda x. x \equiv s)\ s \Rightarrow (\lambda x. x \equiv s)\ t}\ \bigwedge\text{-}E}{\bigwedge s\ t.\ s \equiv t \Rightarrow s \equiv s \Rightarrow t \equiv s}\ conv}{\bigwedge t.\ s \equiv t \Rightarrow s \equiv s \Rightarrow t \equiv s}\ \bigwedge\text{-}E}{s \equiv t \Rightarrow s \equiv s \Rightarrow t \equiv s}\ \bigwedge\text{-}E$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{s \equiv t \Rightarrow s \equiv s \Rightarrow t \equiv s}\,\textstyle\bigwedge\text{-}E \qquad [s \equiv t]^1}{s \equiv s \Rightarrow t \equiv s}\,\Rightarrow\text{-}E \qquad \dfrac{\dfrac{}{\bigwedge t.\ t \equiv t}\,\equiv\text{-}\mathit{refl}}{s \equiv s}\,\textstyle\bigwedge\text{-}E}{t \equiv s}\,\Rightarrow\text{-}E}{s \equiv t \Rightarrow t \equiv s}\,\Rightarrow\text{-}I^1}{\bigwedge t.\ s \equiv t \Rightarrow t \equiv s}\,\textstyle\bigwedge\text{-}I}{\bigwedge s\ t.\ s \equiv t \Rightarrow t \equiv s}\,\textstyle\bigwedge\text{-}I
$$

10.2 Encoding Syntax and Provability

We use FOL (➜ p.60) and its subset propositional logic (➜ p.10) (which we call here *Prop*) as exemplary object logic.

We already know how to encode syntax (➜ p.196).

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 (➜ p.162), the term language (the signature (➜ p.201)) and the proof rules. The thus extended logic will be called $\mathcal{M}_L$.

Encoding Syntax: Review

As before, $i, o \in \mathcal{B}$ (➜ p.211). Previously:

$$\begin{aligned}\Sigma \supseteq\ & \langle \mathit{not} : o \to o, \mathit{and} : o \to o \to o, \mathit{imp} : o \to o \to o,\\ & \mathit{all} : (i \to o\ (\text{➜ p.216})) \to o, \mathit{exists} : (i \to o) \to o\rangle\end{aligned}$$

Two types[243] for truth values: o and prop.

We now need a more concise (sweeter (➜ p.37)) syntax or things will become hopelessly unreadable.

But this is also quite demanding: you should always be able to “unsugar” the syntax.

[243] So we have truth values in the metalogic (type *prop*) and in the object logic (type o). To distinguish them clearly there are two different types for them.

Encoding Syntax Readably

$$\begin{array}{rl}\Sigma \supseteq & \langle \bot : o, \\ & \neg : o \to o, \\ & \wedge, \vee, \to \text{[244]} : o \to o \to o, \\ & \forall, \exists : (i \to o) \to o, \\ & \mathit{true} : o \to \mathit{prop}\rangle. \end{array}$$

- $\to$ is both a constant declared in Σ and the function type arrow (➜ p.163).
- $\wedge, \vee, \to$ will be written infix (➜ p.65), and we may write $\forall x.\phi$ for $\forall(\lambda x.\phi)$, and likewise for $\exists$.
- *true* A[245] is usually written $[\![A]\!]$.

[244] We write

$$\begin{array}{l}\langle \bot : o, \\ \wedge, \vee, \to : o \to o \to o, \\ \forall, \exists : (i \to o) \to o, \\ \mathit{true} : o \to \mathit{prop}\rangle \end{array}$$

as shorthand for

$$\begin{array}{l}\langle \bot : o, \\ \wedge : o \to o \to o, \\ \vee : o \to o \to o, \\ \to : o \to o \to o, \\ \forall : (i \to o) \to o, \\ \exists : (i \to o) \to o \\ \mathit{true} : o \to \mathit{prop}\rangle \end{array}$$

[245] So we have truth values in the metalogic (type *prop*) and in the object logic (type o).

Paulson [Pau89] says: “the meta-formula $[\![A]\!]$ abbreviates *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 $\Rightarrow$-E.
- There is a similar object rule that we call the $\rightarrow$-E rule.
- It is encoded as a meta-axiom that we call the $\rightarrow$-E axiom.

say that $[\![A]\!]$ is a meta-formula that may or may not be derivable in $\mathcal{M}_L$ (➜ p.244), and that this should reflect derivability of A in L (➜ p.249).

In the file `IFOL.thy` in your Isabelle distribution (➜ p.289), you find

```
Trueprop   :: "o => prop"
```

`Trueprop` corresponds to *true*.

Encoding of the Rules of Propositional Logic

$$
\begin{array}{ll}
\bigwedge AB.[\![A]\!] \Rightarrow ([\![B]\!] \Rightarrow [\![A \wedge B]\!]) & (\wedge\text{-}I) \\
\bigwedge AB.[\![A \wedge B]\!] \Rightarrow [\![A]\!] & (\wedge\text{-}EL) \\
\bigwedge AB.[\![A \wedge B]\!] \Rightarrow [\![B]\!] & (\wedge\text{-}ER) \\
\bigwedge AB.[\![A]\!] \Rightarrow [\![A \vee B]\!] & (\vee\text{-}IL) \\
\bigwedge AB.[\![B]\!] \Rightarrow [\![A \vee B]\!] & (\vee\text{-}IR) \\
\begin{array}{l} \bigwedge ABC.[\![A \vee B]\!] \Rightarrow \\ ([\![A]\!] \Rightarrow [\![C]\!]) \Rightarrow ([\![B]\!] \Rightarrow [\![C]\!]) \Rightarrow [\![C]\!] \end{array} & (\vee\text{-}E) \\
\bigwedge AB.([\![A]\!] \Rightarrow [\![B]\!]) \Rightarrow [\![A \rightarrow B]\!] & (\rightarrow\text{-}I) \\
\bigwedge AB.[\![A \rightarrow B]\!] \Rightarrow [\![A]\!] \Rightarrow [\![B]\!] & (\rightarrow\text{-}E) \\
\bigwedge A.[\![\bot]\!] \Rightarrow [\![A]\!] & (\bot\text{-}E)
\end{array}
$$

Faithful Metalogics

For any object logic L, we define:

- $\mathcal{M}_L$ is sound for L if, for every proof of $[\![A_1]\!] \Rightarrow \ldots \Rightarrow [\![A_m]\!] \Rightarrow [\![B]\!]$ in $\mathcal{M}_L$, there is a proof of B from assumptions $A_1, \ldots, A_m$ in L.

- $\mathcal{M}_L$ is complete for L if, for every proof of B from assumptions $A_1, \ldots, A_m$ in L, there is a proof of $[\![A_1]\!] \Rightarrow \ldots \Rightarrow [\![A_m]\!] \Rightarrow [\![B]\!]$ 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* (➜ p.244).

An Example Proof

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\begin{array}{r}\bigwedge AB.(\llbracket A\rrbracket \Rightarrow \llbracket B\rrbracket)\\ \Rightarrow \llbracket A\to B\rrbracket\end{array}}\ \to\text{-}I\ (\text{➜ p.248})
}{\begin{array}{r}\bigwedge B.(\llbracket P\wedge Q\rrbracket \Rightarrow \llbracket B\rrbracket)\\ \Rightarrow \llbracket P\wedge Q\to B\rrbracket\end{array}}\ \bigwedge\text{-}E\ (\text{➜ p.239})
}{\begin{array}{c}(\llbracket P\wedge Q\rrbracket \Rightarrow \llbracket P\rrbracket)\\ \Rightarrow \llbracket P\wedge Q\to P\rrbracket\end{array}}\ \bigwedge\text{-}E\ (\text{➜ p.239})
\qquad
\dfrac{
\dfrac{
\dfrac{}{\begin{array}{r}\bigwedge AB.\llbracket A\wedge B\rrbracket\\ \Rightarrow \llbracket A\rrbracket\end{array}}\ \wedge\text{-}EL\ (\text{➜ p.248})
}{\begin{array}{r}\bigwedge B.\llbracket P\wedge B\rrbracket\\ \Rightarrow \llbracket P\rrbracket\end{array}}\ \bigwedge\text{-}E\ (\text{➜ p.239})
}{\llbracket P\wedge Q\rrbracket \Rightarrow \llbracket P\rrbracket}\ \bigwedge\text{-}E\ (\text{➜ p.239})
}{\llbracket P\wedge Q\to P\rrbracket}\ \Rightarrow\text{-}E\ (\text{➜ p.238})
$$

$\wedge$-*EL* (➜ p.248) $\to$-*I* (➜ p.248) are not object rules but (➜ p.247) meta-axioms!

10.3 Converting Proofs to Metaproofs

An application of a proof rule corresponds to applying $\bigwedge$-E and $\Rightarrow$-E in the metaproof, e.g.,

$$\dfrac{P \wedge Q}{Q}\ \wedge\text{-}EL \qquad \dfrac{\dfrac{\dfrac{\bigwedge A\ B.\, [\![A \wedge B]\!] \Rightarrow [\![B]\!]}{\bigwedge B.\, [\![P \wedge B]\!] \Rightarrow [\![B]\!]}\ \bigwedge\text{-}E}{[\![P \wedge Q]\!] \Rightarrow [\![Q]\!]}\ \bigwedge\text{-}E \qquad [\![P \wedge Q]\!]}{[\![Q]\!]}\ \Rightarrow\text{-}E$$

Natural Deduction Rules

For natural deduction style proofs $\Rightarrow$-I is used if the proof rule eliminates assumptions, e.g.,

$$\dfrac{\begin{matrix}[P]\\ \vdots \\ Q\end{matrix}}{P \to Q}\ {\to}\text{-}I$$

converts to the metaproof

$$\dfrac{\dfrac{\dfrac{\bigwedge A\ B.\,(\llbracket A\rrbracket \Rightarrow \llbracket B\rrbracket) \Rightarrow \llbracket A\to B\rrbracket}{\bigwedge B.\,(\llbracket P\rrbracket \Rightarrow \llbracket B\rrbracket)\Rightarrow \llbracket P\to B\rrbracket}\ \bigwedge\text{-}E}{(\llbracket P\rrbracket \Rightarrow \llbracket Q\rrbracket)\Rightarrow\llbracket P\to Q\rrbracket}\ \bigwedge\text{-}E \qquad \dfrac{\begin{matrix}[\llbracket P\rrbracket]\\ \vdots\\ \llbracket Q\rrbracket\end{matrix}}{\llbracket P\rrbracket\Rightarrow\llbracket Q\rrbracket}\ {\Rightarrow}\text{-}I}{\llbracket P\to Q\rrbracket}\ {\Rightarrow}\text{-}E$$

10.4 Quantification

We add the following meta-axioms to obtain $\mathcal{M}_{\mathrm{FOL}}$ (➜ p.244):

$$
\begin{array}{ll}
\bigwedge F.(\bigwedge x.[\![F\,x]\!]) \Rightarrow [\![\forall x.F\,x]\!] & (\forall\text{-}I) \\
\bigwedge Fy.[\![\forall x.F\,x]\!] \Rightarrow [\![F\,y]\!] & (\forall\text{-}E) \\
\bigwedge Fy.[\![F\,y]\!] \Rightarrow [\![\exists x.F\,x]\!] & (\exists\text{-}I) \\
\bigwedge FB.[\![\exists x.F\,x]\!] \Rightarrow (\bigwedge x.[\![F\,x]\!] \Rightarrow [\![B]\!]) \Rightarrow [\![B]\!] & (\exists\text{-}E)
\end{array}
$$

Similarly as for *Prop* (➜ p.249), one can show that $\mathcal{M}_{\mathrm{FOL}}$ is faithful for FOL.

Side condition checking is shifted to the meta-level (➜ p.259).

Proof of $(\forall z.G\,z) \to (\forall z.G\,z \lor H\,z)$

$$\dfrac{\dfrac{\dfrac{\dfrac{[\forall z.G\,z]^1}{G\,z}\,\forall\text{-}E}{G\,z \lor H\,z}\,\lor\text{-}IL}{\forall z.G\,z \lor H\,z}\,\forall\text{-}I}{(\forall z.G\,z) \to (\forall z.G\,z \lor H\,z)}\,\to\text{-}I^1$$

How is it proved using metalogic?

Proof of $(\forall z. G\,z) \to (\forall z. G\,z \vee H\,z)$ (1)

We do the proof top-down. First $\forall$-E and $\vee$-IL:

$$\dfrac{\dfrac{\bigwedge F y. [\![\forall x. F\,x]\!] \Rightarrow [\![F\,y]\!]}{[\![\forall z. G\,z \Rightarrow G\,z]\!]}\,\bigwedge\text{-}E \qquad [\![\forall z. G\,z]\!]}{[\![G\,z]\!]}\,\Rightarrow\text{-}E$$

$$\dfrac{\dfrac{\bigwedge F G. [\![F]\!] \Rightarrow [\![F \vee G]\!]}{[\![G\,z]\!] \Rightarrow [\![G\,z \vee H\,z]\!]}\,\bigwedge\text{-}E \qquad \dfrac{\vdots}{[\![G\,z]\!]}\,\Rightarrow\text{-}E}{[\![G\,z \vee H\,z]\!]}\,\Rightarrow\text{-}E$$

Proof of $(\forall z.G\,z) \rightarrow (\forall z.G\,z \vee H\,z)$ **(2)**

Now $\forall$-E. Note that the proof of $[\![G\,z \vee H\,z]\!]$ does not have z in any free assumption.

$$\dfrac{\dfrac{\dfrac{\bigwedge F.(\bigwedge x.[\![F\,x]\!]) \Rightarrow [\![\forall x.F\,x]\!]}{\bigwedge F.(\bigwedge z.[\![F\,z]\!]) \Rightarrow [\![\forall z.F\,z]\!]}\,\mathit{conv}}{(\bigwedge z.[\![Gz \vee Hz]\!]) \Rightarrow [\![\forall z.Gz \vee Hz]\!]}\,\bigwedge\text{-}E \qquad \dfrac{\dfrac{\vdots}{[\![G\,z \vee H\,z]\!]}\Rightarrow\text{-}E}{\bigwedge z.[\![G\,z \vee H\,z]\!]}\,\bigwedge\text{-}I}{[\![\forall z.G\,z \vee H\,z]\!]}\Rightarrow\text{-}E$$

The *conv*-step uses α-renaming.

Strictly speaking, another *conv*-step after $\bigwedge$-E is necessary:

$$\dfrac{\dfrac{\bigwedge F.(\bigwedge z.[\![F\,z]\!]) \Rightarrow [\![\forall z.F\,z]\!]}{(\bigwedge z.[\![(\lambda x.G\,x \vee H\,x)z]\!]) \Rightarrow [\![\forall z.(\lambda x.G\,x \vee H\,x)z]\!]}\,\bigwedge\text{-}E}{(\bigwedge z.[\![G\,z \vee H\,z]\!]) \Rightarrow [\![\forall z.G\,z \vee H\,z]\!]}\,\mathit{conv}$$

Proof of $(\forall z.G\,z) \to (\forall z.G\,z \lor H\,z)$ (3)

In the final step we use $\Rightarrow$-I and $\to$-I.

$$\dfrac{\dfrac{\bigwedge A\,B.([\![A]\!] \Rightarrow [\![B]\!]) \Rightarrow [\![A \to B]\!]}{\begin{array}{c}([\![\forall z.G\,z]\!] \Rightarrow [\![\forall z.G\,z \lor H\,z]\!])\\ \Rightarrow [\![(\forall z.G\,z) \to (\forall z.G\,z \lor H\,z)]\!]\end{array}}\bigwedge\text{-}E \qquad \dfrac{\begin{array}{c}[[\![\forall z.G\,z]\!]]\\ \vdots\\ [\![\forall z.G\,z \lor H\,z]\!]\end{array}}{[\![\forall z.G\,z]\!] \Rightarrow [\![\forall z.G\,z \lor H\,z]\!]}\Rightarrow\text{-}I}{[\![(\forall z.G\,z) \to (\forall z.G\,z \lor H\,z)]\!]}\Rightarrow\text{-}E$$

Checking Side Conditions

To demonstrate how side conditions are checked, we show a proof attempt that fails due to a side condition.

The formula $[\![\neg p(x)]\!] \Rightarrow [\![\exists x.p(x)]\!] \Rightarrow [\![\bot]\!]$ should not be provable.

Why? Because, $p(x)$ may be false for some x and true for another x.

We “prove” $\llbracket\neg p(x)\rrbracket \Rightarrow \llbracket\exists x.p(x)\rrbracket \Rightarrow \llbracket\bot\rrbracket$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\exists\text{-}E}{\cdots}\bigwedge\text{-}E \quad [\llbracket\exists x.p(x)\rrbracket]^2}{\cdots}\Rightarrow\text{-}E \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg\text{-}E}{\cdots}\bigwedge\text{-}E \quad [\llbracket\neg p(x)\rrbracket]^1}{\cdots}\Rightarrow\text{-}E \quad [\llbracket p(x)\rrbracket]^3}{\llbracket\bot\rrbracket}\Rightarrow\text{-}E}{\llbracket p(x)\rrbracket\Rightarrow\llbracket\bot\rrbracket}\Rightarrow\text{-}I^3}{\bigwedge x.\llbracket p(x)\rrbracket\Rightarrow\llbracket\bot\rrbracket}\bigwedge\text{-}I}{\llbracket\bot\rrbracket}\Rightarrow\text{-}E}{\llbracket\exists x.p(x)\rrbracket\Rightarrow\llbracket\bot\rrbracket}\Rightarrow\text{-}I^2}{\llbracket\neg p(x)\rrbracket\Rightarrow\llbracket\exists x.p(x)\rrbracket\Rightarrow\llbracket\bot\rrbracket}\Rightarrow\text{-}I^1$$

Where is the problem?
Side condition of $\bigwedge$-I is not satisfied!

10.5 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 (➜ p.162) is very powerful for expressing syntax and syntactic manipulations (→ substitution). $\mathcal{M}$ must be extended by appropriate signature (➜ p.165) for an object logic.
- Rules of the object logic can be encoded and added to $\mathcal{M}$[246] as axioms.

[246] In some course on propositional logic, you may have learned that the connective $\rightarrow$ is not really necessary since $A \rightarrow B$ is equivalent to $\neg A \vee B$. Likewise, we considered $\neg A$ as syntactic sugar (➜ p.16) for $A \rightarrow \bot$.

Therefore, when we introduce a logic $\mathcal{M}$ that is so extremely simple as far as the number of logical symbols (➜ p.236) is concerned (just $\Rightarrow$, $\equiv$, $\bigwedge$), 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 (➜ p.10) or FOL (➜ p.60) in $\mathcal{M}$, we must add their rules as axioms.

Later (➜ p.320), 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.231): shared support and high degree of confidence.

11 Resolution

Three Sections on Deduction Techniques

After encoding syntax (➜ p.196) and encoding of proofs (➜ p.**??**), the next topics are automated proof methods.

- Resolution (➜ p.263)
- Proof search (➜ p.274)
- Term rewriting (➜ p.299)

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 constructing proofs by applying proof rules (metatheorems).

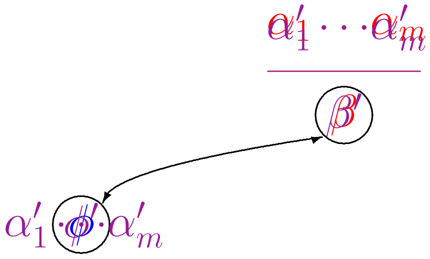
It involves unifying (➜ p.192) a certain part of the current goal (state) with a certain part of a rule, and replacing that part of the current goal.

We have already explained this in the labs and you have been working with it all the time, but now we want to understand it more thoroughly.

We look at several variants of resolution.

Note: The following slides on Resolution rely heavily on animation features. It is therefore advised that you study them on a screen in slide or screen-notes form.

Resolution (`rule`, as in Prolog[247])


$\alpha'_1 \cdots \alpha'_m$
β'
$\alpha'_1 \cdot \phi' \cdot \alpha'_m$

ϕ is the current subgoal. Isabelle displays

goal ... (1 **subgoal**) :
1. ϕ

$[\![\alpha_1; \ldots ; \alpha_m]\!] \Longrightarrow \beta$ is rule.

Simple scenario where ϕ has no premises[248]. Now β must be unifiable with selected subgoal ϕ.

We apply the unifier ($'$[249])

We replace ϕ' by the premises of the rule.

[247] Prolog is a logic programming language [Apt97].

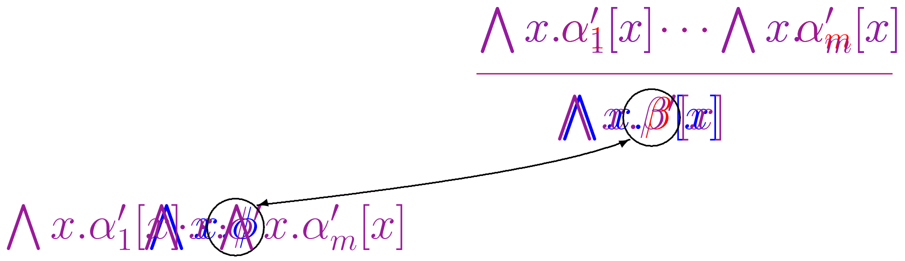
The computation mechanism of Prolog is resolution of a current goal (corresponding to our $\phi_1, \ldots, \phi_n$) with a Horn clause (corresponding to our $[\![\alpha_1; \ldots ; \alpha_m]\!] \Longrightarrow \beta$).

[248] ϕ is the current subgoal. With Isar proof scripts one usually only has one active subgoal, namely the lemma that is given by the `have` or `lemma` command preceeding the `proof` statement.

We assume here that ϕ is a formula, i.e., it contains no $\Longrightarrow$ (metalevel implication).

[249] In all illustrations that follow, we use $'$ to suggest the application of the appropriate unifier.

Resolution (with Lifting over Parameters)



$$\frac{\bigwedge x.\alpha'_1[x] \cdots \bigwedge x.\alpha'_m[x]}{\bigwedge x.\beta'[x]}$$

$\bigwedge x.\alpha'_1[x] \cdots \bigwedge x.\phi' \cdots \bigwedge x.\alpha'_m[x]$

Now suppose the subgoal is preceded by $\bigwedge$ (metalevel universal quantifier[250]).

Rule is lifted[251] over x: Apply $[?X \leftarrow ?X(x)]$.

As before, β must be unifiable with ϕ; apply the unifier.

We replace ϕ' by the premises of the rule. $\alpha'_1, \ldots, \alpha'_m$ are preceded by $\bigwedge x$.

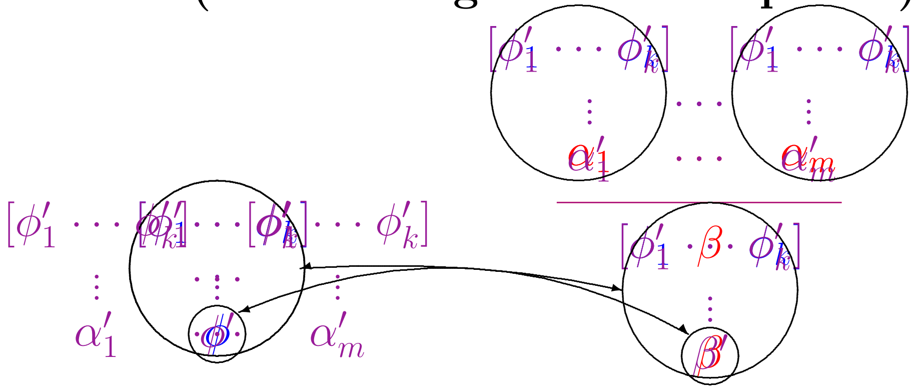
[250] $\bigwedge$ is the metalevel universal quantification (also written !!). If a goal is preceded by $\bigwedge x$, this means that Isabelle must be able to prove the subgoal in a way which is independent from x, i.e., without instantiating x.

[251] The metavariables of the rule are made dependent on x. That is to say, each metavariable $?X$ is replaced by a $?X(x)$. You may also say that $?X$ is now a Skolem function of x.

This process is called lifting the rule over the parameter x.

We denote by $\rho[x]$ the result of lifting ρ over x.

Resolution (with Lifting over Assumptions)



Now, suppose the subgoal has assumptions $\phi_1, \ldots, \phi_k$.

As before, we have a rule. Here, β is (hopefully) unifiable with ϕ, but β is not[252] unifiable with the entire subgoal.

Rule must be lifted over assumptions[253]. No unification so far!

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

[252] The subgoal is $[\![\phi_1, \ldots, \phi_k]\!] \Longrightarrow \phi$ where $\phi_1, \ldots, \phi_k, \phi$ are object-level formulae. So the selected subgoal is not an object-level formula, but it has $\Longrightarrow$ (➜ p.186) 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 $\Longrightarrow$ as"top-level constructor'.

[253] Each premise of the rule, as well as the conclusion of the rule, are preceded by the assumptions $[\![\phi_1, \ldots, \phi_k]\!]$ of the current subgoals. Actually, the rule

$$\dfrac{\begin{array}{ccc}[\phi_1 \cdots \phi_k] & & [\phi_1 \cdots \phi_k]\\ \vdots & \cdots & \vdots\\ \alpha_1 & \cdots & \alpha_m\end{array}}{\begin{array}{c}[\phi_1 \cdots \phi_k]\\ \vdots\\ \beta\end{array}}$$

may look different from any rules you have seen so far, but unifiable[254].

Non-trivially[255], β must be unifiable with ϕ.

We apply the unifier.

We replace the subgoal.

---

it can be formally derived from the rule:

$$\frac{\alpha_1 \quad \cdots \quad \alpha_m}{\beta}$$

The derived rule should be read as: If for all $j \in \{1, \ldots, m\}$, we can derive α_j from $\phi_1, \ldots, \phi_k$, then we can derive β from $\phi_1, \ldots, \phi_k$.

[254] Still assuming that ϕ and β are unifiable.

[255] Both the subgoal and the conclusion of the lifted rule are preceded by assumptions $\phi_1, \ldots, \phi_k$. Hence the assumption list of the subgoal and the assumption list of the rule are trivially unifiable since they are identical.

Rule Premises Containing $\Longrightarrow$

$$\begin{array}{c} \qquad [\phi'_1 \cdots \phi'_k]; \gamma'_1 \cdots \gamma'_l] \\ \vdots \\ \phi'_1 \cdots \llbracket \gamma_1; \ldots ; \rlap{δ'}\rlap{α'_j}\gamma_l \rrbracket \Longrightarrow \delta \cdots \phi'_n \\ \hline \psi' \end{array}$$

What if some α'_j has the form $\llbracket \gamma_1; \ldots ; \gamma_l \rrbracket \Longrightarrow \delta$?

Is this what we get?

Well, we write $\vdots$ for $\Longrightarrow$, and use $A \Longrightarrow B \Longrightarrow C \equiv \llbracket A; B \rrbracket \Longrightarrow C$[256].

[256] Generally, Isabelle makes no distinction between

$$\llbracket \psi_1; \ldots ; \psi_n \rrbracket \Longrightarrow \llbracket \mu_1; \ldots ; \mu_k \rrbracket \Longrightarrow \phi$$

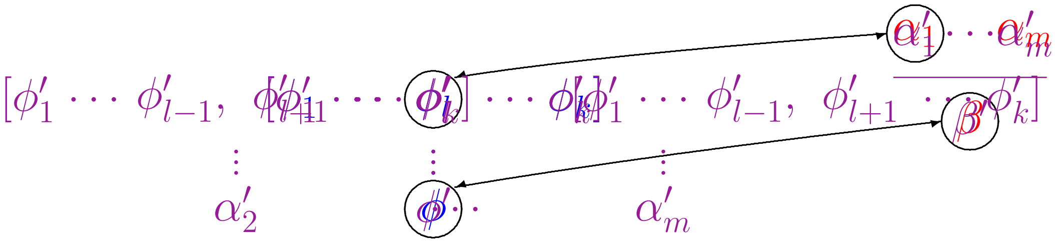
and

$$\llbracket \psi_1; \ldots ; \psi_n; \mu_1; \ldots ; \mu_k \rrbracket \Longrightarrow \phi$$

and displays the second form. Semantically, this corresponds to the equivalence of $A_1 \wedge \ldots \wedge A_n \rightarrow B$ and $A_1 \rightarrow \ldots \rightarrow A_n \rightarrow B$.

We have seen this in the exercises.

Elimination-Resolution


$[\phi'_1 \cdots \phi'_{l-1}, \phi'_{l+1} \cdots \phi'_k]$
α'_2
ϕ'
ϕ
α'_m
α_1
$\alpha'_1 \cdots \alpha'_m$
β

Same scenario as before[257], but now β must be unifiable with ϕ, and α_1 must be unifiable with ϕ_l, for some l.

Apply the unifier.

We replace ϕ' by the premises of the rule except the first[258]. $\alpha'_2, \ldots, \alpha'_m$ inherit the assumptions of ϕ', except ϕ'_l.

[257] So the scenario looks as for resolution with lifting over assumptions (➜ p.268). However, this time we do not show the lifting over assumptions in our animation.

[258] Elimination-resolution is used to eliminate a connective in the premises.

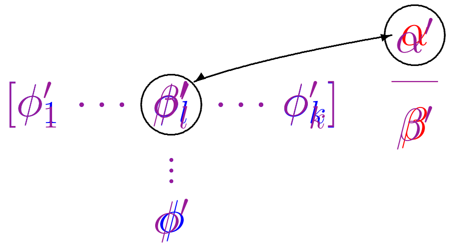
For example, if the current goal is

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

and the rule is

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

Destruct-Resolution



Simple rule, and α must be unifiable with ϕ_l, for some l. We apply the unifier.
We replace premise[259] ϕ'_l with the conclusion of the rule.

11.1 Summary on Resolution

- Build proof resembling sequent style notation (➜ p.47);
- technically: replace goals with rule premises, or goal premises with rule conclusions;

then the result of elimination resolution is

$$\frac{\begin{matrix}[A;B]\\ \vdots \\ B\end{matrix}}{A\wedge B\to B}$$

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.

[259] Destruct-resolution is used to eliminate a connective in the premises. The difference compared to elimination-resolution (➜ p.271) can be seen in the following example. Unlike elimination-resolution, destruct-resolution "throws information away".

- metavariables and unification (➜ p.192) to obtain appropriate instance of rule, delay commitments;
- lifting over parameters (➜ p.267) and assumptions (➜ p.268);
- various techniques to manipulate premises or conclusions, as convenient: `rule` (➜ p.266), `erule` (➜ p.271), `drule` (➜ p.272).

For example, if the current goal is

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

and the rule is

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

then the result of destruct-resolution is

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

If we had instead used rule

$$\frac{P \wedge Q}{P}\ \texttt{conjunct1}$$

the result would have been

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

and we would be stuck. We accidentally “threw away” the assumption B.

Outline of this Part

- Classifying rules
- Proof search (➜ p.291) and backtracking
- Proof procedures (➜ p.296)

12.1 Classifying Rules

In your early Isabelle exercises, you only used backward reasoning (**`rule`**) (➜ p.266). You experienced that some rules can be applied blindly most of the time, e.g. $\rightarrow$-I (➜ p.48) or $\wedge$-I (➜ p.48). Others involve "guessing", e.g. $\wedge$-EL (➜ p.48) or $\wedge$-ER (➜ p.48) (you do not know which to apply to deal with a $\wedge$ in the premises).

Later on you learned about **`erule`** (➜ p.271) combined with specially tailored rules (they have an "E" in their name). That helps reduce the "guessing".

In the following we will explain some underlying principles of this using sequent style notation (➜ p.47).

Review: Natural Deduction

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

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

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

Classification into safe and unsafe

The introduction rule

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

can be applied blindly: if you can proof $A \wedge B$ you can always proof A and then B.

The introduction rule

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

is not safe to apply blindly. You may not be able to show A even though showing $A \vee B$ is possible.

Safe Disjunction Rule

For some rules it is possible to find safe alternatives

In classical logic, we can instead use a safe introduction rule:

$$\dfrac{\begin{array}{c}[\neg B]\\ \vdots \\ A\end{array}}{A \vee B}\,\vee\text{-}IC$$

Elim and Destruction Rules

In natural deduction we have two kinds of elimination rules, e.g.

$$\frac{A \to B \quad A}{B}\,{\to}\text{-}E \qquad \frac{A \lor B \quad \begin{array}{c}[A]\\ \vdots\\ C\end{array} \quad \begin{array}{c}[B]\\ \vdots\\ C\end{array}}{C}\,{\lor}\text{-}E$$

In Isabelle the first is called a destruction rule and would be applied using `drule`. The second is a elimination rule applied using `erule`.

Making (safe) Elimination Rules

One can easily make an elimination rule from a destruction rule:

$$\dfrac{A \to B \quad A \quad \begin{matrix}[B]\\ \vdots \\ C\end{matrix}}{C}\;\to\text{-}E$$

We can combine $\wedge$-EL and $\wedge$-ER to a safe elimination rule:

$$\dfrac{A \wedge B \qquad \begin{matrix}[A, B]\\ \vdots \\ C\end{matrix}}{C}\;\wedge\text{-}E$$

Safe Rules for Propositional Logic

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

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

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \to B}\ \to\text{-}I \qquad \frac{A \to B \quad \begin{array}{c}[\neg A]\\ \vdots \\ C\end{array}\begin{array}{c}[B]\\ \vdots \\ C\end{array}}{C}\ \to\text{-}EC \qquad \frac{\bot}{A}\ \bot\text{-}E$$

A Proof by Blind Rule Application

Lets proof $(P \wedge Q) \rightarrow R \rightarrow Q$. Only applicable rule is $\rightarrow$-I.
New subgoal: $P \wedge Q \Rightarrow R \rightarrow Q$.

Applying $\rightarrow$-I again yields
new subgoal: $P \wedge Q \Rightarrow R \Rightarrow Q$.

Applicable is only $\wedge$-E.
New subgoal: $P \Rightarrow Q \Rightarrow R \Rightarrow Q$.

The proof can now be completed by the assumption rule (➜ p.48).

Safe and Unsafe Rules

Combined tactics (➜ p.297) rely on classification of rules, maintained in Isabelle (➜ p.293) data structure `claset`[260]. New theorems can be added to these sets by giving them the attribute `intro` or `elim`.

Class:	To add use attribute:
Safe introduction rules	`intro`!
Safe elimination rules	`elim`!
Unsafe introduction rules	`intro`
Unsafe elimination rules	`elim`

[260] `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.

Adapting Rules for Automated Proof Search

As seen for $\wedge$-E (➜ p.**??**), rules must be suitably adapted in order to be useful in automated proof search. Another example:

Goal: $(P \to Q) \vee (Q \to P)$

Applying rule $\vee$-IC yields

Goal: $\neg(P \to Q) \Rightarrow (Q \to P)$

Applying rule $\to$-I yields

Goal: $\neg(P \to Q) \Rightarrow Q \Rightarrow P$

Applying rule $\to$-$swapE$ yields

Goal: $P \Rightarrow \neg Q \Rightarrow Q \Rightarrow P$

Rule $\rightarrow$-*swapE*

The rule $\rightarrow$-*swapE*[261] is

$$\dfrac{\neg(A \rightarrow B) \qquad \begin{array}{c}[A, \neg C] \\ \vdots \\ B\end{array}}{C}\ \rightarrow\text{-}swapE$$

It is derived from $\rightarrow$-*I* and *swap*. The elimination rule *swap* allows to bring a negated assumption to the rhs:

$$\dfrac{\neg A \qquad \begin{array}{c}[\neg C] \\ \vdots \\ A\end{array}}{C}\ swap$$

[261] The rule $\rightarrow$-*swapE* is

$$\dfrac{A, \neg C, \Gamma \vdash B}{\neg(A \rightarrow B), \Gamma \vdash C}\ \rightarrow\text{-}swapE$$

To derive it you need classical (➜ p.40) 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 (➜ p.**??**). For dealing with negated formulas in the premises of the current subgoal, introduction rules are combined with `swap` using `erule`.

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].

Handling Quantifiers

Can derive[262] $\forall$-E' ($\equiv$ `allE`[263]) using $\forall$-E ($\equiv$ `spec`):

$$\dfrac{\forall x.A(x) \qquad \begin{matrix}[A(x), \forall x.A(x)]\\ \vdots \\ B\end{matrix}}{B}\;\forall\text{-}E'\forall\text{-}dupE$$

This is effective for getting rid of a $\forall$ in the premises.

Problem: $\forall x.A(x)$ may still be needed (unsafe rule).

[262] You should do it in Isabelle. The rule is:

$$[\![\texttt{ALL}\ x.\ P(x);\ P(x) \Longrightarrow R]\!] \Longrightarrow R$$

[263] As you may have noticed earlier, there is a confusion between the names of proof rules as we present them for the theory and the names used in Isabelle. For example, rule $\rightarrow$-E is called `mp` in Isabelle. This confusion concerns elimination rules.

There is however a good reason for these choices. In traditional presentations of logic, one sets up the simplest possible elimination rules for the connectives which naturally arise from the meaning of those connectives. This is what we have done as well. However, as we see in this lecture, these rules cannot be applied blindly and are thus not very suitable for automation. Therefore, combined tactics (➜ p.297) in Isabelle use derived rules such as $\wedge$-E (➜ p.**??**) (called `conjE` in Isabelle).

Solution: Introduce duplicating[264] rules. Turns search infinite[265]!

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.

[264] You should recall that elimination rules are used in combination with `erule` (➜ p.271). Using `allE` will eliminate the quantifier.

You should try a proof of the formula $(\forall x.P(x)) \rightarrow (P(a) \wedge P(b))$ in Isabelle to convince yourself that this is a problem since the quantified formula $\forall x.P(x)$ is needed twice as an assumption, with two different instantiations of x.

The duplicating rule ∀-*dupE* has the effect that the universally quantified formula will still remain as an assumption.

[265] Given only the rules so far (in combination with the appropriate tactics, `rule` and `erule` (➜ p.263), and swapping (➜ p.**??**)), excluding ∀-*dupE*, the proof search

Check out `allE` and `all_dupE` in `IFOL_lemmas.ML`[266]!

would be finite.

The rule ∀-*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.

[266] These files should be contained in your Isabelle (➜ p.293) distribution. Or, if you only have an Isabelle executable, you can find the sources here:

`http://isabelle.in.tum.de/library/`

Side question: What is the difference[267] to $\exists$-E[268]?

[267] The difference between

$$\dfrac{\exists x.A(x) \qquad \begin{matrix}[A(x)]\\ \vdots \\ B\end{matrix}}{B}\ \exists\text{-}E$$

and

$$\dfrac{\forall x.A(x) \qquad \begin{matrix}[A(x)]\\ \vdots \\ B\end{matrix}}{B}\ \forall\text{-}E'$$

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 (➜ p.290).

[268] The rule

$$\dfrac{\exists x.A(x) \qquad \begin{matrix}[A(x)]\\ \vdots \\ B\end{matrix}}{B}\ \exists\text{-}E$$

12.2 Proof Search and Backtracking

- Need for more automation[269]
- Some aspects in proof construction are non-deterministic:
  - unification: which unifier (➜ p.192) to choose?

---

was derived previously (➜ p.93) (but in Isabelle, it is a basic rule in `IFOL.ML`). It is

$$[\![\texttt{ALL}\ x.\ P(x);\ !!x.\ P(x) \Longrightarrow R]\!] \Longrightarrow R$$

Note that the rule **`allE`** (➜ p.287) ($\forall$-E') is

$$!!x.[\![\texttt{ALL}\ x.\ P(x);\ P(x) \Longrightarrow R]\!] \Longrightarrow R$$

The difference is that the former rule contains a nested 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'.

[269] We have seen in the exercises that doing a proof step by step is very tedious and often involves difficult guessing or alternatively, backtracking. We cannot hope to prove anything about realistic systems if proving simple theorems is so tedious.

Efficiency considerations are important for automation.

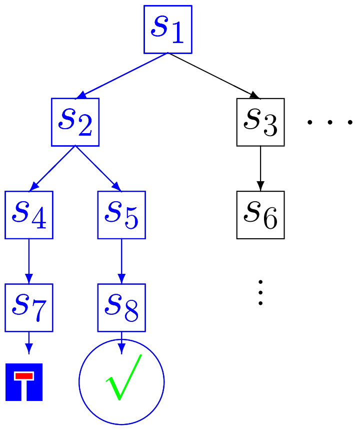
- resolution: where[270] to apply a rule (which 'subgoal')?
- which rule to apply?

The non-determinacy in proof search obviously leads to inefficiencies as many possibilities have to be explored.

[270] We have seen in the exercises (and also in the lecture (➜ p.263)) that one can choose the subgoal to which one wants to apply a rule.

Organizing Proof Search Conceptually

Organize proof search as a tree[271] of theorems[272] (**thm**'s).



- Applications of proof rules move us along leftmost path.
- Using **undo**();[273] moves us upwards (previous proof state).
- Using **back**(); moves us (up and) right (alternative successors[274] due to different unifiers (➜ p.194)).

[271] We have seen in the previous lecture (➜ p.263) 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.

[272] Technically, a proof state is an Isabelle theorem, (**thm**), i.e. something which Isabelle regards as true.

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

[274] Note that when there are no more successors (you cannot go right) anymore, **back**(); will go to the previous proof state, i.e., go up 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 this tree can only be built on-demand:

- Explicit tree representation[275] expensive in time and space.
- Even worse: Branching of the tree is infinite in general (HO-unification (➜ p.192)).

[275] Obviously, an infinite tree cannot be represented explicitly. But even if the tree is finite, it is generally expensive to represent it explicitly. In particular, the tree may contain many failing branches and only few successful ones, which begs the question if representing the unsuccessful branches cannot be avoided somehow.

Organizing Proof Search Operationally

Organize proof search as a function on theorems[276] (`thm`'s)

$$\texttt{type tactic} = \texttt{thm} \to \texttt{thm seq}$$

where `seq`[277] is the type constructor for infinite lists.

This allows us to have tacticals[278]:

- `THEN`
- `ORELSE`
- `REPEAT`
- ...

[276] This way of understanding and organizing proof search is 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 (➜ p.293).

[277] For any type τ, the type τ `seq` (recall the notation (➜ p.179)) is the type of (possibly) infinite lists of elements of type τ. This is of course an abstract datatype. There should be functions to return the head and the tail of such an infinite list.

An abstract datatype is a type whose terms cannot be represented explicitly and accessed directly, but only via certain functions for that type.

[278]

- `THEN`

12.3 Proof Procedures (Simplified)

Tactics in Isabelle (➜ p.293) are performed in order[279]:

1. `REPEAT` (`rule` *safe_I_rules* `ORELSE` `erule` *safe_E_rules*)
2. canonize: propagate "$x = t$" throughout subgoal
3. `rule` *unsafe_I_rules* `ORELSE` `erule` *unsafe_E_rules*
4. `assumption`

There are variants of this. We do not study them in detail, we just use them ...

- `ORELSE`
- `REPEAT`
- ...

are called tacticals.

Tacticals are operations on tactics. They play an important role in automating proofs in Isabelle (➜ p.293). The most basic tacticals are `THEN` and `ORELSE`. Both of those tacticals are of type `tactic` $*$ `tactic` $\rightarrow$ `tactic` and are written infix: tac_1 `THEN` tac_2 applies tac_1 and then tac_2, while tac_1 `ORELSE` tac_2 applies tac_1 if possible and otherwise applies tac_2 [Pau05, Ch. 4].

[279] Tactics in Isabelle (➜ p.293) are performed in order:

1. `REPEAT` (`rule` *safe_I_rules* `ORELSE` `erule` *safe_E_rules*);
2. canonize: propagate "$x = t$" ... throughout subgoal;
3. `rule` *unsafe_I_rules* `ORELSE` `erule` *unsafe_E_rules*;

Combined Proof Search Tactics (➜ p.293)

- `step_tac` : `claset` → `int` → `tactic`
  (just safe steps)
- `fast_tac` : `claset` → `int` → `tactic`
  (safe and unsafe steps in depth-first stategy)
- `best_tac` : `claset` → `int` → `tactic`
  (safe and unsafe steps in breadth-first stategy)
- `slow_tac` : `claset` → `int` → `tactic`
  (like `fast_tac`, but with backtracking `assumption`'s)
- `blast_tac` : `claset` → `int` → `tactic`
  (like `fast_tac`, but often more powerful)

---

4. `assumption`.

One elementary proof step consists of trying a safe introduction rule with `rule` (➜ p.263), or, if that is not possible, a safe elimination rule with `erule` (➜ p.271). 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 wil be replaced by t.

Then Isabelle will try one application of an unsafe introduction rule with `rule` (➜ p.263), or, if that is not possible, an unsafe elimination rule with `erule` (➜ p.271).

Finally, she will use `assumption`. Note that `assumption` is unsafe. In general, there are several premises in a subgoal and `assumption` may unify the conclusion of the subgoal with the wrong premise.

12.4 Summary on Automated Proof Search

- Proof search can be organized as a tree of theorems (➜ p.293).
- Calculi can be set up to facilitate proof search (although this must be done by specialists).
- Combined with search strategies (➜ p.297), powerful automatic procedures arise. Can prove well-known hard problems such as $((\exists y.\forall x.J(y,x) \vee \neg J(x,x)) \to \neg(\forall x.\exists y.\forall z.J(z,y) \vee \neg J(z,x))$
- Unfortunately, failure is difficult to interpret[280].

[280] `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 (➜ p.284) wrongly, the search strategy (➜ p.297) was not adequate for the problem, enumeration of unifiers (➜ p.192) in a bad order. Or a combination thereof. Or it might be that too many unsafe steps (➜ p.296) are needed, since `fast_tac` limits their number.

13 Term Rewriting

13.1 Higher-Order Rewriting

Motivation: Recall equational proofs (➜ p.119). They work by replacing equals by equals. They can be formally justified (➜ p.122).

It is practical to view deduction to some extent as equational proving and give it some attention algorithmically. This will be even more true later. We speak of simplification or (higher-order) (➜ p.301) rewriting.

Simplification: Examples

- In a FOL proof: rewrite $(\forall x. P\, x \wedge Q\, x)$ to $(\forall x. P\, x) \wedge (\forall x. Q\, x)$.
- In school arithmetic: simplify $0 + (x + 0)$[281] to x.
- In functional programming: simplify $[a, b, d] @ [a, b]$[282] to $[a, b, d, a, b]$.

This is all based on rewrite rules as in functional programming[283]:

$$\begin{aligned} [] @ X &= X \\ (x :: X) @ Y &= x :: (X @ Y) \end{aligned}$$

[281] Simplifying $0+(x+0)$ to x is something you have learned in school. It is justified by the usual semantics of arithmetic expressions. Here, however, we want to see more formally how such simplification works, rather than why it is justified.

[282] Lists are a common datatype in functional programming. $[a, b, d, a, b]$ is a list. Actually, this notation is syntactic sugar (➜ p.37) 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 (➜ p.67). In the λ-calculus, we would speak of a (special kind of) constant (this will become clear later (➜ p.304)).

[283] For example, the lines

$$\begin{aligned} [] @ X &= X \\ (x :: X) @ Y &= x :: (X @ Y) \end{aligned}$$

Why Higher-Order?

- Formally, rewriting operates on λ-terms, since we use the λ-calculus to encode object logics (➜ p.196).
- We speak of higher-order rewriting because the variables in the rewriting rules might have functional type such as $i \to o$ or $(i \to o) \to o$. Higher-order rewriting involves higher-order unification (➜ p.192).

---

define the list concatenation function @.

Term Rewriting: Foundation

- Recall (➜ p.118): An equational theory consists of rules (➜ p.104)

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

$$\frac{x = y \quad P(x)}{P(y)}\,\textit{subst}\ (\text{➜ p.110})$$

- plus additional (possibly conditional) rules of the form $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Rightarrow \phi = \psi$.

The additional rules can be interpreted as rewrite rules[284], i.e. they are applied from left to right.

[284] An equational theory is a formalism based on equational rules of the form $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$.

A term rewriting system (to be defined shortly) is another formalism, based of rewrite rules. They also have the form $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$, 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 $e(t)$ (➜ p.110) (resp. $e'(t)$ (➜ p.110))
   (b) for a rewrite rule $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$, match[285] (unify) ϕ against t, i.e., find θ such that $\phi\theta = t$
   (c) solve[286] $(\phi_1 = \psi_1, \ldots, \phi_n = \psi_n)\theta$
   (d) replace $e(t)$ (➜ p.110) by $e(\psi\theta)$ (➜ p.110) (resp. $e'(t)$ (➜ p.110) by $e'(\psi\theta)$ (➜ p.110))

[285] Given two terms s and t, a unifier (➜ p.192) 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).

[286] This 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 system[287].

[287] The 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 (➜ p.178) 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

Rewriting: Example

$$\begin{array}{lcrl} x+0 & = & x & \text{(neutr)} \\ x+y & = & y+x & \text{(comm)} \\ (x+y)+z & = & x+(y+z) & \text{(assoc)} \end{array}$$

$$\begin{array}{rcl} (1+3)+5 & = & 1+((5+0)+3) \\ 1+(3+5) & = & 1+((5+0)+3) \\ 1+(3+5) & = & 1+(5+3) \\ 1+(3+5) & = & 1+(3+5) \end{array}$$

Similar to equational proofs (➜ p.120).

from that, there may be function symbols (➜ p.67) in some object logic. On the metalevel (and hence also for the purpose of term rewriting), these would be constants (➜ p.307).

- There may be predicate symbols (➜ p.67) in some object logic. On the metalevel (and hence also for the purpose of term rewriting), these would be constants (➜ p.307).
- A constant is a λ-term consisting of just one symbol from a set *Const*. Constants (➜ p.307) 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.

Term Rewriting is Non-Trivial

- There are two major problems: this decision procedure may fail because:
  - it diverges (the rules are <u>not terminating</u>), e.g. $x + y = y + x$ or $x = y \Longrightarrow y = x$;
  - rewriting does not yield a unique normal form (the rules are not confluent (➜ p.159)), e.g. rules $a = b$,

---

See the following example:

The following is an example rewrite sequence, using the rules (➜ p.300) for lists (➜ p.300). The picked subterm which is being replaced is underlined in each step:

$$\begin{array}{lcll}
\underline{(a :: (b :: (d :: []))) @ (a :: (b :: []))} & = & [a, b, d, a, b] & \leadsto \\
a :: (\underline{(b :: (d :: [])) @ (a :: (b :: []))}) & = & [a, b, d, a, b] & \leadsto \\
a :: (b :: (\underline{(d :: []) @ (a :: (b :: []))})) & = & [a, b, d, a, b] & \leadsto \\
a :: (b :: (d :: (\underline{[] @ (a :: (b :: []))}))) & = & [a, b, d, a, b] & \leadsto \\
a :: (b :: (d :: (a :: (b :: [])))) & = & [a, b, d, a, b] & \leadsto
\end{array}$$

Note the we are done now, as the right-hand side is identical to the left-hand side, modulo the use of syntactic sugar (➜ p.300).

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 (➜ p.304) of a term.

One could also imagine that object-level function and pred-

$a = c$[288].

- 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 (➜ p.176).

[288] For a rewriting system consisting of rules $a = b$, $a = c$, one cannot rewrite $b = c$ to prove the equality, although it holds:

$$\dfrac{\dfrac{a = b}{b = a}\,\mathit{sym} \qquad a = c}{b = c}\,\mathit{trans}$$

13.2 Extensions of Rewriting

- Symmetric rules are problematic, e.g. ACI:[289]

$$\begin{array}{rcll} (x+y)+z & = & x+(y+z) & \text{(A)} \\ x+y & = & y+x & \text{(C)} \\ x+x & = & x & \text{(I)} \end{array}$$

- Idea: apply only if replaced term gets smaller w.r.t. some term ordering[290]. In example, if $(y+x)\theta$ (➜ p.303) is smaller than $(x+y)\theta$ (➜ p.303).

- Ordered rewriting solves rewriting modulo ACI[291], using derived rules (exercise).

[289] ACI stands for associative, commutative and idempotent. In

$$\begin{array}{rcll} (x+y)+z & = & x+(y+z) & \text{(A)} \\ x+y & = & y+x & \text{(C)} \\ x+x & = & x & \text{(I)} \end{array}$$

the constant $+$ (➜ p.304) is written infix (➜ p.65).

[290] The biggest problem for term rewriting is (non-)termination. For some crucial rules, this problem is solved by ordered term rewriting. A term ordering is any partial order (➜ p.113) between ground (i.e., not containing free variables) terms.

One can define a term ordering by giving some function, called norm, from ground terms to natural numbers. Then a term is smaller than another term if the number assigned to the first term is smaller that the number assigned to the second term.

[291] Consider an equational theory consisting only of those rules (apart from *refl*, *sym*, *trans*, *subst* (➜ p.118)). Apart

from that, the language may contain arbitrary other constant symbols. For such a language, it is possible to give a term ordering that will assign more weight to the same term on the left-hand-side of a + than on the right-hand side. We can base such a term ordering on a norm. For example, the inductive definition of a norm $|\_|$ (➜ p.310) might include the line:

$$|s + t| := 2|s| + |t|$$

This means that if $|s| > |t|$, then $|s + t| = 2|s| + |t| > 2|t| + |s| = |t + s|$.

This has two effects:

- Applications of (A) or (I) always decrease the weight of a term (provided the weight of s is > 0):

  $$|(s + t) + r| = 2|s + t| + |r| = 4|s| + 2|t| + |r| \; > \; 2|s| + 2|t| + |r| = 2|s| + |t + r| = |s + (t + r)|.$$
- Applications of (C) are only possible if the left-hand side is heavier than the right-hand side.

Rules such as $F(G\,c) = \ldots$[292] lead to highly ambiguous matching (➜ p.303) and hence inefficiency.

Solution is to restrict to higher-order pattern rules:

A term t is a HO-pattern if

- it is in β-normal form (➜ p.153); and
- any free (➜ p.148) F in t occurs in a subterm $F\,x_1 \ldots x_n$ where the x_i are η-equivalent (➜ p.158) to distinct bound variables.

Matching (unification) (➜ p.303) is decidable, unitary ('unique') and efficient algorithms exist.

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 (➜ p.117)) is used in standard mathematics for the absolute value of a number and is standard for norms as well.

[292] For higher-order rewriting, it is very problematic to have rules containing terms of the form $F(G\,c)$ on the left-hand side, where F and G are free variables and c is a constant or bound variable. The reason can be seen in an example: Suppose you want to rewrite the term $f(g(h(i\,c)))$ where f, g, h, i are all constants. There are four unifiers of $F\,(G\,c)$ and $f(g(h(i\,c)))$:

$$\begin{array}{l}
[F \leftarrow f,\ G \leftarrow (\lambda x.g(h(i\,x)))],\\
[F \leftarrow (\lambda x.f(g\,x)),\ G \leftarrow (\lambda x.h(i\,x))],\\
[(F \leftarrow \lambda x.f(g(h\,x))),\ G \leftarrow (\lambda x.i\,x)],\\
[(F \leftarrow \lambda x.f(g(h(i\,x)))),\ G \leftarrow (\lambda x.x)].
\end{array}$$

This ambiguity makes such TRSs (➜ p.304) very inefficient.

HO-Pattern Rewriting (Cont.)

A rule $\ldots \Rightarrow \phi = \psi$ is a HO-pattern rule if:

- ϕ is a HO-pattern;
- all free (➜ p.148) variables in ψ occur also in ϕ; and
- ϕ is constant-head, i.e. of the form $\lambda x_1..x_m.c\,p_1 \ldots p_n$ (where c is a constant (➜ p.304), $m \geq 0$, $n \geq 0$).

Example:[293] $(\forall x.P\,x \wedge Q\,x) = (\forall x.P\,x) \wedge (\forall x.Q\,x)$

Result: HO-pattern rules allow for very effective quantifier reasoning.

[293] Further examples:

- $(\exists x.P\,x \vee Q\,x) = (\exists x.P\,x) \vee (\exists x.Q\,x)$
- $(\exists x.P \to Q\,x) = P \to (\exists x.Q\,x)$
- $(\exists x.P\,x \to Q) = (\forall x.P\,x) \to Q$

In these examples, you may assume that first-order logic is our object logic.

On the metalevel (➜ p.178), and hence also for the sake of term rewriting, $\forall, \exists$ are constants (➜ p.304).

In the notation $(\forall x.P\,x \wedge Q\,x)$, 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: $(\forall x.P\,x \wedge Q\,x)$ is a "pretty-printed" version of `ALL` $(P$ & $Q)$.

You may want to look at a theory file (say, `IFOL.thy` (➜ p.289)) to get a flavor of this. The principle was explained thoroughly before (➜ p.216).

Extensions Related to `if` − `then` − `else`

The **`if-then-else`** construct will play an important role later (➜ p.362). It asks for special rewrite rules.

Extension: Congruence Rewriting

Problem :

$$\texttt{if}\,A\,\texttt{then}\,P\,\texttt{else}\,Q \;=\; \texttt{if}\,A\,\texttt{then}\,P'\,\texttt{else}\,Q$$

where $P = P'$ under condition A

is not a rule[294].

Solution in Isabelle (➜ p.293): explicitely admit this extra class of rules (<u>congruence rewriting</u>)

$$[\![A \Longrightarrow P = P']\!] \Longrightarrow$$
$$\texttt{if}\,A\,\texttt{then}\,P\,\texttt{else}\,Q \;=\; \texttt{if}\,A\,\texttt{then}\,P'\,\texttt{else}\,Q$$

[294] Rewrite rules (➜ p.302) have the form $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$ (several equations imply one equation). It is not possible that any of the equations $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n$ again depend on some condition, as in

$$\texttt{if}\,A\,\texttt{then}\,P\,\texttt{else}\,Q \;=\; \texttt{if}\,A\,\texttt{then}\,P'\,\texttt{else}\,Q$$

where $P = P'$ under condition A

Extension: Splitting Rewriting

Problem:

$$P(\texttt{if}\,A\,\texttt{then}\,x\,\texttt{else}\,y) = \texttt{if}\,A\,\texttt{then}\,(P\,x)\,\texttt{else}\,(P\,y)$$

is not a HO-pattern rule (since it is not constant-head (➜ p.311)).

Solution in Isabelle (➜ p.293): explicitely admit this extra class of rules (case splitting).

13.3 Organizing Simplification Rules

- Standard (HO-pattern conditional ordered rewrite (➜ p.309)) rules;
- congruence rules (➜ p.313);
- splitting rules (➜ p.314).

Isabelle (➜ p.293) data structure: `simpset`[295]. Some operations[296]:

- `addsimps` : `simpset` $*$ `thm list` $\to$ `simpset`
- `delsimps` : `simpset` $*$ `thm list` $\to$ `simpset`
- `addcongs` : `simpset` $*$ `thm list` $\to$ `simpset`
- `addsplits` : `simpset` $*$ `thm list` $\to$ `simpset`

[295] The `simpset` is an abstract datatype and at the same time an ML unit function for returning the current simplifier set. This is in analogy to the classifier set (➜ p.284).

[296] These functions manipulate the simplifier set, in analogy to the classifier set (➜ p.??).

Commutativity (➜ p.308) can be added without losing termination.

How to Apply the Simplifier?

Several versions (➜ p.293) of the simplifier:

- `simp_tac` : `simpset` → `int` → `tactic`
- `asm_simp_tac` : `simpset` → `int` → `tactic`
  (includes assumptions into `simpset`)
- `asm_full_simp_tac` : `simpset` → `int` → `tactic`
  (rewrites assumptions, and includes them into simpset)

Using global[297] simplifier sets: `Simp_tac`, `Asm_simp_tac`, `Asm_full_simp_tac`.

[297] `Simp_tac`, `Asm_simp_tac`, `Asm_full_simp_tac` work like their lower-case counterparts but use the current (global) simplifier set and hence do not take a simplifier set as first argument (e.g., `Simp_tac` has type `int` → `tactic`)
There are analogous capitalized versions for the tactics of the classical reasoner (➜ p.297).

13.4 Summary on Term Rewriting

Simplifier is a powerful proof tool for

- conditional equational formulas (➜ p.309)
- ACI-rewriting (➜ p.308)
- quantifier reasoning (➜ p.311)
- congruence rewriting (➜ p.313)
- automatic proofs by case splitting (➜ p.314).

Fortunately, failure is quite easy to interpret[298].

[298] When you use `simp_tac`, usually you can just look at the term that you get to understand which simplification has not worked although you think that it should have worked.

13.5 Summary on Last Three Sections

- Although Isabelle is an interactive theorem prover, it is a flexible environment with powerful automated proof procedures.
- For classical (➜ p.40) logic and set theory, procedures (➜ p.296) like `blast_tac` and `fast_tac` decide many tautologies.
- For equational theories (datatypes (➜ p.568), evaluating functional programs (➜ p.539), but also higher-order logic (➜ p.320)) `simp_tac` (➜ p.317) decides many tautologies (and is fairly easy to control).

14 HOL: Foundations

14.1 Overview

HOL is expressive foundation[299] for

- Mathematics: analysis, algebra, ...
- Computer science: program correctness, hardware verification, ...

HOL is very similar to $\mathcal{M}$ (➜ p.228), but it "is" an object logic[300]!

- HOL is classical[301].
- Still[302] important: modeling of problems/domains (now within HOL).
- Still important: deriving relevant reasoning principles.

[299] Theorem proving in higher-order logic is an active research area with some impressive applications.

[300] The differences between $\mathcal{M}$ (➜ p.228) 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 logics that can be represented by $\mathcal{M}$, just like propositional logic (➜ p.10) or first-order logic (➜ p.60). That is to say, we use HOL as object logic.

[301] Recall (➜ p.40) the distinction between classical and intuitionistic logics. There is a particular rule (➜ p.351) in HOL from which the rule of the excluded middle (➜ p.391) can be derived. This is in contrast to constructive (➜ p.321) (intuitionistic) logics.

[302] We have previously looked at metatheory (➜ p.228), i.e., how can one logic be represented/modeled in a metalogic.

Isabelle/HOL vs. Alternatives

We will use Isabelle/HOL[303].

- Could forgo the use of a metalogic[304] and employ alternatives, e.g., HOL system or PVS, or constructive provers[305] such as Coq or Nuprl.
- Choice depends on culture and application.

In particular, we have seen how general reasoning principles (➜ p.231) 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?

[303] We use Isabelle/HOL, and this means that HOL is an object logic represented by the metalogic $\mathcal{M}$ (➜ p.228).

[304] There are theorem proving systems that have no metalogic, but rather have a particular logic hard-wired into them, e.g. a HOL system or PVS.

[305] Constructive provers are based on intuitionistic logic. The rationale is that one has to give evidence (➜ p.40) for any statement. Coq and Nuprl are examples of such systems.

Safety through Strength

Safety[306] via conservative (definitional) extensions (➜ p.398):

- Small kernel of constants and rules;
- extend theory with new constants and types defined using existing ones;
- derive properties/theorems.

Contrast with:

- Weak logics (e.g., propositional logic): can't define much;
- axiomatic extensions[307]: can lead to inconsistency.

Bertrand Russell once likened the advantages of postulation over definition to the advantages of theft over honest toil!

[306] The principle is simple: the smaller a system is, the easier it is to check that it is correct, and the more confident one can be about it.

We have seen this before when we argued for the use of metalogics (➜ p.231). However, in that context, we still had to add further axioms (➜ p.261) 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.

[307] What we attempt to do here has similarities to the process of representing (➜ p.228) 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 (➜ p.398). Other than that, we will not add any axioms (➜ p.261)!

Set Theory as Alternative?

Set theory is the logician's choice as basis for modern mathematics.

- ZFC[308] [Zer07, Frä22]: has been implemented in Isabelle, with impressive applications!
- Neumann-Bernays-Gödel [Ber91]: equivalent to ZFC, but finitely axiomatizable[309].

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 (➜ p.138).
- A class cannot belong to another class (let alone a set)!

[308] ZFC stands for Zermelo-Fränkel set theory with choice [Dev93, Ebb94].

[309] Strictly speaking, an axiom (➜ p.48) within the object language in question. In this sense, the axiom of the excluded middle (➜ p.40) from propositional logic, $A \vee \neg A$ (for example) is not an axiom, because A is a meta-variable which could stand for an arbitrary formula, and thus $A \vee \neg A$ is not within the object language of propositional logic. One says that $A \vee \neg A$ is an axiom schema that represents infinitely many axioms.

So far we have not made this distinction explicit in most places, although we have raised this issue very early on (➜ p.29).

Now a theory is finitely axiomatizable if it only uses axioms, but no axiom schemata.

Finally: We Choose HOL!

HOL developed by [Chu40, Hen50] and rediscovered by [And02, GM93].

- Rationale: one usually works (➜ p.141) with typed entities.
- Reasoning is then easier with support for types.

  HOL is classical logic based on $\lambda^{\rightarrow}$ (➜ p.330).
- Isabelle/HOL also supports "mod cons"[310] like polymorphism (➜ p.179) and type classes (➜ p.182)!

  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. (Larry Paulson)

[310] "Mod cons" stands for "modern conveniences".

What Does Higher-Order Mean?

“Type” order[311]	Logic order	
		Example
Just o	0?	$A \wedge B \to B \wedge A$
1	1	$\forall x,\, y.\, R(x,y) \to R(y,x)$
+ quantification	2	$\mathit{False} \equiv \forall P.\, P$ $P \wedge Q \equiv \forall R.\, (P \to Q \to R)$
2	3	
+ quantification	4	$\forall X.\, (X(R,S) \leftrightarrow (\forall x.\, R(x) \to S(x)))$ $\to X(R',S')$ $(\equiv \mathit{subrel}(R',S'))$
$\vdots$	$\vdots$	$\vdots$

[311] Recall the definition of an order on types (➜ p.219) and assume here, as we did in the lecture on representing syntax (➜ p.211), that there is a type i of individuals and a type o for truth values.

In the sequel, we follow [And02, §50], who uses a definition of order slightly different from ours (➜ p.219). I will phrase his definition using the concept of predicate type:

- i is a type of order 0.
- every type of the form

  $$\underbrace{i \to \ldots i \to}_{n \text{ times}} o,$$

  where $n \geq 0$, is a predicate type of order 1.
- If $\tau_1, \ldots, \tau_n$ are predicate types, then $\tau_1 \to \ldots \to \tau_n \to o$ is a predicate type whose order is 1+ the maximum of the orders of $\tau_1, \ldots, \tau_n$.

Note that this means that there are no function symbols, since we did not consider types of the form $\ldots \to i$. However it is better to say that we simply disregard them in the subsequent explanations, for simplicity.

In the table, we classify logics by the order of the non-logical symbols (➜ p.102) (e.g., for first-order logic: variables, 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.

We start this hierarchy with first-order logic.

It has symbols of first-order type (predicate symbols), but quantification is allowed only over individuals, which are of order 0.

Now, if one admits quantification over symbols of first-order type, i.e., over symbols of type o or $i \to \ldots \to i \to o$, one obtains second-order logic.

Now, if one admits symbols of second-order type (symbols taking predicate symbols as arguments), one obtains third-order logic.

Now, if one admits quantification over symbols of second-order type, one obtains fourth-order logic.

Hence quantification over nth-order variables corresponds to $(2n)$th-order logic.

In the end, one will never bother to discuss, say, $7th$-order logic, since higher-order logic is the union of all logics of finite order, and this is what we will be working with.

Andrews has said that propositional logic might be regarded as zeroth order logic, but unfortunately, propositional logic cannot be found in this hierarchy in a straightforward way. According to the hierarchy, below first-order logic there should be a logic where the symbols are of order 0 and quantification over such symbols is allowed. But in fact, in propo-

Explanation for $subrel(R', S')$.[312]

sitional logic the symbols are of type o, which is of order 1 but is not the only type of order 1, and no quantification is allowed at all.

However, once you take higher-order logic as your point of reference and not propositional or first-order logic, which can just be viewed as special cases, you will probably not find this bothering anymore.

[312] Consider 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 $\forall x.\, R(x) \to S(x)$.

Now instead of defining such a predicate and writing, say, a formula $subrel(R', S')$, one could abstract from that name and write

$$\forall X.\, (X(R, S) \leftrightarrow (\forall x.\, R(x) \to S(x))) \to X(R', S')$$

The subformula $X(R, S) \leftrightarrow (\forall x.\, R(x) \to S(x))$ 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 $\lambda^{\rightarrow}$ (➜ p.162) with certain default types and constants.

Default constants can be called logical symbols (➜ p.102).

Types (Review)

Given a set of type constructors (➜ p.187), say $\mathcal{B}$[313] $= \{$*bool*, $\_ \rightarrow \_$ (➜ p.188), *ind*[314], $\_ \times \_$[315], $\_$ *list*, $\_$ *set*, $\ldots\}$, polymorphic types (➜ p.189) are defined by τ ::= (➜ p.16) α | $(\tau, .., \tau)\ T$ (➜ p.187) where α is a type variable.

- *bool* is also called o in literature [Chu40, And02]. Confusingly (➜ p.245), 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 polymorphism[316]!

[313] As before (➜ p.187), 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 (➜ p.162).

$\mathcal{B}$ 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.

[314] *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 (➜ p.334).

[315] For any two types τ and σ, we write $\tau \times \sigma$ for the type of pairs where the first component is of type τ and the second component is of type σ.

The infix syntax is in analogy to $\rightarrow$ (➜ p.188).

The pair type is not in the core of HOL, but it can be defined (➜ p.420) in it.

[316] We have seen the generalization (➜ p.181) of $\lambda^{\rightarrow}$ to poly-

Terms

Reminder (➜ p.163): $e \;\; ::= \;\;$ (➜ p.16) $x \;\; | \;\; c \;\; | \;\; (ee) \;\; | \;\; (\lambda x^{\tau}$[317]$. \, e)$

Typing rules as in polymorphic λ-calculus (➜ p.181), with Σ defining and typing (➜ p.165) constants.

Terms of type *bool* are called (well-formed) formulae.

In HOL, Σ always includes:

$$\begin{array}{rcl}
\mathit{True}, \mathit{False}^{318} & : & \mathit{bool} \\
= & : & \alpha \to \alpha \to \mathit{bool} \quad \text{(polymorphic, or set}^{319}\text{)} \\
\to & : & \mathit{bool} \to \mathit{bool} \to \mathit{bool} \\
\epsilon & : & (\alpha \to \mathit{bool}) \to \alpha \quad \text{(in Isabelle: } \texttt{Eps} \text{ or } \texttt{SOME}^{320}\text{)}
\end{array}$$

morphism.

Note that in order to simplify the presentation, we neglect polymorphism in the section on semantics (➜ p.333). 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 (➜ p.128) + functions

- FOL: structure (➜ p.71) is domain and functions/relations. Many-sorted FOL: domains are sort-indexed

$$\mathcal{A} = \langle \mathcal{D}_1, \ldots, \mathcal{D}_n, I_{\mathcal{A}} \rangle$$

- HOL extends idea: $\mathcal{D}$ indexed by (infinitely many) types.
- Complications due to polymorphism (➜ p.331) [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 $Y \subseteq X$ and $Y \neq \emptyset$, then $Y \in \mathcal{U}$

Prod: If $X, Y \in \mathcal{U}$ then $X \times Y \in \mathcal{U}$.

Pow: If $X \in \mathcal{U}$ then $\wp(X) = \{Y \mid Y \subseteq X\} \in \mathcal{U}$

Infty: $\mathcal{U}$ contains a distinguished infinite set[321] I

Choice: There is a function $ch \in \Pi_{X \in \mathcal{U}}.X$ (➜ p.336).

[321] The infinity axiom

$$\frac{}{\exists f^{(ind \to ind)} \text{ (➜ p.364)}.\mathit{injective}\ f \wedge \neg \mathit{surjective}\ f}\ \mathit{infty}$$

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 (➜ p.549).

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"[322]. E.g.: $(x, y) \equiv \{\{x\}, \{x, y\}\}$.

[322] According to usual mathematical practice, one would argue that if two sets A and B are well-defined, then the set $A \times B$ of pairs (tuples) (a, b) where $a \in A$ and $b \in B$ is also well-defined.

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 (➜ p.233). (One can only "hope" [GM93, page 193] that no confusion arises.) 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 (➜ p.340) invisible, because x and y might be se-

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 $ch \in \Pi_{X \in \mathcal{U}}.X$[323], i.e., ch is of dependent type.

Essentially, the constant ϵ will be interpreted as ch, but you will see the technical details later (➜ p.342).

mantic objects "of different type".

[323] When we write $ch \in \Pi_{X \in \mathcal{U}}.X$, i.e., ch is of dependent type, then this is a statement on the semantic level. The expression $\Pi_{X \in \mathcal{U}}.X$ is not part of the formal syntax of HOL (unlike in LF, a system we have not treated here), and its meaning is only described in plain English, by saying that ch takes a set $X \in \mathcal{U}$ as argument and returns a member of X.

Function Space in $\mathcal{U}$

Define set $X \to Y$ as (graphs of) functions[324] from X to Y.

- For nonempty X and Y[325], this set is nonempty and is a subset of $\wp(X \times Y)$.
- From closure conditions (➜ p.334): $X, Y \in \mathcal{U}$ then $X \to Y \in \mathcal{U}$.

[324] In any basic math course on algebra, we learn that a binary relation between X and Y is set of a pairs 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 $X \to Y$, is a nonempty subset of the set of relations on X and Y, i.e., $\wp(X \times Y)$. Since $X \to Y$ is nonempty, by **Prod** (➜ p.334) we have that $X \to Y \in \mathcal{U}$.

[325] It is crucial in the semantics that any type is inhabited (➜ p.334), i.e., has an element. The reason for this is that otherwise, there would be terms (➜ p.332) for which we cannot give a semantics:

Suppose ρ was an empty (non-inhabited) type. Then we cannot give any semantics to the term x^ρ. Moreover,

Distinguished Sets

From

Infty: $\mathcal{U}$ contains a distinguished infinite set (➜ p.334) I

Sub: If $X \in \mathcal{U}$ and $Y \subseteq X$ and $Y \neq \emptyset$, then $Y \in \mathcal{U}$

it follows that the following sets exist in $\mathcal{U}$:

---

if the signature (➜ p.332) includes a constant c^ρ, then we cannot give a semantics to c^ρ. Even if we only consider closed (➜ p.404) terms (i.e., terms without free variables), and we explicitly forbid the existence of a constant c^ρ for an empty type ρ, there will be terms for which we cannot give a semantics. The simplest example is the term $\lambda x^\rho.x$.

We know (➜ p.142) that λ-terms denote functions, as in $\lambda x^\rho.x$, and so it is natural to expect that all functions we can write in the λ-calculus actually exist in the semantics. Generally, the function space $X \to Y$ is empty if X or Y is empty. This means that $\mathcal{D}_{\tau\to\sigma}$ (➜ p.340) would necessarily be empty if τ is empty.

One way of understanding why it would be bad if some λ-terms denoting functions had no semantics is by looking at β-reduction: for any types τ,σ and a constant c of type σ, we expect $(\lambda x^\tau.c)\,x = c$. But this wouldn't hold if we cannot give a semantics to $(\lambda x^\tau.c)$ since $\mathcal{D}_{\tau\to\sigma}$ is empty.

Therefore: inhabitation.

Unit: A distinguished 1-element[326] set $\{1\}$

Bool: A distinguished 2-element set $\{T, F\}$.

One specific point where inhabitation is crucial is related to the ϵ-operator (➜ p.342), as we will see later.

In the book [GM93] that is one of the sources for this lecture, inhabitation is mentioned, but it is not explained why it is crucial.

Here we speak of semantic inhabitation, i.e., our semantic universe must be big enough so that all terms (of type τ) can be given a meaning (in $\mathcal{D}_\tau$). This is a different question from whether there might be types that are not inhabited (syntactically) in the first place, i.e., types for which there exists no term of this type (compare this to the Curry-Howard isomorphism (➜ p.172)). Thus we are concerned with making sure that every term has a meaning, not that every meaning has a term. However, it turns out that in HOL, each type τ is also syntactically inhabited, namely e.g. by the term $\epsilon_{(\tau \to bool) \to \tau}(\lambda x^\tau . \mathit{True})$.

[326] Of course, the conditions on $\mathcal{U}$ do not per se enforce the existence of sets containing the elements 1 or T or F. Just

Frames

For semantics, we neglect polymorphism (➜ p.331). τ and σ range over types.

A frame is a collection $\{\mathcal{D}_\tau\}_\tau$ of non-empty sets (domains (➜ p.334)) $\mathcal{D}_\tau \in \mathcal{U}$, one for each type τ, where:

- $\mathcal{D}_{bool} = \{T, F\}$;
- $\mathcal{D}_{\tau\to\sigma} \subseteq \mathcal{D}_\tau \to \mathcal{D}_\sigma$, i.e., some collection of functions (➜ p.337) from $\mathcal{D}_\tau$ to $\mathcal{D}_\sigma$.
- $\mathcal{D}_{ind}$ (➜ p.331) $= I$ (➜ p.334).

Note: for fundamental reasons discussed later (➜ p.354), one cannot simply define $\mathcal{D}_{\tau\to\sigma} = \mathcal{D}_\tau \to \mathcal{D}_\sigma$ at this stage.

---

as well, one could say that they enforce the existence of sets containing elements ☕ or 🚲 or ⚽.

The name of a semantic element is ultimately irrelevant, and therefore 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.

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 $\mathcal{D}_\tau$ where:

- $\mathcal{J}(\mathit{True}) = T$ and $\mathcal{J}(\mathit{False}) = F$;
- $\mathcal{J}(=_{\tau\to\tau\to \mathit{bool}})$[327] is equality on $\mathcal{D}_\tau$;
- $\mathcal{J}(\to)$ is implication function over $\mathcal{D}_{\mathit{bool}}$. For $b, b' \in \{T, F\}$,

$$\mathcal{J}(\to)(b, b') = \begin{cases} F & \text{if } b = T \text{ and } b' = F \\ T & \text{otherwise} \end{cases}$$

[327] For $=$ and ϵ, we give type subscripts in the presentation of the semantics since we assume, conceptually, that there are infinitely many copies (➔ p.332) of those constants, one for each type. We do this to avoid explicit polymorphism in this presentation.

Interpretations (Cont.)

- $\mathcal{J}(\epsilon_{(\tau \to bool) \to \tau}$ (➜ p.341)$)$ is defined by (for $f \in (\mathcal{D}_\tau \to \mathcal{D}_{bool})$):

  $$\mathcal{J}(\epsilon_{(\tau \to bool) \to \tau})(f)^{328} = \begin{cases} ch(f^{-1}(\{T\})) & \text{if } f^{-1}(\{T\}) \neq \emptyset \\ ch(\mathcal{D}_\tau) & \text{otherwise} \end{cases}$$

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.

[328] We have

$$\mathcal{J}(\epsilon_{(\tau \to bool) \to \tau})(f) = \begin{cases} ch(f^{-1}(\{T\})) & \text{if } f^{-1}(\{T\}) \neq \emptyset \\ ch(\mathcal{D}_\tau) & \text{otherwise} \end{cases}$$

ch is a (semantic) function (➜ p.336) which takes a nonempty set and returns an element from that set. f is a semantic function from $\mathcal{D}_\tau$ 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 (➜ p.416) of the set $f^{-1}(\{T\})$.

Now the type of ϵ is $(\tau \to bool) \to \tau$ (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 (➜ p.340) as nonempty set (➜ p.334)) is picked. Note the importance of inhabitation (➜ p.337).

A Terminological Note

The terminology is slightly different from FOL:

In FOL, "$\langle \{\mathcal{D}_\tau\}_\tau, \mathcal{J}\rangle$" is called structure (➜ p.71) and "$\mathcal{J}$" is called interpretation (➜ p.71).

In HOL, $\langle \{\mathcal{D}_\tau\}_\tau, \mathcal{J}\rangle$ is called interpretation and $\mathcal{J}$ is called denotation function.

The Value of Terms (Naïve)

In analogy to FOL (➜ p.73), given an interpretation $\mathfrak{M} = \langle\{\mathcal{D}_\tau\}_\tau, \mathcal{J}\rangle$ and a type-indexed collection of assignments[329] $A = \{A_\tau\}_\tau$, 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. $\mathcal{V}_A^{\mathfrak{M}}(s_{\tau\to\sigma}$[330]$t_\tau) = (\mathcal{V}_A^{\mathfrak{M}}(s))(\mathcal{V}_A^{\mathfrak{M}}(t))$, 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 $\mathcal{D}_\tau$ into $\mathcal{D}_\sigma$ whose value for each $e \in \mathcal{D}_\tau$ is $\mathcal{V}_{A[x\leftarrow e]}^{\mathfrak{M}}$[331]$(t)$.

What is the problem? Condition 4!

[329] An assignment (previously called valuation (➜ p.71)) maps variables to elements of a domain (➜ p.334).

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 $\mathcal{D}_\tau$ [GM93]. Note that a variable has a type by virtue of a context Γ, which is suppressed in our presentation of models.

[330] In the presentation of models, we give type subscripts for the cases $\mathcal{V}_A^{\mathfrak{M}}(s_{\tau\to\sigma}t_\tau)$ 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.

[331] $A[x \leftarrow e]$ 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 $\mathcal{D}_\tau$ into $\mathcal{D}_\sigma$ in condition 4 must live in $\mathcal{D}_{\tau\to\sigma}$[332]; for this, $\mathcal{D}_{\tau\to\sigma}$ 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)[333] model.

[332] In condition 4, the semantics of $\lambda x^\tau .\, t_\sigma$ is defined unambiguously as a certain function. But in general, there is no guarantee that this function is actually in $\mathcal{D}_{\tau\to\sigma}$, and in this case, $\mathfrak{M} = \langle \{\mathcal{D}_\tau\}_\tau, \mathcal{J} \rangle$ would not be a model.

[333] General models must be distinguished from standard models, as we will see later (➜ p.346).

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 $\mathcal{D}_{\tau\to\sigma}$ is the set of all functions from $\mathcal{D}_\tau$ to $\mathcal{D}_\sigma$, then it is certainly "big enough". In this case, we speak of a standard model. Important for completeness (➜ p.354).

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 (➜ p.75), "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 $\mathfrak{M}$ (➜ p.344) if there exists an assignment A such that $\mathcal{V}_A^{\mathfrak{M}}(\phi) = T$.

A formula ϕ is valid wrt. a model $\mathfrak{M}$ (➜ p.344) 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 (➜ p.345).

A formula ϕ is valid in the standard sense if it is valid in every standard model (➜ p.346).

Existence of Values

Closure conditions (➜ p.344) 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 $\mathcal{V}^{\mathfrak{M}}_{A}(\lambda x^{\tau}.\, x)$ is defined, the identity function from $\mathcal{D}_{\tau}$ to $\mathcal{D}_{\tau}$ must always belong to $\mathcal{D}_{\tau\to\tau}$.
- Closure under application: let $\mathcal{D}_{\mathbb{N}}$ be the natural numbers, and suppose we have a term ϕ denoting the successor function, then $\mathcal{D}_{\mathbb{N}\to\mathbb{N}}$ must contain k where $k\, x = x + 2$, since $k = \mathcal{V}^{\mathfrak{M}}_{A}(\lambda x_{\mathbb{N}}.\, \phi(\phi\, x))$.

Idea: if you can write it down, then it exists!

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* (➜ p.351) ("true or false"), which would be hard to read if we did that.

So we allow ourselves to "cheat"[334] and also use constants *True*, *False*, $\vee$ to write rule *tof* (➜ p.351).

Later we will define those constants, i.e., regard them as syntactic sugar (➜ p.37).

[334] Rule *tof* (➜ p.351) can be written as follows:

$$\frac{}{\begin{aligned}&(\lambda\psi.\ \ (\phi = (\lambda x.x = \lambda x.x) \rightarrow \psi) \rightarrow\\ &\qquad\quad (\phi = ((\lambda\eta.\eta) = \lambda x.(\lambda x.x = \lambda x.x)) \rightarrow \psi) \rightarrow \psi) =\\ &(\lambda x.(\lambda x.x = \lambda x.x))\end{aligned}}\ \mathit{tof}$$

This is very complicated, so let's trace this back to the more readable phrasing

$$\begin{aligned}&(\lambda\psi.\ \ (\phi = \mathit{True} \rightarrow \psi) \rightarrow\\ &\qquad\quad (\phi = ((\lambda\eta.\eta) = \lambda x.\mathit{True}) \rightarrow \psi) \rightarrow \psi) =\\ &(\lambda x.\mathit{True}) \equiv\\ &(\forall\psi.\ \ (\phi = \mathit{True} \rightarrow \psi) \rightarrow\\ &\qquad\quad (\phi = (\forall\eta.\eta) \rightarrow \psi) \rightarrow \psi) \equiv\\ &(\forall\psi.\ \ (\phi = \mathit{True} \rightarrow \psi) \rightarrow\\ &\qquad\quad (\phi = \mathit{False} \rightarrow \psi) \rightarrow \psi) \equiv\\ &\phi = \mathit{True} \vee \phi = \mathit{False}\end{aligned}$$

Our notation for rule *tof* (➜ p.351) is thus based on the following definitions:

$$\begin{array}{lcl}
\textit{True} \ (\text{➜ p.362}) & = & (\lambda x^{\textit{bool}} \ (\text{➜ p.362}).x = \lambda x.x) \\
\textit{False} \ (\text{➜ p.362}) & = & \forall \phi^{\textit{bool}} \ (\text{➜ p.362}).\phi \ (\text{➜ p.362}) \\
\vee \ (\text{➜ p.362}) & = & \lambda \phi \eta . \forall \psi . (\phi \to \psi) \to (\eta \to \psi) \to \psi
\end{array}$$

$$\frac{}{\Gamma \vdash \phi = \phi}\,\mathit{refl} \qquad \frac{\Gamma \vdash \phi = \eta \quad \Gamma \vdash P(\phi)}{\Gamma \vdash P(\eta)}\,\mathit{subst}$$

$$\frac{\Gamma \vdash \phi\, x = \eta\, x}{\Gamma \vdash \phi = \eta}\,\mathit{ext}^{*}$$[335]

$$\frac{\Gamma, \phi \vdash \eta}{\Gamma \vdash \phi \rightarrow \eta}\,\mathit{impI}$$

$$\frac{\Gamma \vdash \phi \rightarrow \eta \quad \Gamma \vdash \phi}{\Gamma \vdash \eta}\,\mathit{mp}$$

$$\frac{}{\Gamma \vdash (\phi \rightarrow \eta) \rightarrow (\eta \rightarrow \phi) \rightarrow (\phi = \eta)}\,\mathit{iff}$$

$$\frac{}{\phi = \mathit{True} \vee \phi = \mathit{False}}\,\mathit{tof}$$ (➜ p.349)

$$\frac{\Gamma \vdash \phi x}{\Gamma \vdash \phi(\epsilon x.\phi x)}\,\mathit{selectI}$$[336] [337]

[335] The rule

$$\frac{\Gamma \vdash \phi\, x = \eta\, x}{\Gamma \vdash \phi = \eta}\,\mathit{ext}$$

has the side condition that $x \notin FV(\Gamma)$.

Phrased like

$$\frac{\phi\, x = \eta\, x}{\phi = \eta}\,\mathit{ext}$$

the rule has the side condition that x must not occur freely (➜ p.69) in the derivation of $\phi\, x = \eta\, x$.

[336] You may wonder why there is no rule for eliminating ϵ. We will later (➜ p.386) 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 (➜ p.361) ∃ and the if-then-else constructs. Once we have derived the appropriate rules for those, we will not explicitly encounter ϵ anymore.

[337] For 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 (➜ p.334):

$$\frac{}{\exists f^{(ind \to ind)}.injective^{338}\ f \wedge \neg surjective\ f}\ infty$$

Has special role. Interesting to look at HOL with or without infinity (➜ p.354). Won't (➜ p.549) consider infinity today.

Note "cheating" (➜ p.361) (use of $\exists$).

These eight (nine) rules are the entire basis!

---

more used to than higher-order abstract syntax (➜ p.216):

$\epsilon x.\phi x$ stands for $\epsilon(\phi)$.

$\forall x.\phi(x)$ stands for $\forall(\phi)$, and likewise for $\exists$.

We have done the same previously (➜ p.246) for $\mathcal{M}$.

Soundness and Completeness

Soundness is straightforward [And02, p. 240].

Soundness and Completeness

Completeness only follows w.r.t. general models (➜ p.345), as opposed to standard (➜ p.346) models. Recall that a standard model is one where $\mathcal{D}_{\tau\to\sigma}$ is always the set of all functions from $\mathcal{D}_\tau$ to $\mathcal{D}_\sigma$.

There are formulas that are valid (➜ p.347) in all standard models, but not in all general models, and which cannot be proven in our calculus (➜ p.349). Our calculus can prove the formulas that are true in all general models including non-standard ones (Henkin models [Hen50]). This reconciles HOL with Gödel's incompleteness theorem[339] [Hen50, Mil92].

If we consider a version of HOL without infinity (➜ p.334), then every model is a standard model[340] and so completeness holds.

[339] This is a standard trick when faced with the problem that a deductive system is not complete. One can either enlarge the set of axioms, or one can weaken the models by permitting more models. If we allow more models, then fewer theorems will be valid (i.e., hold in all models), and so fewer theorems will have to be provable in the derivation system.

Here, completeness is based on general models, and not standard (➜ p.346) models. This resolves the apparent contradiction with Gödel's incompleteness theorem: HOL with infinity contains I (➜ p.334), hence the natural numbers (➜ p.554), 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].

[340] We 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 Isabelle[341].

---

one where each domain (➜ p.334) is finite (note that $\mathcal{U}$ is still infinite, since there are infinitely many types, e.g., *bool*, *bool* → *bool*, *bool* → *bool* → *bool*, ...)).

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 (➜ p.344) $\mathcal{D}_{\tau\to\sigma} = \mathcal{D}_\tau \to \mathcal{D}_\sigma$.

For details consult [And02, §54].

[341] 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/`

There you will also find all the derivations of the rules presented in this lecture.

However, the presentation of this lecture is partly based on HOL.thy of Isabelle 98, which in turn is based on a standard book [GM93]. 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 display-

We present language and rules[342] using "mathematical" syntax, but also comparing with Isabelle (concrete/HOAS (➜ p.216)) syntax.

We take polymorphism (➜ p.331) back on board.

ing Isabelle files which uses some symbols beyond the usual ASCII set (➜ p.364).

[342] We will mix natural deduction (➜ p.23) (with discharging assumptions), natural deduction written in sequent style (➜ p.47), 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

$$\frac{\begin{matrix}[\rho] \\ \vdots \\ \psi\end{matrix}}{\phi}$$

is written as

$$\frac{\rho, \Gamma \vdash \psi}{\Gamma \vdash \phi}$$

in sequent style or

$$\frac{\rho \Longrightarrow \psi}{\phi}$$

using the Isabelle meta-implication $\Longrightarrow$.

A rule

$$\frac{\psi}{\phi(x)}$$

with side condition that x must not occur free in any undischarged assumption on which ψ depends is written as

$$\frac{\Gamma \vdash \psi}{\Gamma \vdash \phi(x)}$$

in sequent style, where the side condition reads: x must not occur free in Γ. Using the Isabelle meta-universal quantification, the rule is written

$$\frac{\bigwedge x.\psi}{\phi(x)}$$

$$\Sigma_0 =$$

$$\begin{array}{ll}
\{\ \mathit{True},\ \mathit{False}\text{[343]} & : \mathit{bool},\\
\neg\_\text{[344]} & : \mathit{bool} \to \mathit{bool},\\
\_ \wedge \_,\ \_ \vee \_,\ \_ \to \_ & : \mathit{bool} \to \mathit{bool} \to \mathit{bool},\\
\forall\_,\ \exists\_ & : (\alpha \to \mathit{bool}) \to \mathit{bool},\\
\epsilon\_ & : (\alpha \to \mathit{bool}) \to \alpha,\\
\mathit{if\_then\_else\_} & : \mathit{bool} \to \alpha \to \alpha \to \alpha,\\
\_ = \_ & : \alpha \to \alpha \to \mathit{bool}\}
\end{array}$$

We will switch between the various ways of writing the rules! This means in particular that we will use $\Longrightarrow$ and $\bigwedge$ from Isabelle's metalogic (➜ p.228).

[343] For convenience (and to save space, we write $\ldots a : \tau,\ b : \tau \ldots$ as $\ldots a, b : \tau \ldots$ in a signature. This is of course syntactic sugar (➜ p.37).

[344] We use a notation with _ to indicate the arity and fixity of constants, as this has been done for type constructors (➜ p.187) before.

The whole matter of arity of fixity is one of notational convenience. For example, as the type of $\wedge$ indicates, we should write $(\wedge\phi)\psi$ (Curryed notation (➜ p.156)), 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.355).

Basic Rules in Mixed (➜ p.356) Notation

$$\frac{}{\phi = \phi}\,\mathit{refl} \qquad \frac{\phi = \eta \quad P(\phi)}{P(\eta)}\,\mathit{subst}$$

$$\frac{\phi\, x = \eta\, x}{\phi = \eta}\,\mathit{ext}^{*}\ (\text{➜ p.351}) \qquad \frac{\phi \Longrightarrow \eta}{\phi \to \eta}\,\mathit{impI}$$

$$\frac{\phi \to \eta \quad \phi}{\eta}\,\mathit{mp}$$

$$\frac{}{(\phi \to \eta) \to (\eta \to \phi) \to (\phi = \eta)}\,\mathit{iff}$$

$$\frac{}{\phi = \mathit{True} \lor \phi = \mathit{False}}\,\mathit{tof} \qquad \frac{\phi x}{\phi(\epsilon x.\phi x)}\,\mathit{selectI}$$

No more “Cheating”: The Definitions

$$\begin{array}{rcl}
\mathit{True}\text{[345]} & =\text{[346]} & (\lambda x^{\mathit{bool}}\text{ (➜ p.362)}.x = \lambda x.x) \\
\forall\text{[347]} & = & \lambda \phi^{\alpha\to \mathit{bool}}\text{ (➜ p.332)}.(\phi = \lambda x.\mathit{True}) \\
\mathit{False}\text{[348]} & = & \forall \phi^{\mathit{bool}\text{[349]}}.\phi\text{[350]} \\
\vee\text{[351]} & = & \lambda \phi\eta.\forall\psi.(\phi \to \psi) \to (\eta \to \psi) \to \psi \\
\wedge\text{[352]} & = & \lambda \phi\eta.\forall\psi.(\phi \to \eta \to \psi) \to \psi \\
\neg\text{[353]} & = & \lambda\phi.(\phi \to \mathit{False}) \\
\exists\text{[354]} & = & (\lambda\phi.\phi(\epsilon x.\phi x)) \\
\mathit{If}\text{[355]} & = & \lambda \phi^{\mathit{bool}} xy.\epsilon z.(\phi = \mathit{True} \to z = x)\wedge \\
 & & (\phi = \mathit{False} \to z = y)
\end{array}$$

[345]

$$\mathit{True} = (\lambda x^{\mathit{bool}}.x = \lambda x.x)$$

The term $\lambda x^{\mathit{bool}}.x = \lambda x.x$ evaluates to T (➜ p.341), 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 $\mathit{bool} \to \mathit{bool}$.

This is necessary for reasons that will become clear later (➜ p.404).

Note that $(\lambda x^{\mathit{bool}}.x = \lambda x.x)$ is closed (➜ p.404). Definitions must always be closed (➜ p.404).

[346] It is a design choice if we want to add these definitions at the level of the object logic (HOL) (➜ p.320) or at the level of the metalogic $\mathcal{M}$ (➜ p.228). In the first case, we would use = and have axioms such as

$$\mathit{True} = (\lambda x^{\mathit{bool}}.x = \lambda x.x)$$

In the second case, we would have meta-axioms

$$\mathit{True} \equiv (\lambda x^{\mathit{bool}}.x = \lambda x.x)$$

This would mean that we would regard *True* merely as syntactic sugar (➜ p.37). The second way corresponds to what is done in Isabelle, see `HOL.thy` (➜ p.355). It is technically more convenient since rewriting (➜ p.299) is based on meta-level equalities.

Logically, it is not a big difference which way one chooses.

$\mathit{If} = \lambda\phi xy.\epsilon z.(\phi = \mathit{True} \to z = x) \wedge (\phi = \mathit{False} \to z = y)$

The constant *If* stands for the if-then-else construct. Note first that $\epsilon z.(\phi = \mathit{True} \to z = x) \wedge (\phi = \mathit{False} \to z = y)$ is η-equivalent to $\epsilon z.(\lambda z.(\phi = \mathit{True} \to z = x) \wedge (\phi = \mathit{False} \to z = y))\, z$, which is written $\epsilon(\lambda z.(\phi = \mathit{True} \to z = x) \wedge (\phi = \mathit{False} \to z = y))$ in the "real" HOL syntax, which uses the concept of HOAS (➜ p.216).

The expression $\epsilon(\lambda z.(\phi = \mathit{True} \to z = x) \wedge (\phi = \mathit{False} \to z = y))$ picks a term from the set of terms z such that $(\phi = \mathit{True} \to z = x) \wedge (\phi = \mathit{False} \to z = y)$ holds. But this means that $z = x$ if $\phi = \mathit{True}$, or $z = y$ if $\phi = \mathit{False}$.

Since *If* should be a function which takes ϕ, x and y as arguments, we must abstract over those variables, giving $\lambda\phi xy.\epsilon z.(\phi = \mathit{True} \to z = x) \wedge (\phi = \mathit{False} \to z = y)$.

Note: Different Syntaxes

Mathematical	vs.	Isabelle, e.g.
$\neg\phi$		`Not Phi`
$\lambda x^{bool}.P$		`%`[356]`x` :: [357]`bool.` P
HOAS (➜ p.216)	vs.	concrete, e.g.
$\forall\,(\lambda x^{\tau}.(\wedge p(x)\,q(x)))$		$\forall x^{\tau}.p(x) \wedge q(x)$
$\epsilon\,(P)$		$\epsilon x.P(x)$

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).

[356] Note that the λ-binder of the object logic HOL is not distinguished from the λ-binder of Isabelle's metalogic $\mathcal{M}$ (➜ p.228). One could introduce an object level constant *lambda*, but one quickly sees that it would be an unnecessary overhead.

[357] As we have learned previously (➜ p.163), λ-abstracted variables should have a type superscript, although this superscript is often omitted since the type can be inferred (➜ p.332).

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 (➜ p.353) straightforward.
- Logic complete (➜ p.354) (w.r.t. general models, but not standard (➜ p.346) models).
- Next lecture we will see how all well-known inference rules can be derived.

15 HOL: Deriving Rules

Outline

Last lecture (➜ p.320): Introduction to HOL

- Basic syntax (➜ p.330) and semantics (➜ p.333)
- Basic eight (or nine) rules (➜ p.350)
- Definitions (➜ p.361) of *True*, *False*, $\wedge$, $\vee$, $\forall$ ...

Today:

- Deriving rules (➜ p.366) for the defined constants
- Outlook on the rest of this course (➜ p.394)

Reminder: Different Syntaxes

Mathematical	vs.	Isabelle, e.g.
$\neg\phi$		`Not Phi`
$\lambda x^{bool}.P$		`%` (➜ p.364)`x` :: `bool`. P
HOAS (➜ p.216)	vs.	concrete, e.g.
$\forall\,(\lambda x^{\tau}$ (➜ p.364)$.(\wedge p(x)\; q(x)))$		$\forall x^{\tau}$ (➜ p.364)$.p(x) \wedge q(x)$
$\epsilon\,(P)$		$\epsilon x.P(x)$

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

$$\begin{aligned}
\mathit{True}\ (\text{➜ p.362}) &= (\lambda x^{\mathit{bool}\ (\text{➜ p.362})}.x = \lambda x.x)\\
\forall\ (\text{➜ p.362}) &= \lambda\phi^{\alpha\rightarrow \mathit{bool}\ (\text{➜ p.332})}.(\phi = \lambda x.\mathit{True})\\
\mathit{False}\ (\text{➜ p.362}) &= \forall\phi^{\mathit{bool}\ (\text{➜ p.362})}.\phi\ (\text{➜ p.362})\\
\vee\ (\text{➜ p.362}) &= \lambda\phi\eta.\forall\psi.(\phi\rightarrow\psi)\rightarrow(\eta\rightarrow\psi)\rightarrow\psi\\
\wedge\ (\text{➜ p.362}) &= \lambda\phi\eta.\forall\psi.(\phi\rightarrow\eta\rightarrow\psi)\rightarrow\psi\\
\neg\ (\text{➜ p.362}) &= \lambda\phi.(\phi\rightarrow \mathit{False})\\
\exists\ (\text{➜ p.362}) &= (\lambda\phi.\phi(\epsilon x.\phi x))\\
\mathit{If}\ (\text{➜ p.362}) &= \lambda\phi xy.\epsilon z.(\phi = \mathit{True}\rightarrow z = x)\wedge\\
&\qquad (\phi = \mathit{False}\rightarrow z = y)
\end{aligned}$$

Derived Rules

The definitions (➜ p.369) 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 derivation[358]

$$\dfrac{s = t \qquad \dfrac{}{s = s}\,\textit{refl}\ (\text{➜ p.350})}{t = s}\,\textit{symsubst}\ (\text{➜ p.350})$$

[358] We present most of those proofs by giving a derivation tree (➜ p.23) for it, but sometimes, we also give an Isabelle proof script.

Note also the mix of syntaxes (➜ p.356).

- Isabelle rule `s=t ==> 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 (➜ p.370)

  $$\dfrac{\dfrac{r = s}{s = r}\,\textit{sym}\ (\text{➜ p.370}) \qquad s = t}{r = t}\,\textit{transsubst}\ (\text{➜ p.350})$$

  Isabelle rule `[| r=s; s=t |] ==> r=t`

- Congruences (only Isabelle forms):

  `(f::'a=>'b) = g ==> f(x)=g(x)` (*fun_cong*)
  `x=y ==> f(x)=f(y)` (*arg_cong*)

  Isabelle proofs using *subst* (➜ p.350) and *refl* (➜ p.350).

Equality of Booleans (*iffI*)

Rule *iffI* and ND derivation (➜ p.370)

$$\dfrac{\dfrac{\dfrac{}{(P \to Q) \to (Q \to P) \to (P = Q)}\,\mathit{iff} \qquad \dfrac{\begin{matrix}[P]\\ \vdots\\ Q\end{matrix}}{P \to Q}\,\mathit{impI}}{(Q \to P) \to (P = Q)}\,\mathit{mp} \qquad \dfrac{\begin{matrix}[Q]\\ \vdots\\ P\end{matrix}}{Q \to P}\,\mathit{impI}}{P = Q}\,\mathit{iffImp}$$

Isabelle rule `[| P ==> Q; Q ==> P |] ==> P=Q`.

Uses *mp* (➜ p.350), *iff* (➜ p.350), *impI* (➜ p.360).

Equality of Booleans (*iffD2*)

Rule *iffD2* and ND derivation (➜ p.370)

$$\dfrac{\dfrac{P = Q}{Q = P}\,\textit{sym}\ (\text{➜ p.370}) \qquad Q}{P}\,\textit{iffD2subst}\ (\text{➜ p.350})$$

Isabelle rule `[| P=Q; Q |] ==> P`.

15.2 *True*

$True = ((\lambda x^{bool}.x) = (\lambda x.x))$

- Rule *TrueI* and ND derivation (➜ p.370)

$$\frac{}{True(\lambda x.x) = (\lambda x.x)}\ \mathit{TrueIrefl}\ (➜\ \text{p.350})$$

- Rule *eqTrueE* and ND derivation (➜ p.370)

$$\frac{P = True \quad \dfrac{}{True}\ \mathit{TrueI}}{P}\ \mathit{eqTrueEiffD2}\ (➜\ \text{p.374})$$

Isabelle rule `P=True ==> P`.

True (Cont.)

- Rule *eqTrueI* and ND derivation (➜ p.370)

  $$\dfrac{\dfrac{}{\mathit{True}}\ \mathit{TrueI}\ (\text{➜ p.375}) \qquad P}{P = \mathit{True}}\ \mathit{eqTrueIiffI}\ (\text{➜ p.373})$$

  Note that 0 assumptions were discharged.

  Isabelle rule `P ==> P=True`.

15.3 Universal Quantification

$\forall P = (P = (\lambda x.\mathit{True}))$

- Rule *allI* and ND derivation (➜ p.370)

$$\cfrac{\cfrac{P(x)}{P(x) = \mathit{True}}\ \mathit{eqTrueI}\ (\text{➜ p.376})}{\forall P P = \lambda x.\ \mathit{True}}\ \mathit{allIext}\ (\text{➜ p.350})$$

  Inherits (➜ p.94) the side condition of *ext* (➜ p.351): x must not occur freely in the derivation of $P(x)$.

  Isabelle rule `(!!x.  P(x)) ==> ALL x.  P(x)`.

Example Illustrating Side Condition

$$\dfrac{\dfrac{[r(x)]^1}{r(x) \to r(x)}\ \to\text{-}I^1}{\forall x.\, r(x) \to r(x)}\ \mathit{allI}$$

Why is this correct? Let's do it without using *allI* explicitly:

$$\dfrac{\dfrac{\dfrac{[r(x)]^2}{r(x) \to r(x)}\ \to\text{-}I^2}{(r(x) \to r(x)) = \mathit{True}}\ \mathit{eqTrueI}}{\lambda x.\, (r(x) \to r(x)) = \lambda x.\, \mathit{True}}\ \mathit{ext}$$

The side condition is respected.

Universal Quantification (Cont.)

- Rule *spec* (recall (➜ p.351) $\forall P$ means $\forall x.Px$) and ND derivation (➜ p.370)

$$\dfrac{\dfrac{\forall P P = \lambda x.\mathit{True}}{P(t) = \mathit{True}}\ \mathit{fun\_cong}\ (\text{➜ p.372})}{P(t)}\ \mathit{speceqTrueE}\ (\text{➜ p.375})$$

  Isabelle rule `ALL x::'a.  P(x) ==> P(x)`.

Note: Need universal quantification to reason about *False* (since $\mathit{False} = (\forall P.P)$).

15.4 *False*

$False = (\forall P.P) \qquad (= \forall(\lambda P.P)$ (➜ p.351))

- FalseI: No rule!
- Rule *FalseE* and ND derivation (➜ p.370)

$$\frac{False\forall P.\,P}{P}\ FalseEspec$$ (➜ p.379)

Isabelle rule `False ==> P`.

False (Cont.)

- Rule *False_neq_True* and ND derivation (➜ p.370)

$$\dfrac{\dfrac{\mathit{False} = \mathit{True}}{\mathit{False}}\ \mathit{eqTrueE}\ (\text{➜ p.375})}{P}\ \mathit{False\_neq\_TrueFalseE}\ (\text{➜ p.380})$$

  Isabelle rule `False=True ==> P`.

- Similar:

$$\dfrac{\mathit{True} = \mathit{False}}{P}\ \mathit{True\_neq\_False}$$

15.5 Negation

$\neg P = P \rightarrow \mathit{False}$

- Rule *notI* and ND derivation (➜ p.370)

$$\dfrac{\begin{matrix}[P] \\ \vdots \\ \mathit{False}\end{matrix}}{\neg P P \rightarrow \mathit{False}}\ \mathit{notIimpI}\ (➜\ \text{p.360})$$

  Isabelle rule `(P ==> False) ==> ~P`.

Negation (2)

- Rule *notE* and ND derivation (➜ p.370)

$$\dfrac{\dfrac{\neg P P \to \mathit{False} \quad P}{\mathit{False}}\,\mathit{mp}\ (\text{➜ p.350})}{R}\,\mathit{notEFalseE}\ (\text{➜ p.380})$$

Isabelle rule `[| ~P; P |] ==> R`.

Negation (3)

- Rule *True_Not_False* and ND derivation (➜ p.370)

$$\dfrac{\dfrac{[\mathit{True} = \mathit{False}]^1}{\mathit{False}}\ \mathit{True\_neq\_False}\ \text{(➜ p.381)}}{\neg(\mathit{True} = \mathit{False})}\ \mathit{True\_Not\_FalsenotI}^1$$

Isabelle rule `True ~= False`.

Uses *notI* (➜ p.382)

15.6 Existential Quantification

$\exists P = P(\epsilon x.P(x))$

- Rule *existsI* and ND derivation (➜ p.370)

  $$\frac{P(x)}{\exists P P(\epsilon x.P(x))} \textit{existsIselectI} \text{ (➜ p.350)}$$

  Isabelle rule `P(x) ==> EX x::'a.P(x)`.

Existential Quantification (Cont.)

- Rule *existsE* and ND derivation (➜ p.370)

$$\dfrac{\exists P P(\epsilon x.P(x)) \qquad \dfrac{\dfrac{\dfrac{\begin{matrix}[P(x)]^1\\ \vdots \\ Q\end{matrix}}{P(x) \to Q}\,\textit{impI}^1}{\forall x.(P(x) \to Q)}\,\textit{allI}}{P(\epsilon x.P(x)) \to Q}\,\textit{spec}}{Q}\,\textit{existsEmp}\text{[359]}$$

  Inherits side condition from *allI* (just like in FOL (➜ p.93)). On the meta-level[360], this derivation is extremely simple.

  Isabelle rule `[| EX x.P(x); !!x.P(x)==>Q |] ==> Q`.

[360] One can write the derivation of *existsE* as follows:

$$\dfrac{P(\epsilon x.P(x)) \qquad \dfrac{\bigwedge x.\, P(x) \Rightarrow Q}{P(\epsilon x.P(x)) \Rightarrow Q}\,\bigwedge\!-E}{Q}\,\textit{existsE}{\Rightarrow}\text{-}E$$

This is an attempt to capture in an ad-hoc tree notation how this derivation can be done in Isabelle. In particular, *existsE* inherits a side condition from the meta-level universal quantification. However, while this may help to understand how this derivation works in Isabelle, it is not very rigorous and you could not be expected to believe that the side condition checking is correct.

For a thorough account of side conditions in ND proofs, consult [SH84].

You might also justify *existsE* in plain English words, i.e., completely on the meta-level: If I have a derivation of Q from $P(x)$ not making any assumptions about x, and in addition I have a derivation of $P(\epsilon x.P(x))$, then I can combine these

15.7 Conjunction

$P \wedge Q = \forall R.(P \to Q \to R) \to R$

- Rule *conjI* and ND derivation (➜ p.370)

$$\cfrac{\cfrac{\cfrac{\cfrac{[P \to Q \to R]^1 \quad P}{Q \to R}\,\textit{mp}\ (\text{➜ p.350}) \qquad Q}{R}\,\textit{mp}\ (\text{➜ p.350})}{(P \to Q \to R) \to R}\,\textit{impI}\ (\text{➜ p.360})^1}{P \wedge Q \forall R.(P \to Q \to R) \to R}\,\textit{conjIallI}$$

Isabelle rule `[| P; Q |] ==> P & Q`.

---

two derivations: modify the first one by instantiating x with $\epsilon x.P(x)$. This justifies having *existsE*.

What happens in our rather complicated derivation is that we are turning a meta-level reasoning into an object-level one, which is more trustworthy for an ND derivation.

Conjunction (Cont.)

- Rule *conjEL* and ND derivation (➜ p.370)

$$\dfrac{\dfrac{P \wedge Q \forall R.(P \to Q \to R) \to R}{(P \to Q \to P) \to P}\,\mathit{spec} \qquad \dfrac{\dfrac{[P]^1}{Q \to P}\,\mathit{impI}}{P \to Q \to P}\,\mathit{impI}^1}{P}\,\mathit{conjELmp}\ (\text{➜ p.350}$$

Isabelle rule `P & Q ==> P`.

Uses *spec*, *impI*.

Conjunction (Cont.)

- $P \wedge Q \Rightarrow Q \qquad (\mathit{conjER})$
- $[\![P \wedge Q;\ [\![P; Q]\!] \Rightarrow R]\!] \Rightarrow R \qquad (\mathit{conjE})$ (rule analogous to *disjE* (➜ p.391))

15.8 Disjunction

$P \vee Q = \forall R.(P \to R) \to (Q \to R) \to R$

- Rule *disjIL* and ND derivation (➜ p.370)

$$\cfrac{\cfrac{\cfrac{\cfrac{[P \to R]^1 \quad P}{R}\,\mathit{mp}\ (\text{➜ p.350})}{(Q \to R) \to R}\,\mathit{impI}\ (\text{➜ p.360})}{(P \to R) \to (Q \to R) \to R}\,\mathit{impI}\ (\text{➜ p.360})^1}{P \vee Q \forall R.(P \to R) \to (Q \to R) \to R}\,\mathit{disjILallI}$$

Isabelle rule `P ==> P|Q`.

Disjunction (Cont.)

- $Q \Rightarrow P \vee Q$ (*disjIR*) similar
- Rule *disjE* and ND derivation (➜ p.370)

$$\dfrac{\dfrac{\dfrac{P \vee Q \forall R.(P \to R) \to (Q \to R) \to R}{(P \to R) \to (Q \to R) \to R}\,\mathit{spec} \qquad \dfrac{\begin{matrix}[P]\\ \vdots \\ R\end{matrix}}{P \to R}\,\mathit{impI}}{(Q \to R) \to R}\,\mathit{mp} \qquad \dfrac{\begin{matrix}[Q]\\ \vdots \\ R\end{matrix}}{Q \to R}\,\mathit{impI}}{R}\,\mathit{disjEmp}$$

Isabelle rule `[| P | Q; P ==> R; Q ==> R |] ==> R`.

- $P \vee \neg P$ (*excl_midd*). Follows using *tof* (➜ p.351). Uses *spec* (➜ p.379), *mp* (➜ p.350), *impI* (➜ p.360).

15.9 Miscellaneous Definitions

See HOL.thy (➜ p.355)!

Typical example (if-then-else (➜ p.362)):

$$\begin{aligned} \mathit{If} = \lambda \phi^{bool} xy.\epsilon z.\ & (\phi = \mathit{True} \to z = x) \\ \wedge\ & (\phi = \mathit{False} \to z = y) \end{aligned}$$

The way rules are derived should now be clear. E.g.,

$$\frac{P = \mathit{True}}{(\mathit{If}\ P\ x\ y) = x} \qquad \frac{P = \mathit{False}}{(\mathit{If}\ P\ x\ y) = y}$$

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 (➜ p.392)) can be defined.
- Rich theories can be obtained by a method we see next lecture (➜ p.398).

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 computer science.

Outline:

- The central method for making HOL scale up: conservative extensions (➜ p.398) (< 1 week)
- How the different parts of mathematics are encoded in the Isabelle/HOL library (➜ p.427) (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 (➜ p.431)
- Sets (➜ p.440)
- Functions (➜ p.451)
- (Least) fixpoints and induction (➜ p.467)
- (Well-founded) recursion (➜ p.497)
- Arithmetic (➜ p.545)
- Datatypes (➜ p.568)

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 (➜ p.590)
- Z[361] and data-refinement, CSP and process-refinement
- Object-oriented languages (Java-Light . . .)

We will do two case studies here: one on functional and one on imperative programming.

[361] Z and CSP are specification languages. CSP stands for communicating sequential processes.

Conservative Extensions: Motivation

- We have already seen that starting from an extremely small kernel containing just $=$, $\rightarrow$ and ϵ, we can define all the well-known logical symbols from FOL, plus the if-then-else.
- Our aim is to reason about a fairly large part of mathematics and computer science.
- When extending our language in this way, the general worry should be: could we get inconsistencies? We now consider more carefully how to prevent them.

16 Conservative Theory Extensions

Outline

In the previous lecture (➜ p.366), 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 $(\mathcal{B}, \Sigma, A)$, where $\mathcal{B}$ is a type signature (➜ p.331), Σ a signature (➜ p.332) and A a set of axioms[362].

Definition (theory extension):

[362] The definition of theory extension requires that A consists of axioms, not proper rules (➜ p.48). However, we have seen (➜ p.114) that any rule one might wish to postulate can also be phrased as an axiom (using $\rightarrow$ rather than $\Rightarrow$).

A theory $T' = (\mathcal{B}', \Sigma', A')$ is an extension of a theory $T = (\mathcal{B}, \Sigma, A)$ iff $\mathcal{B} \subseteq \mathcal{B}'$ and $\Sigma \subseteq \Sigma'$ and $A \subseteq A'$.

Definitions (Cont.)

Definition (conservative extension):

A theory extension $T' = (\mathcal{B}', \Sigma', A')$ of a theory $T = (\mathcal{B}, \Sigma, A)$ is conservative iff for the set of derivable formulas[363] *Th* we have

$$Th(T) = Th(T') \mid_{\Sigma},$$

where $\mid_{\Sigma}$ filters away all formulas not belonging to Σ.

Counterexample:

$$\frac{}{\forall f^{\alpha \to \alpha}.\ Y\, f = f\, (Y\, f)}\ \text{fix}$$ [364]

[363] The derivable formulas are terms of type *bool* derivable using the inference rules of HOL (➜ p.350). We write *Th*(T) for the derivable formulas of a theory T.

[364] Given a function $f : \alpha \to \alpha$, a fixpoint of f is a term t such that $f\, t = t$. Now Y is supposed to be a fixpoint combinator, i.e., for any function f, the term $Y\, f$ should be a fixpoint of f. This is what the rule

$$\frac{}{\forall f^{\alpha \to \alpha}.Y\, f = f\, (Y\, f)}\ \text{fix}$$

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 (➜ p.467).

Consistency Preserved

Corollary (consistency):

If T' is a conservative extension of T, then

$$\mathit{False} \notin \mathit{Th}(T) \Rightarrow \mathit{False} \notin \mathit{Th}(T').$$

Syntactic Schemata for Conservative Extensions

- Constant definition (➜ p.404)
- Type definition (➜ p.410)
- 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 $c \notin dom$[365](Σ);
- $A' = A \cup \{c = E\}$;
- E does not contain[366] c and is closed[367];
- no subterm of E has a type containing a type variable (➜ p.406) that is not contained in the type of c.

[365] The domain of Σ, denoted $dom(\Sigma)$, is $\{c \mid c : A \in \Sigma \text{ for some } A\}$.

Likewise, the domain of Γ, denoted $dom(\Gamma)$, is $\{x \mid x : A \in \Gamma \text{ for some } A\}$.

Note the abuse of notation (➜ p.168).

[366] If E did contain c then we would speak of a recursive definition, but at this stage, recursion (➜ p.497) is forbidden.

[367] A term is closed or ground if it does not contain any free (➜ p.69) variables.

Constant Definitions Are Conservative

Lemma (constant definitions):
Constant definitions are conservative [GM93, page 223].

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 $\phi \in Th(T)$.

The Need for the Side Conditions[368]

Here is a counterexample concerning closedness (➜ p.404) of E: Define $c : \mathit{bool}$ by the axiom $c = x$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{c=x}\,\text{axiom}}{\forall x.c=x}\,\mathit{allI}\ (\text{➜ p.377})}{c=\mathit{False}}\,\mathit{spec}\ (\text{➜ p.379}) \qquad \dfrac{\dfrac{\dfrac{}{c=x}\,\text{axiom}}{\forall x.c=x}\,\mathit{allI}\ (\text{➜ p.377})}{c=\mathit{True}}\,\mathit{spec}\ (\text{➜ p.379})}{\dfrac{\mathit{False}=\mathit{True}}{\mathit{False}}\,\mathit{False\_neq\_True}\ (\text{➜ p.381})}\,\mathit{subst}\ (\text{➜ p.350})$$

Intuition: when you define c as the variable x, then c just isn't a constant! Usually taken for granted.

[368] By side conditions we mean

- E does not contain c and is closed;
- no subterm of E has a type containing a type variable that is not contained in the type of c;

in the definition (➜ p.404).

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 (➜ p.190). 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 (➜ p.163).

The Need for the Side Conditions (2)

Now type-closedness (➜ p.406): Let $E \equiv \exists x^\alpha y^\alpha.\ x \neq y$ and suppose σ is a type inhabited (➜ p.337) by only one term, and τ is a type inhabited (➜ p.337) by at least two terms. Then we would have:

$$\begin{array}{rl} & c = c \qquad \text{holds by } \textit{refl} \text{ (➜ p.350)} \\ \Longrightarrow & (\exists x^\sigma y^\sigma.\, x \neq y) = (\exists x^\tau y^\tau.\, x \neq y) \\ \Longrightarrow & \mathit{False} = \mathit{True} \\ \Longrightarrow & \mathit{False} \end{array}$$

This explains definition of *True*[369]. Other (standard) example later (➜ p.551).

[369] *True* is defined as $\lambda x^{bool}.x = \lambda x.x$ (➜ p.362) and not $\lambda x^\alpha.x = \lambda x.x$. The definition must be type-closed (➜ p.406).

Constant Definition: Examples

Definitions of *True*, *False*, $\wedge$, $\vee$, $\forall$ ... (➜ p.361)

Here the original (➜ p.355) Isabelle syntax (`Ex_def` changed (➜ p.355)) Note the use of `!`[370] and meta-level (➜ p.362) equality.

```
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"
```

[370] "`!`" is just another Isabelle notation for `ALL`, and "`?`" is just another Isabelle notation for `EX`. See `HOL.thy` (➜ p.355) in the section "syntax (HOL)" (this is Isabelle 2005).

More Constant Definitions in Isabelle

Function application (`Let`), if-then-else, unique existence[371]:

```
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)"
```

Note use of `?` (➜ p.408).

Recall: `=>` is function type arrow (➜ p.184); also recall [] syntax (➜ p.185).

[371] We have never used unique existential quantification ($\exists!$) before. $\exists! x_1, \ldots, x_n.\phi(x_1, \ldots, x_n)$ is defined as $\exists x_1, \ldots, x_n.\phi(x_1, \ldots, x_n) \wedge (\forall y_1, \ldots, y_n.\phi(y_1, \ldots, y_n) \rightarrow x_1 = y_1 \wedge \ldots \wedge x_n = y_n)$.

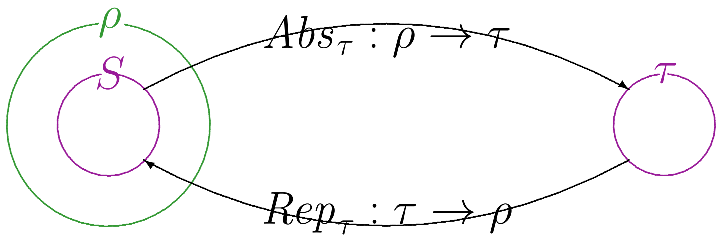
Note that in general $\exists! x.(\exists! y.\phi)$ is not the same as $\exists! xy.\phi)$.

16.3 Type Definitions

Type definitions, explained intuitively: we have

- an existing type ρ;
- a predicate $S : \rho \to \mathit{bool}$, defining a non-empty "subset"[372] of ρ;
- axioms stating an isomorphism between S and the new type τ.

[372] Although a set is formally a different object than a predicate, it is standard to interpret a predicate a set: the set of terms for which the predicate returns true.



Type Definition: Definition

Assume a theory $T = (\mathcal{B}, \Sigma, A)$ and a type ρ and a term S[373] such that $\Sigma \vdash$ (➜ p.190)$S : \rho \to bool$.
A theory extension $T' = (\mathcal{B}', \Sigma', A')$ of T is a type definition for type τ[374] (where τ fresh[375]), iff

[373] Here, S is any "predicate" (➜ p.325), i.e., term of type $\rho \to bool$, not necessarily a constant.

[374] A type definition is supposed to define a type constructor (➜ p.331) (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 (➜ p.189) (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.

[375] The type constructor τ must not occur in $\mathcal{B}$.

$$\begin{array}{lcll} \mathcal{B}' & = & \mathcal{B} \ \uplus\text{[376]} & \{\tau\}, \\ \Sigma' & = & \Sigma \ \cup & \{\mathit{Abs}_\tau\text{[377]} : \rho \to \tau, \mathit{Rep}_\tau \ \text{(➜ p.412)} : \tau \to \rho\} \\ A' & = & A \ \cup & \{\forall x.S\,(\mathit{Rep}_\tau\,x), \\ & & & \ \forall x.\mathit{Abs}_\tau(\mathit{Rep}_\tau\,x) = x\text{[378]}, \\ & & & \ \forall x.S\,x \to \mathit{Rep}_\tau(\mathit{Abs}_\tau\,x) = x \ \text{(➜ p.412)}\} \end{array}$$

Proof obligation[379] $\exists x.\, S\,x$ can be proven inside HOL!

[376] The symbol $\uplus$ denotes disjoint union, so the expression $A \uplus B$ is well-formed only when A and B have no elements in common. One thus uses this notation to indicate this fact.

[377] Of course we are giving a schematic definition here, so any letters we use are metanotation.

Notice that Abs_τ and Rep_τ stand for new constants. For any new type τ to be defined, two such constants must be added to the signature to provide a generic way of obtaining terms of the new type. Since the new type is isomorphic to the "subset" (➜ p.410) 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.

[378] The formulas

$$\begin{array}{l} \forall x.\mathit{Abs}_\tau(\mathit{Rep}_\tau\,x) = x \\ \forall x.S\,x \to \mathit{Rep}_\tau(\mathit{Abs}_\tau\,x) = x \end{array}$$

Type Definitions Are Conservative

Lemma (type definitions):

Type definitions are conservative.

Proof see [GM93, pp.230].

---

state that the "set" S (➜ p.410) and the new type τ are isomorphic. Note that Abs_τ should not be applied to a term not in "set" S (➜ p.410). Therefore we have the premise $S\,x$ in the above equation.

Note also that S could be the "trivial filter" $\lambda x. True$. In this case, Abs_τ and Rep_τ would provide an isomorphism between the entire type ρ and the new type τ.

[379]We have said previously (➜ p.410) that S should be a non-empty "subset" (➜ p.410) of τ. Therefore it must be proven that $\exists x. S\,x$. This is related to the semantics (➜ p.334).

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 (➜ p.422). Using this syntax, the "author" of a type definition can either explicitly provide a proof (see `Product_Type.thy` (➜ p.423)), 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* (➜ p.331) and → (➜ p.331), the types in HOL are already very rich.

We now give three examples to convince you.

`Sum_Type.thy` (➜ p.425)).

Example: Typed Sets

General scheme, substituting $\rho \equiv \alpha \to \mathit{bool}$ (α is any type variable (➜ p.189)), $\tau \equiv \alpha\ \mathit{set}$ (➜ p.411) (or *set* (➜ p.411)), $S \equiv \lambda x^{\alpha \to \mathit{bool}}.\mathit{True}$

$$\begin{array}{rcll}
\mathcal{B}' & = & \mathcal{B} & \uplus\ \{\tau \mathit{set}\}, \\
\Sigma' & = & \Sigma & \cup\ \{\mathit{Abs}_{\tau set} : \rho(\alpha \to \mathit{bool}) \to \tau\alpha\, \mathit{set}, \\
 & & & \quad \mathit{Rep}_{\tau set} : \tau\alpha\, \mathit{set} \to \rho(\alpha \to \mathit{bool})\} \\
A' & = & A & \cup\ \{\forall x.S\,(\mathit{Rep}_{\tau set}\, x), \mathit{True}, \\
 & & & \quad \forall x.\mathit{Abs}_{\tau set}(\mathit{Rep}_{\tau set}\, x) = x, \\
 & & & \quad \forall x.S\, x\, \mathit{True} \to \mathit{Rep}_{\tau set}(\mathit{Abs}_{\tau set}\, x) = x\}
\end{array}$$

Simplification since $S \equiv \lambda x.\mathit{True}$. Proof obligation (➜ p.412): $(\exists x.Sx)$ trivial since $(\exists x.\mathit{True}) = \mathit{True}$. Inhabitation propagates[380]!

[380] We have $S \equiv \lambda x^{\alpha \to \mathit{bool}}.\mathit{True}$, and so in $(\exists x.Sx)$, the variable x has type $\alpha \to \mathit{bool}$. The proposition $(\exists x.Sx)$ is true since the type $\alpha \to \mathit{bool}$ is inhabited (➜ p.337), e.g. by the term $\lambda x^{\alpha}.\mathit{True}$ or $\lambda x^{\alpha}.\mathit{False}$.

Beware of a confusion: This does not mean that the new type $\alpha\, \mathit{set}$, 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 $\mathit{Abs}_{set}\ (\lambda x.\mathit{False})$.

So we see that inhabitation of types propagates in the following sense: since each type τ is inhabited, the type $\tau\ \mathit{set}$ is inhabited as well.

Sets: Remarks

Any function $r : \alpha \to \mathit{bool}$ can be interpreted as a set of α; r is called characteristic function. That's what $\mathit{Abs}_{\mathit{set}}\ r$ does; $\mathit{Abs}_{\mathit{set}}$ is a wrapper saying "interpret r as set".

$S \equiv \lambda x.\,\mathit{True}$ and so S is trivial[381] in this case.

[381] We said that in the general formalism for defining a new type, there is a term S of type $\rho \to \mathit{bool}$ that defines a "subset" (➜ p.410) 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 $\alpha\,\mathit{set}$. 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 $\alpha\,\mathit{set}$. This is not the case!

Rather, S is $\lambda x.\,\mathit{True}$ and hence trivial in this example. Moreover, in the example, ρ is $\alpha \to \mathit{bool}$, and any term r of type ρ defines a set whose elements are of type α; $\mathit{Abs}_{\mathit{set}}\ r$ is that set.

More Constants for Sets

For convenient use of sets, we define more constants:

$$\begin{array}{rcl} \{x \mid f\,x\} &=& \mathit{Collect}\,[382]\; f = \mathit{Abs}_{set}\, f \\ x \in A &=& (\mathit{Rep}_{set}\ A)\,[383]\; x \\ A \cup B\ (\text{➜ p.129}) &=& \{x \mid x \in A \vee x \in B\} \\ &\vdots& \end{array}$$

Consistent set theory[384] adequate for most of mathematics

[382] We have seen *Collect* before in the theory file `NSet.thy` (naïve set theory (➜ p.124)).

Collect f is the set whose characteristic function (➜ p.416) is f. There is also a concrete (➜ p.364) (i.e., according to mathematical practice) syntax $\{x \mid f\,x\}$. It is called set comprehension. The correspondence between the HOAS (➜ p.364) *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 (➜ p.125).

Note also that *Collect* is the same (➜ p.443) 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/`

[383] We define

$$x \in A \;=\; (\mathit{Rep}_{set}\ A)\ x$$

and computer science.

In Isabelle/HOL however, sets are a special case (➜ p.443).

Here, sets are just an example to demonstrate type definitions. Later (➜ p.440) we study them for their own sake.

Since Rep_{set} has type (➜ p.415) $\alpha\, set \to (\alpha \to bool)$, this means that (➜ p.190) x is of type α and A is of type $(\alpha \to bool)$. Therefore $\in$ is of type $\alpha \to (\alpha\, set) \to bool$ (but written infix (➜ p.65)).

In the Isabelle theory file `Set.thy` (➜ p.418), you will indeed find that the constant : (Isabelle syntax for $\in$) has type $\alpha \to (\alpha\, set) \to bool$.

However, you will not find anything (➜ p.443) directly corresponding to Rep_{set}.

[384]Typed set theory is a conservative extension (➜ p.413) of HOL and hence consistent (➜ p.402).

Recall the problems with untyped set theory (➜ p.139).

Example: Pairs

Consider type $\alpha \to \beta \to \mathit{bool}$. We can regard a term $f : \alpha \to \beta \to \mathit{bool}$ as a representation of the pair (a, b), where $a : \alpha$ and $b : \beta$, iff $f\,x\,y$ is true exactly for $x = a$ and $y = b$. Observe:

- For given a and b, there is exactly one[385] such f (namely, $\lambda x^\alpha y^\beta .\, x = a \wedge y = b$).
- Some functions of type $\alpha \to \beta \to \mathit{bool}$ represent pairs and others don't (e.g., the function $\lambda xy.\, \mathit{True}$ 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.

[385] When we say that there is "exactly one" f, this is meant modulo equality in HOL. This means that e.g. $\lambda x^\alpha y^\beta .y = b \wedge x = a$ is also such a term since $(\lambda x^\alpha y^\beta .x = a \wedge y = b) = (\lambda x^\alpha y^\beta .y = b \wedge x = a)$ is derivable in HOL.

Type Definition for Pairs

This gives rise to a type definition where S (➜ p.410) is non-trivial:

$$\begin{array}{lcl} \rho & \equiv & \alpha \to \beta \to bool \\ S & \equiv & \lambda f^{\alpha\to\beta\to bool}.\exists ab.f = \lambda x^\alpha y^\beta.x = a \wedge y = b \\ \tau & \equiv & \alpha \times \beta \qquad\qquad\qquad\qquad\qquad\qquad (\times \text{ infix}) \end{array}$$

It is convenient to define a constant `Pair_Rep` (not to be confused with $Rep_\times$[386]) as $\lambda a^\alpha b^\beta.\lambda x^\alpha y^\beta.\ x = a \wedge y = b$[387]. Then `Pair_Rep` $a\, b = \lambda x^\alpha y^\beta.\ x = a \wedge y = b$.

[386] $Rep_\times$ would be the generic name for one of the two isomorphism-defining functions (➜ p.412).

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.

[387] We write $\lambda a^\alpha b^\beta.\lambda x^\alpha y^\beta.x = a \wedge y = b$ rather than $\lambda a^\alpha b^\beta x^\alpha y^\beta.x = a \wedge y = b$ to emphasize the idea that one first applies $Pair\_Rep$ to a and b, and the result is a function representing a pair, wich can then be applied to x and y.

Now in Isabelle

Isabelle has a special set-based[388] syntax for type definitions:

```
typedef
  ⟨typevars⟩ "T" ⟨fixity⟩
  = "{x.ϕ}"
```

How is this linked to our scheme (➜ p.411):

- the new type is called T' (➜ p.421);
- ρ is the type of x (inferred (➜ p.332));
- S is $\lambda x.\phi$;
- constants (➜ p.421) `Abs_`*T* and `Rep_`*T* are automatically generated.

[388] The syntax "$\{x.\phi\}$" does not just look like a set comprehension (➜ p.418), 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 (➜ p.443) 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

```
definition
      Pair_Rep :: "'a => 'b => 'a => 'b => bool"
where "Pair_Rep == (%a b. %x y. x=a & y=b)"
```

```
typedef
 ('a, 'b) prod (infixr "*" 20) =
   "{f.?a b. f=Pair_Rep(a::'a)(b::'b)}"
```

The keyword `definition`[389] introduces a constant definition. The definition and use of `Pair_Rep` (➜ p.421) is for convenience. There are "two names" (➜ p.421) $*$ and `prod`.

See `Product_Type.thy`[390].

[389] In Isabelle theory files, `consts` is the keyword preceding a sequence of constant declarations (i.e., this is where the Σ (➜ p.399) is defined), and `defs` is the keyword preceding the axioms that define these constants (i.e., this is where the A (➜ p.399) is defined).

`definition` combines the two, i.e. it allows for both constant declarations and definitions. When the `definition` syntax is used to define a constant c, then the identifier c_def is generated automatically. E.g.

```
definition
 id :: "'a => 'a"
where "id == %x. x"
```

will bind `id_def` to $id \equiv \lambda x.x$.

[390] 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/`

Example: Sums

An element of (α, β) `sum`[391] 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 $\alpha \to \beta \to bool \to bool$. We can regard $f : \alpha \to \beta \to bool \to bool$ 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 (➜ p.420).

[391] Idea of <u>sum</u> or <u>union</u> 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 (➜ p.178), 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

$$\texttt{datatype}\ (\alpha, \beta)\ \texttt{sum} = \mathit{Inl}\ \alpha \mid \mathit{Inr}\ \beta$$

So an element of (α, β) `sum` is either *Inl* a where $a : \alpha$ or *Inr* b where $b : \beta$.

Isabelle Syntax for Sum Example

```
definition
  Inl_Rep :: "'a => 'a => 'b => bool => bool"
where "Inl_Rep == (%a. %x y p. x=a & p)"
definition
  Inr_Rep :: "'b => 'a => 'b => bool => bool"
where "Inr_Rep == (%b. %x y p. y=b & ~p)"
```

```
typedef ('a,'b) sum =
  "{f. (?a. f = Inl_Rep(a::'a)) |
        (?b. f = Inr_Rep(b::'b))}"
```

See `Sum_Type.thy`[392].

How would you define[393] a type `even` based on `nat`?

[392] 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/`

[393] Suppose we have a type `nat` and a constant + with the expected meaning. We want to define a type `even` of even numbers. What is an even number?

The following choice of S (➜ p.411) 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 (➜ p.404): new constant must be defined using old constants. No recursion! Subtle side condition (➜ p.407) concerning types.
- Type definition (➜ p.410): new type must be isomorphic to a "subset" (➜ p.410) 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 (➜ p.414).

---

say as follows:

```
definition
  PLUS::[even,even] => even (infixl 56)
where PLUS_def "PLUS ==
                  %xy. Abs_Even (Rep_Even(x)+Rep_Even(x))"
```

Note that we chose to use names `even` and `Even` (➜ p.421), 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 [Zer07, Frä22, WR25, Göd31, Ber91, Chu40].

We call this the Principia Mathematica Structure [WR25].

The libraries of theorem provers follow this Principia Mathematica Structure — in reverse order![394]

[394] It is not surprising that the logical built-up of theorem prover is reversed w.r.t. to the historical development of mathematics and logics. Research usually starts from applications and the intuition and works its way back to the foundations.

The Roadmap

18 Orders

The Roadmap

We are looking at how the different parts of mathematics are encoded in the Isabelle/HOL library (➜ p.427).

- Orders
- Sets (➜ p.440)
- Functions (➜ p.451)
- (Least) fixpoints and induction (➜ p.467)
- (Well-founded) recursion (➜ p.497)
- Arithmetic (➜ p.545)
- Datatypes (➜ p.568)

Three Order Classes

We first define a syntactic class (➜ p.184) `ord`. It is the class of types for which symbols < and <= exist.

We then define two axiomatic classes (➜ p.185) `order` and `linorder` for which < and <= are required to have certain properties, that of being a partial order (➜ p.113), or a linear order (➜ p.115), resp.

Orders (in `Orderings.thy`[395])

```
axclass ord < type
consts
 "op <"  :: ['a::ord, 'a] => bool
 "op <=" :: ['a::ord, 'a] => bool
definition
  min :: "['a::ord, 'a] => 'a"
  where "min a b == (if a <= b then a else b)"
definition
  max :: "['a::ord, 'a] => 'a"
  where "max a b == (if a <= b then b else a)"
```

Recall `definition` (➜ p.423) syntax and note two uses of <[396].

[395] In previous versions of Isabelle (➜ p.355), there used to be a theory file `Ord.thy`. Nowadays orders are defined in `Orderings.thy`.

[396] The line

```
axclass order < ord
```

in the theory file states that `order` is a subclass (➜ p.184) 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 (➜ p.355), it used to be called `term`.

Orders (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 `Orderings.thy` (➜ p.434), 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.332).

Monotonicity

In `Orderings.thy` (➜ p.434), monotonicity used to be defined as:

```
mono      :: ['a::ord => 'b::ord] => bool
mono_def     "mono(f)  ==
             (!A B. A <= B --> f(A) <= f(B))
```

Now it is done using a completely different syntax, but one can still use monotonicity as before.

Some Theorems[397] about Orders

`monoI`	$(\bigwedge AB. A \leq B \Longrightarrow f\, A \leq f\, B) \Longrightarrow \mathit{mono}\, f$
`monoD`	$[\![\mathit{mono}\, f; A \leq B]\!] \Longrightarrow f\, A \leq f\, B$
`order_eq_refl`	$x = y \Longrightarrow x \leq y$
`order_less_irrefl`	$\neg\, x < x$
`order_le_less`	$(x \leq y) = (x < y \vee x = y)$
`linorder_less_linear`	$x < y \vee x = y \vee y < x$
`linorder_neq_iff`	$(x \neq y) = (x < y \vee y < x)$
`min_same`	$\mathit{min}\, x\, x = x$
`le_min_iff_conj`	$(z \leq \mathit{min}\, x\, y) = (z \leq x \wedge z \leq y)$

18.1 Summary on Orders

Type classes are a structuring mechanism in Isabelle:

[397] In 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 (➜ p.228), what we also call a **`thm`** (➜ p.293). Such theorems may contain the meta-level implication $\Longrightarrow$ and universal quantifier $\bigwedge$.

So they are not theorems within HOL. Logically, this is not a big deal as one switches between object and meta-level by the introduction and elimination rules for $\rightarrow$ (➜ p.350) and $\forall$ (➜ p.377). 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 (➜ p.184) (e.g. $t :: \alpha :: ord$ as in Haskell [HHPW96]): merely a mechanism to structure visibility of operations.
- Axiomatic classes (➜ p.185) (e.g. $t :: \alpha :: order$): a mechanism for structuring semantic knowledge[398] in types (foundation to be discussed later (➜ p.451)).

[398] The Isabelle type system records for any type variable what class constraints (➜ p.184) there are for this type variable. These class constraints may arise from the types of the constants used in an expression, or they may be given explicitly by the user in a goal. E.g. one might type

```
Goal "(x::'a::order)<y ==> x<=y";
```

to specify that `x` must be of a type in the type class `order`.

The axioms of an axiomatic class can only be applied if any constant declared in the axiomatic class (or a syntactic superclass) is applied to arguments of a type in the axiomatic class. E.g. `order_refl` (➜ p.435) can only be used to prove $y <= y$ if the type of y is in the type class `order`.

In this sense the type information (y is of type in class `order`) is semantic knowledge ($y <= y$ holds).

19 Sets

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library (➜ p.427).

- Orders (➜ p.431)
- Sets
- Functions (➜ p.451)
- (Least) fixpoints and induction (➜ p.467)
- (Well-founded) recursion (➜ p.497)
- Arithmetic (➜ p.545)
- Datatypes (➜ p.568)

Set.thy

```
theory Set = HOL:
typedecl '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 (➜ p.417) to Abs_{set} (➜ p.418) and Rep_{set} (➜ p.418).

Sets Are a Special Case

Recall that the **typedef** (➜ p.422) syntax is based on set comprehension (➜ p.422). Therefore, sets are a special case of type definitions.

In deviation from our conservative approach (➜ p.349), sets are axiomatized as follows[399]:

```
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[400] to the type definition scheme (➜ p.410).

[399] Theorems marked with `[iff]` and `[simp]` flags are automatically added to the simplifier (➜ p.315). Additionally `[iff]` marked theorems are added to the classical reasoner (➜ p.284).

[400] We earlier (➜ p.417) presented a definition of α *set* according to the scheme of type definitions (➜ p.410). However, in Isabelle/HOL (`Set.thy` (➜ p.418)), it is not done exactly like that. The reason lies in the special set-based syntax (➜ p.422) used for type definitions.

The type α *set* is defined in Isabelle/HOL in a way which essentially corresponds to the type definition scheme, but is different in the technical details. In particular, there are no constants Abs_{set} and Rep_{set}. Instead, we have *Collect* (➜ p.418) and the ∈-sign (➜ p.418). We will now explain how.

Concerning Abs_{set}, there is no worry, since it corresponds exactly to *Collect* (➜ p.418).

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 (➜ p.355) here but only since it is more concise.

In what follows, recall that

$$\{x \mid f\,x\} = \mathit{Collect}\ (\text{➜ p.418})\ f = \mathit{Abs}_{\mathit{set}}\,f$$

$\mathit{Rep}_{\mathit{set}}$ is related to the $\in$-sign (➜ p.418) via

$$x \in A \;=\; (\mathit{Rep}_{\mathit{set}}\ A)\ x$$

Let us see that this setup is equivalent to the scheme of type definitions (➜ p.410). There are two axioms in Set.thy (➜ p.418):

```
axioms
  mem_Collect_eq [iff]: "(a : {x. P(x)}) = P(a)"
  Collect_mem_eq [simp]: "{x. x:A} = A"
```

We translate these axioms using the definitions (➜ p.417):

$$\begin{array}{l}
a \in \{x \mid P\,x\} = P\,a \leadsto \\
a \in (\mathit{Collect}\,P) = P\,a \leadsto \\
a \in (\mathit{Abs}_{\mathit{set}}\,P) = P\,a \leadsto \\
\mathit{Rep}_{\mathit{set}}(\mathit{Abs}_{\mathit{set}}\,P)\,a = P\,a \leadsto \\
\mathit{Rep}_{\mathit{set}}(\mathit{Abs}_{\mathit{set}}\,P) = P
\end{array}$$

The last step uses extensionality (➜ p.350).

`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

Now the second one:

$$\{x \mid x \in A\} = A \rightsquigarrow$$
$$\{x \mid (\mathit{Rep}_{\mathit{set}}\, A)\, x\} = A \rightsquigarrow$$
$$\mathit{Collect}(\mathit{Rep}_{\mathit{set}}\, A) = A$$

Ignoring some universal quantifications (these are implicit in Isabelle), these are the isomorphy axioms for *set* (➜ p.415).

`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 <=[401] instead of ⊆!

Nice syntax:

`UN` $x : A. S\,x$ or $\bigcup_{x \in A} . S\,x$ for `UNION` $A\ S$

`INT` $x : A. S\,x$ or $\bigcap_{x \in A} . S\,x$ for `INTER` $A\ S$

[401] Sets are an instance of the type class `ord` (➜ p.431), where the generic constant <= is the subset relation in this particular case.

In fact, the subset relation is reflexive, transitive and antisymmetric, and so sets are an instance of the axiomatic class (➜ p.185) `order`. This is non-obvious and must be proven, which is done not in `Set.thy` itself but in `Fun.thy`, later (➜ p.451). 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 (➜ p.438) 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`	$x \in \texttt{UNIV}$
`subset_UNIV`	$A \subseteq \texttt{UNIV}$
`empty_subsetI`	$\{\} \subseteq A$
`Pow_iff`	$(A \in Pow\,B) = (A \subseteq B)$
`IntI`	$[\![c \in A; c \in B]\!] \Longrightarrow c \in A \cap B$

More Theorems (➜ p.438) in `Set.thy`

`insert_iff`	$(a \in \mathit{insert}\, b\, A) = (a = b \vee a \in A)$
`image_Un`	$f`(A \cup B) = f`A \cup f`B$
`Inter_lower`	$B \in A \Longrightarrow \bigcap A \subseteq B$
`Inter_greatest`	$(\bigwedge X. X \in A \Longrightarrow C \subseteq X) \Longrightarrow C \subseteq \bigcap A$

19.1 Summary on Sets

Rich and powerful set theory available in HOL:

- No problems with consistency (➜ p.418)
- Weaker than ZFC (➜ p.323) (due to the type system): there is no "union of sets[402]"; but: complement-closed[403]
- Good mechanical support (➜ p.274) for many set tautologies (`Fast_tac` (➜ p.297), `fast_tac set_cs`, `fast_tac eq_cs`, . . . `simp_tac set_ss` (➜ p.317) . . .)

[402] In typed set theory (what we have here in HOL), it is not possible to form the union of two sets of different type. This is in contrast to ZFC (➜ p.323).

[403] The complement of a typed set A, i.e.

$$\{x \mid x \notin A\}$$

is again a set, whose type is the same as the type of A. In ZFC (➜ p.323), the complement construction is not generally allowed since it opens the door to Russell's Paradox (➜ p.139).

- 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 (➜ p.427).

- Orders (➜ p.431)
- Sets (➜ p.440)
- Functions
- (Least) fixpoints and induction (➜ p.467)
- (Well-founded) recursion (➜ p.497)
- Arithmetic (➜ p.545)
- Datatypes (➜ p.568)

`Fun.thy`

The theory `Fun.thy`[404] defines some important notions on functions, such as concatenation, the identity function, the image of a function, etc.

We look at it briefly.

[404] 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/`

`Fun.thy` builds on `Set.thy` (➜ p.418), 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:

```
definition
 id :: "'a => 'a"
where "id == %x. x"

 comp :: "['b => 'c, 'a => 'b, 'a] => 'c"
"f o g == %x. f(g(x))"
```

Recall **`definition`** (➜ p.423) syntax.

Instantiating an Axiomatic Class

Sets are partial orders (➜ p.113): `set` is an instance of the axiomatic class `order` (➜ p.433).

For some reason (➜ p.446), this is proven in `Fun.thy`.

```
instance set :: (type) order
  by (intro_classes,
      (assumption | rule subset_refl
        subset_trans subset_antisym psubset_eq)+)
```

- Axiomatic classes result in proof obligations[405].
- These are discharged[406] whenever instance is stated.
- Type-checking (➜ p.439) has access to the established properties.

[405] To claim that a type is an instance of an axiomatic class (➜ p.185), 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.

[406] The Isabelle mechanism is such that the line

```
instance set :: (type) order
  by (intro_classes,
     (assumption | rule
     subset_refl subset_trans subset_antisym psubset_eq)+)
```

instructs Isabelle to prove the axioms using the previously proven theorems (➜ p.438) `subset_refl`, `subset_trans`, `subset_antisym`, and `psubset_eq`.

20.1 Conclusion of Orders, Sets, Functions

- Theory says: conservative extensions can be used (➜ p.398) to build consistent libraries.
- Sets as one important package (➜ p.440) of Isabelle/HOL library:
  - Set theory is typed, but very rich and powerfully supported.
  - Sets are instance of `ord` (➜ p.431) and `order` (➜ p.455) type class, demonstrates type classes as structuring mechanism in Isabelle.
- Will see more examples: Isabelle/HOL contains some 10000 `thm` (➜ p.293)'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.427).

- Orders (➜ p.431)
- Sets (➜ p.440)
- Functions (➜ p.451)
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic (➜ p.545)
- Datatypes (➜ p.568)

Recursion Based on Set Theory

Current stage of our course:

- On the basis of conservative extensions (➜ p.398), set theory (➜ p.440) 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

$$\frac{}{Y = \lambda F.F(YF)}\,\text{fix}$$

This axiom is not a constant definition[407].

Then we could easily derive

$$\forall F^{\alpha\to\alpha}.Y\,F = F\,(Y\,F)^{408}.$$

- Why are we interested in Y?
- What is the problem with such a definition?

[407] The axiom

$$Y = \lambda F.F(YF)$$

is not a constant definition (➜ p.404), since Y occurs again on the right-hand side.

[408] In 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 equations[409])?

- Recursively defined (➜ p.525) functions are solutions of such equations (example: *fac*[410]).
- Inductively defined (➜ p.488) sets are solutions of such

[409] By a recursive equation, we mean an equation of the form

$$f = e$$

where f occurs in e. A fortiori, such an equation does not qualify as constant definition (➜ p.404).

[410] In the following explanations, any constants like 1 or $+$ or **if-then-else** are intended to have their usual meaning.

A fixpoint combinator (➜ p.460) 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

$$\mathit{fac}\ n = (\textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathit{fac}\ (n-1)).$$

We now massage this equation a bit. Doing λ-abstraction (➜ p.153) on both sides we get

$$\lambda n.\ \mathit{fac}\ n = (\lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathit{fac}(n-1))$$

which is the η-conversion (➜ p.158) of

$$\mathit{fac} = (\lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathit{fac}(n-1))$$

which in turn is a β-reduction (➜ p.153) of

$$\mathit{fac} = (\underline{(\lambda f.\, \lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * f(n-1))}\ \mathit{fac}) \qquad (3)$$

We are looking for a solution to (3). We abbreviate the underlined expression by *Fac*. We claim $\mathit{fac} = Y\ \mathit{Fac}$, i.e., it is a solution to (3). Simply replacing *fac* with $Y\ \mathit{Fac}$ in (3) we get

$$Y\ \mathit{Fac} = \mathit{Fac}\ (Y\ \mathit{Fac})$$

equations (example: *Finites*[411], the set of all finite sets).

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).

[411] We want to define the set of all finite sets (of a given type τ), call it *Finites*.

How do you construct the set of all finite sets? The following pseudo-code suggests what you have to do:

```
$S := \{\{\}\};$
forever do
   foreach $a :: \tau$ 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 type τ, it will indeed reach a fixpoint, i.e., adding new sets won't change anything).

Generally (even if τ is infinite), *Finites* is a set such that adding new sets as suggested by the pseudo-code won't change anything. Written as recursive equation:

$$\mathit{Finites} = \{\{\}\} \cup \bigcup x :: \tau.((\texttt{insert}\ (\text{➜ p.445})\ x)\ `\ (\mathit{Finites}))$$

Recall that ‘ is nice syntax for *image* (➜ p.447), defined in `Set.thy` (➜ p.418).

The above is a β-reduction (➜ p.153) of

$$\mathit{Finites} = \\ (\underline{\lambda X.\ \{\{\}\} \cup \bigcup x :: \tau.((\texttt{insert}\ (\text{➜ p.445})\ x)\ `\ X)})\ (\mathit{Finites}) \qquad (4)$$

We are looking for a solution to (4). We abbreviate the underlined expression by *FA*. We claim

$$\mathit{Finites} = Y\ \mathit{FA},$$

i.e., it is a solution to (4). Simply replacing *Finites* with $Y\ \mathit{FA}$ in (4) we get

$$Y\ \mathit{FA} = \mathit{FA}(Y\ \mathit{FA}),$$

recursions. With Y, recursive axioms can be converted[412] into constant definitions (➜ p.404).

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.

[412] Any recursive function can be defined by an expression (functional) which is not itself recursive, but instead relies on the recursive equation defining Y.

Consider *fac* (➜ p.461) or *Finites* as an example.

What's the Problem with such an Axiom?

Such a definition would lead to inconsistency (➜ p.401).

This is not surprising because not all functions have a fixpoint.

Therefore we only consider special forms (➜ p.543) of fixpoint combinators.

We consider two approaches: Least fixpoints (➜ p.467) (Tarski) and well-founded (➜ p.497) orderings.

22 Least Fixpoints

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library (➜ p.427).

- Orders (➜ p.431)
- Sets (➜ p.440)
- Functions (➜ p.451)
- (Least) fixpoints and induction
- (Well-founded) recursion (➜ p.497)
- Arithmetic (➜ p.545)
- Datatypes (➜ p.568)

22.1 First Approach: Least Fixpoints (Tarski)

- Recall: (➜ p.460) We would like to define $Y = \lambda F.F(YF)$, where F is of arbitrary type $\alpha \rightarrow \alpha$, but we must not (➜ p.466).
- Restriction: F (➜ p.460) is of set type (α *set* $\rightarrow \alpha$ *set*).
- Instead of Y define *lfp* by an equation which is not recursive.
- *lfp* is fixpoint combinator, but only under additional condition that F is monotone[413], and: this is not obvious (requires non-trivial proof)!

This leads us towards recursion and induction (➜ p.488).

[413] A function f is monotone w.r.t. a partial order (➜ p.113) $\leq$ 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.thy[414]

```
Lfp = Product_Type +
constdefs
 lfp :: ['a set => 'a set] => 'a set
 "lfp(f) == Inter({u. f(u) <= u})"
```

- `=>` is function type arrow (➜ p.184).
- `<=` (“$\subseteq$” (➜ p.446)) is a partial order (➜ p.113).
- `Inter` (“$\bigcap$”) (➜ p.447) gives a “minimum”: $\forall A \in S.(\bigcap S) \subseteq A$. Note that $\{u|f(u) \subseteq u\} \neq \emptyset$, because $f($`UNIV`$) \subseteq$ `UNIV`.

[414] These 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/`

Is it a Fixpoint?

We have

$$\mathit{lfp}(f) := \bigcap\{u|f(u) \subseteq u\}$$

Definition of *lfp* is conservative (➜ p.404). That's fine. But is it a fixpoint combinator? (➜ p.477)

22.2 Tarski's Fixpoint Theorem

Theorem (Tarski):
If f is monotone (➜ p.469), then $\mathit{lfp}\ f = f\ (\mathit{lfp}\ f)$.

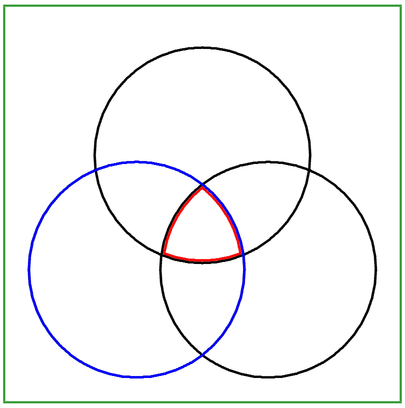
In Isabelle, the theorem is shown in `Lfp.ML` (➜ p.470) and called `lfp_unfold`.

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 *lfp* $f \subseteq A$.



The box denotes "the set" α[415]. The three circles[416] denote the sets A for which $f\ A \subseteq A$.

By definition (➜ p.470), *lfp* f is the intersection.

Pick an A for which $f\ A \subseteq A$. Clearly, *lfp* $f \subseteq A$.

Or as proof tree (➜ p.479).

[415] 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` (➜ p.445). `UNIV` of type τ `set` is the set containing all terms of type τ.

[416] In general, needless to say, there could be any number of such sets, but the picture is to be understood in the sense that the three circles are all the sets A with the property $f\ A \subseteq A$.

Tarski's Fixpoint Theorem (2)

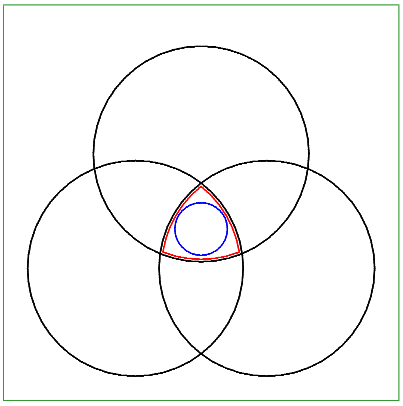
Claim 2 ("*lfp* greatest"): For all A, if for all U, $f\ U \subseteq U$ implies $A \subseteq U$, then $A \subseteq \mathit{lfp}\ f$.

The three circles denote the sets U for which $f\ U \subseteq U$.

By hypothesis, $A \subseteq U$ for each U (1st, 2nd, 3rd $\ldots$).

By definition (➜ p.470), $\mathit{lfp}\ f$ is the intersection.

Clearly, $A \subseteq \mathit{lfp}\ f$.



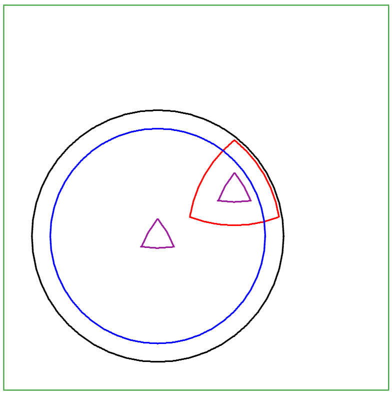
Or as proof tree (➜ p.480).

Tarski's Fixpoint Theorem (3)

Claim 3: If f is monotone (➜ p.469) then $f(\mathit{lfp}\ f) \subseteq \mathit{lfp}\ f$.

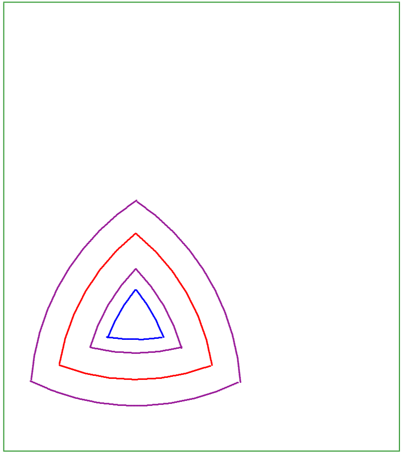
First show Claim 3*: $\underline{f\ U \subseteq U}$ implies $f(\mathit{lfp}\ f) \subseteq U$.

Let the circle be such a U. By Claim 1 (➜ p.473), $\mathit{lfp}\ f \subseteq U$. $f\ U \subseteq U$ ($\underline{\text{hypothesis}}$). $f(\mathit{lfp}\ f) \subseteq f\ U$ (monotonicity (➜ p.469)). $f(\mathit{lfp}\ f) \subseteq U$ (transitivity (➜ p.451) of $\subseteq$). Claim 3* shown. By Claim 2 (➜ p.474) (letting $A := f(\mathit{lfp}\ f)$), $f(\mathit{lfp}\ f) \subseteq \mathit{lfp}\ f$.



Tarski's Fixpoint Theorem (4)

Claim 4: If f is monotone (➜ p.469) then $\mathit{lfp}\ f \subseteq f(\mathit{lfp}\ f)$.



By Claim 3 (➜ p.475), $f(\mathit{lfp}\ f) \subseteq \mathit{lfp}\ f$.
By monotonicity (➜ p.469), $f(f(\mathit{lfp}\ f)) \subseteq f(\mathit{lfp}\ f)$.
By Claim 1 (➜ p.473) (letting $A := f(\mathit{lfp}\ f)$), $\mathit{lfp}\ f \subseteq f(\mathit{lfp}\ f)$.

Or as proof tree (➜ p.482).

Tarski's Fixpoint Theorem: QED

Claim 3 (➜ p.475) ($\mathit{lfp}\ f \subseteq f(\mathit{lfp}\ f)$) and Claim 4 (➜ p.476) ($f(\mathit{lfp}\ f) \subseteq \mathit{lfp}\ f$) together give the result:

> If f is monotone, then $\mathit{lfp}\ f = f\ (\mathit{lfp}\ f)$.

So under appropriate conditions, *lfp* is a fixpoint combinator (➜ p.461).

We will later reuse Claim 1 (➜ p.473).

Alternative: A Natural-Deduction Style Proof

The proof can also be presented in natural deduction style (➜ p.23).

Tarski's Fixpoint Theorem (1)

Claim 1 ("*lfp* lower bound"): If $f\ A \subseteq A$ then $lfp\ f \subseteq A$.

$$\cfrac{\cfrac{\cfrac{\cfrac{[f\ A \subseteq A]^1}{A \in \{u. fu \subseteq u\}}\ \text{CollectI (➜ p.448)}}{\bigcap\{u. fu \subseteq u\} \subseteq A}\ \text{Inter\_lower (➜ p.448)}}{lfp\ f \subseteq A}\ \text{Def. } lfp \text{ (➜ p.470)}}{f\ A \subseteq A \to lfp\ f \subseteq A}\ \to\text{-}I \text{ (➜ p.360)}^1$$

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 $A \subseteq \textit{lfp}\ f$.

$$\dfrac{\dfrac{\dfrac{\dfrac{[\forall x. fx \subseteq x \to A \subseteq x]^1}{\forall x. x \in \{u. fu \subseteq u\} \to A \subseteq x}\ \textit{subst}\ (\text{➜ p.360}), \text{CollectI}\ (\text{➜ p.448})}{A \subseteq \bigcap\{u. fu \subseteq u\}}\ \text{Inter\_greatest (➜ p.448)}}{A \subseteq \textit{lfp} f}\ \text{Def. } \textit{lfp}\ (\text{➜ p.470})}{(\forall x. fx \subseteq x \to A \subseteq x) \to A \subseteq \textit{lfp} f}\ \to\text{-}I\ (\text{➜ p.360})^1$$

Tarski's Fixpoint Theorem (3)

Claim 3: If f is monotone (➜ p.469) then $f(\mathit{lfp}\ f) \subseteq \mathit{lfp}\ f$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\mathit{mono}\ f]^1 \quad \dfrac{[fx \subseteq x]^2}{\mathit{lfp}\ f \subseteq x}}{f(\mathit{lfp}\ f) \subseteq f\ x} \qquad [fx \subseteq x]^2}{f(\mathit{lfp}\ f) \subseteq x}\ \text{order\_trans (➜ p.435)}}{\forall x. fx \subseteq x \to f(\mathit{lfp}\ f) \subseteq x}\ \forall\text{-}I\text{ (➜ p.355)}, \to\text{-}I\text{ (➜ p.360)}^2}{f(\mathit{lfp}\ f) \subseteq \mathit{lfp}\ f}\ \text{lfp\_greatest (➜ p.480)}, \to\text{-}E\text{ (➜ p.360)}}{\mathit{mono}\ f \to f(\mathit{lfp}\ f) \subseteq \mathit{lfp}\ f}\ \to\text{-}I\text{ (➜ p.360)}^1$$

Tarski's Fixpoint Theorem (4)

Claim 4: If f is monotone (➜ p.469) then $\mathit{lfp}\ f \subseteq f(\mathit{lfp}\ f)$.

$$\dfrac{\dfrac{\dfrac{[\mathit{mono}\ f]^1 \quad \dfrac{[\mathit{mono}\ f]^1}{f(\mathit{lfp}\ f) \subseteq \mathit{lfp}\ f}\text{Claim 3 (➜ p.481), }\rightarrow\text{-}E\text{ (➜ p.360)}}{f(f(\mathit{lfp}\ f)) \subseteq f(\mathit{lfp}\ f)}\text{monoD (➜ p.438)}}{\mathit{lfp}\ f \subseteq f(\mathit{lfp}\ f)}\text{lfp\_lowerbound (➜ p.479), }\rightarrow\text{-}E\text{ (➜ p.360)}}{\mathit{mono}\ f \rightarrow \mathit{lfp}\ f \subseteq f(\mathit{lfp}\ f)}\rightarrow\text{-}I\text{ (➜ p.360)}^1$$

Completing Proof Tree

$$\dfrac{\dfrac{\dfrac{[\mathit{mono}\ f]^1}{\mathit{lfp}\ f \subseteq f(\mathit{lfp}\ f)}\text{Claim 4 (➜ p.482)} \qquad \dfrac{[\mathit{mono}\ f]^1}{f(\mathit{lfp}\ f) \subseteq \mathit{lfp}\ f}\text{Claim 3 (➜ p.481}}{\mathit{lfp}\ f = f(\mathit{lfp}\ f)}\text{equalityI}}{\mathit{mono}\ f \to \mathit{lfp}\ f = f(\mathit{lfp}\ f)}\to\text{-}I\text{ (➜ p.360)}^1$$

22.3 Induction Based on `Lfp.thy`

Theorem (lfp induction):

If

- f is monotone (➜ p.469), and
- $f(\mathit{lfp}\ f \cap \{x \mid P\,x\}) \subseteq \{x \mid P\,x\}$,

then $\mathit{lfp}\ f \subseteq \{x \mid P\,x\}$.

In Isabelle[417], it is called `lfp_induct`:

$$[\![a \in \mathit{lfp}\ f; \mathit{mono}\ f; \bigwedge x. x \in f(\mathit{lfp}\ f \cap \{x. P\,x\}) \Longrightarrow P\,x]\!]$$
$$\Longrightarrow P\,a$$

We now show the theorem similarly as Tarski's Theorem (➜ p.472).

[417] The theorem is phrased a bit differently in the "mathematical" version we give here and in the Isabelle version (see `Lfp.ML` (➜ p.470)). This is convenient for the graphical illustration of the proof.

The "mathematical phrasing" corresponding closely to the Isabelle version would be the following:

Theorem (Induct (alternative)):

If

- $a \in \mathit{lfp}\ f$, and
- f is monotone (➜ p.469), and
- for all x, $x \in f(\mathit{lfp}\ f \cap \{x \mid P\,x\})$ implies $P\ x$

then $P\ a$ holds.

Other phrasings, which may help to get some intuition about the theorem:

Theorem (Induct (alternative)):

If

- $a \in \mathit{lfp}\ f$, and
- f is monotone (➜ p.469), and
- $f(\mathit{lfp}\ f \cap \{x \mid P\,x\}) \subseteq \{x \mid P\,x\}$

then $P\ a$ holds.

Theorem (Induct (alternative)):
If

- f is monotone (➜ p.469), and
- $f(\mathit{lfp}\ f \cap \{x \mid P\,x\}) \subseteq \{x \mid P\,x\}$

then for all x in $\mathit{lfp}\ f$, we have $P\ x$.

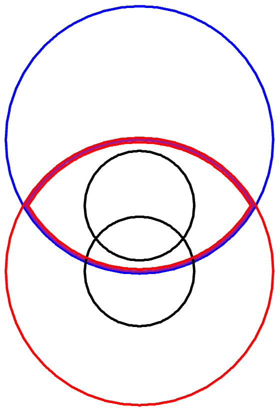
Circles denote $\mathit{lfp}\ f$ and $\{x \mid P\,x\}$.

By monotonicity[418], $f(\mathit{lfp}\ f \cap \{x \mid P\,x\}) \subseteq f(\mathit{lfp}\ f)$. By Tarski (➜ p.472), $\mathit{lfp}\ f = f(\mathit{lfp}\ f)$. Hence $f(\mathit{lfp}\ f \cap \{x \mid P\,x\}) \subseteq \mathit{lfp}\ f$.

By hypothesis (➜ p.484), $f(\mathit{lfp}\ f \cap \{x \mid P\,x\}) \subseteq \{x \mid P\,x\}$, and so we must adjust picture: $f(\mathit{lfp}\ f \cap \{x \mid P\,x\}) \subseteq \mathit{lfp}\ f \cap \{x \mid P\,x\}$.

By Claim 1[419], $\mathit{lfp}\ f \subseteq \mathit{lfp}\ f \cap \{x \mid P\,x\}$ and so[420] $\mathit{lfp}\ f = \mathit{lfp}\ f \cap \{x \mid P\,x\}$.

Conclusion: $\mathit{lfp}\ f \subseteq \{x \mid P\,x\}$.



[418] $\mathit{lfp}\ f \cap \{x \mid P\,x\} \subseteq \mathit{lfp}\ f$, so by monotonicity (➜ p.484), $f(\mathit{lfp}\ f \cap \{x \mid P\,x\}) \subseteq f(\mathit{lfp}\ f)$.

[419] We have just seen $f(\mathit{lfp}\ f \cap \{x \mid P\,x\}) \subseteq \mathit{lfp}\ f \cap \{x \mid P\,x\}$.

By Claim 1 (➜ p.473)

$$\text{If } f\ A \subseteq A \text{ then } \mathit{lfp}\ f \subseteq A$$

(setting $A := \mathit{lfp}\ f \cap \{x \mid P\,x\}$), this implies $\mathit{lfp}(f) \subseteq \mathit{lfp}\ f \cap \{x \mid P\,x\}$.

[420] We have $\mathit{lfp}\ f \cap \{x \mid P\,x\} \subseteq \mathit{lfp}(f)$ and $\mathit{lfp}(f) \subseteq \mathit{lfp}\ f \cap \{x \mid P\,x\}$, and so $\mathit{lfp}(f) = \mathit{lfp}\ f \cap \{x \mid P\,x\}$ by the antisymmetry of $\subseteq$ (➜ p.451).

Approximating Fixpoints

Looking ahead: Suppose we have the set $\mathbb{N}$ of natural numbers (the type is formally introduced later (➜ p.554)). The theorem `approx`

$(\forall S.\ f(\bigcup$ (➜ p.447)$S) = \bigcup(f$ ‘ (➜ p.447) $S)) \Longrightarrow$
$\bigcup_{n\in\mathbb{N}}(f^n\{\})) = \mathit{lfp}\ f$

shows a way of approximating *lfp*, which is important for algorithmic solutions[421] (e.g. in program analysis).

There will be an exercise on this.

[421] The theorem The theorem `approx`

$(\forall S.\ f(\bigcup$ (➜ p.447)$S) = \bigcup(f$ ‘ (➜ p.447) $S)) \Longrightarrow$
$\bigcup_{n\in\mathbb{N}}(f^n\{\})) = \mathit{lfp}\ f$

says that under a certain condition, *lfp* f can be computed by applying f to the empty set over and over again:

- although the expression uses the union over all natural numbers, which is an infinite set, this can sometimes effectively be computed. Under certain conditions, there exists a k such that $f^k\ \{\} = f^{k+1}\{\}$.
- Even if $\bigcup n \in \mathbb{N}.f^n\ \{\}$ cannot be effectively computed, it can still be approximated: for any k, we know that $\bigcup i \leq k.f^i\ \{\} \subseteq \bigcup n \in \mathbb{N}.f^n\ \{\}$.

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 (➜ p.461) $S = F(S)$.

Induction and recursion are two faces of the same coin.

`Lfp.thy` for Inductive Definitions (➜ p.470)

Least fixpoints are for building inductive definitions of sets in a definitional way[422]: $S := \mathit{lfp}\ F$.

This is obviously (➜ p.470) well-defined, so why this fuss about monotonicity (➜ p.469) and Tarski (➜ p.472)?

Tarski (➜ p.472) allows us to exploit the equation $\mathit{lfp}\ f = f(\mathit{lfp}\ f)$ in proofs about S! That's what *lfp* is all about.

[422] Recall why we were interested (➜ p.461) in fixpoints.

The problem with Y (➜ p.466) is that it leads to inconsistency (➜ p.401) (and of course (➜ p.405), the definition of Y is not a constant definition (➜ p.404)/conservative extension.).

The definition of *lfp* is conservative.

And in appropriate situations, it can be used to define recursive functions.

Compared to Y (➜ p.457), the type of *lfp* is restricted (➜ p.469).

This restriction means that there is no obvious way to use *lfp* for defining recursive numeric functions such as *fac* (➜ p.461).

Example: Finite Sets

The set of all finite sets (of a certain type, of course) is defined by:

$$Finites = lfp\ F$$

where $F \equiv \lambda S.\ \{x \mid x = \{\} \vee (\exists A\ a.\ x = insert\ a\ A \wedge A \in S)\}$.

Thus we can do using *lfp* what we would have wanted to do using Y (➜ p.463).

To show: F is monotone[423]!

We next consider the Isabelle support for this ...

[423] This proof is of course done in Isabelle.

22.4 The Package for Inductive Sets

Since monotonicity proofs can be automated, Isabelle has special proof support for inductive definitions.

Consider again the example of finite sets. In `Finite_Set.thy`[424], we have:

```
inductive "Finites"
  intros
    emptyI [simp, intro!]: "{} : Finites"
    insertI [simp, intro!]: "A : Finites ==>
          insert a A : Finites"
```

[424] 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/`

Translation of `inductive`

The Isabelle mechanism[425] of interpreting the keyword `inductive` translates this into the following definition: $\mathit{Finites} = \mathit{lfp}\ F$ where

$$F \equiv \lambda S.\ \{x \mid x = \{\} \vee (\exists A\ a.\ x = \mathit{insert}\ a\ A \wedge A \in S)\}$$

as stated above.

In `Finite_Set.thy` (➜ p.491) a constant *Finites* is defined. It has polymorphic type $\alpha\,\mathit{set}\,\mathit{set}$. We have $A \in \mathit{Finites}$ 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.331) of *Finites* of type $\tau\,\mathit{set}\,\mathit{set}$ containing all finite sets of element type τ.

You can see this by typing in your proof script:

```
open Finites;
defs;
```

Talking (ML-)technically, `Finites` is a structure (➜ p.493) (module), and `defs` is a value (component) of this structure (➜ p.493).

As a sanity-check, consider the type (➜ p.330) of this expression. The expression $\mathit{insert}\,a\,A$ forces A to be of type $\tau\ \mathit{set}$ for some τ and a to be of type τ. Next, $\mathit{insert}\,a\,A$ is of type $\tau\ \mathit{set}$, and hence x is also of type $\tau\ \mathit{set}$. Moreover, the expression $A \in S$ forces S to be of type $\tau\ \mathit{set}\ \mathit{set}$. The expression $\{x \mid x = \{\} \vee (\exists A\, a.\ x = \mathit{insert}\, a\, A \wedge A \in S)\}$ is of type $\tau\ \mathit{set\ set}$. Next, G is of type $\tau\ \mathit{set\ set} \to \tau\ \mathit{set\ set}$, and so finally, *Finites* is of type $\tau\ \mathit{set\ set}$. 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 : Finites"
```

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.

Package relies on proven lemma[426] `lfp_unfold` (➜ p.472).

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` (➜ p.293) called C`_def` which is the definition of C.
- Sometimes (when a newer format is used), it will insert the definition of C into a database which can be accessed by a function called `thm` taking a string as argument. In this case, not C`_def` would be the definition of C, but rather

  `thm` "C`_def`"

You should be aware of such problems, but we do not treat them in this course.

[426] If you look around in the `ML`-files of the Is-

Technical Support for Inductive Definitions

Support important in practice since many constructions are based on inductively defined sets (datatypes (➜ p.568), . . .). Support provided for:

- Automatic proof of monotonicity
- Automatic proof of induction rule (➜ p.484), for example[427]:

  $$[\![xa \in \mathit{Finites}; P\ \{\}; \bigwedge a\, b. [\![b \in \mathit{Finites}; P\, b]\!] \Longrightarrow P\, (\mathit{insert}\, a\, b)]\!] \Longrightarrow P\, xa$$

---

abelle/HOL library, you might not find any uses of **lfp_unfold** (➜ p.472), so you may wonder: why is it important then? But you must bear in mind that the package for inductive sets (➜ p.491) 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.

[427] The theorem

$$[\![xa \in \mathit{Finites}; P\ \{\}; \bigwedge ab. [\![b \in \mathit{Finites}\ A; P\, b]\!] \Longrightarrow P\, (\mathit{insert}\, a\, b)]\!] \Longrightarrow P\, xa$$

is an instance of the general induction scheme (➜ p.484). That is to say, if we take the general induction scheme **lfp_induct** (➜ p.484)

$$[\![a \in \mathit{lfp}\ f; \mathit{mono}\, f; \bigwedge x. x \in f(\mathit{lfp}\ f \cap \{x. P\, x\}) \Longrightarrow P\, x]\!] \Longrightarrow P\, a$$

and instantiate f to $\lambda S.\ \{x \mid x = \{\} \vee (\exists A\ \ a.\ x = \mathit{insert}\ a\ A \wedge A \in S)\}$ (➜ p.490) then some massaging using

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 (➜ p.466), but we have, by conservative extension (➜ p.404), least fixpoints for defining sets.

There is an induction scheme (lfp induction (➜ p.484)) for proving theorems about an inductively defined set.

Restriction of F to set type (➜ p.469) ($\alpha\ set \to \alpha\ set$) means that least fixpoints are not generally suitable for defining functions ...

---

the definitions will give us the first theorem.

Note here that monotonicity has disappeared from the assumptions. This is because the monotonicity of F (➜ p.490) is shown by Isabelle once and for all. This is one aspect of what we mean by special proof support for inductive definitions (➜ p.491).

The least fixpoint of the functional is *Finites* (the set of finite sets) in this case.

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.427).

- Orders (➜ p.431)
- Sets (➜ p.440)
- Functions (➜ p.451)
- (Least) fixpoints and induction (➜ p.467)
- (Well-founded) recursion
- Arithmetic (➜ p.545)
- Datatypes (➜ p.568)

Well-Founded Recursion

After least fixpoints (➜ p.467), 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 (➜ p.420)), such as converse, composition, image of a set and a relation, the identity relation, ...

These are provided by `Relation.thy`[428].

[428] 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.451).

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`[429].

How would you define those inductively, ad-hoc?[430]

[429] 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/`

[430] r^* is the smallest set such that:

- Id (➜ p.500) $\subseteq r^*$;
- if $r' \subseteq r^*$ then $r' \cup r \circ r' \subseteq r'$.

Or, in line with the schema for inductive definitions (➜ p.488):

- $\emptyset \subseteq r^*$;
- if $r' \subseteq r^*$ then $(\lambda s.Id$ (➜ p.500) $\cup (r \circ s))r' \subseteq r^*$.

The latter form corresponds to the definition in `Transitive_Closure.thy` (➜ p.502).

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`[431].

```
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? r is well-founded if well-founded (Noetherian) induction based on r is a valid proof scheme[432].

[431] 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/`

In older versions the file used to be called `WF.thy`.

[432] For a moment, forget everything you have ever heard about proofs using induction! The definition of *wf* has the form

$$wf(r) \equiv \forall P.\phi(r,P) \to \forall x.P(x)$$

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 (➜ p.404) (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 $\forall P.\phi(r,P) \to \forall x.P(x)$ holds, and show $\phi(r,P)$.

Once again, this method is correct regardless of what ϕ is. Forget about induction!

But how is that possible? How is it ensured that only true statements can be proven, if the method is correct for any ϕ no matter how strange?

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 $\forall P.\phi(r, P) \to \forall x.P(x)$ or $\phi(r, P)$.

Example: Is $\emptyset$ well-founded[433]? $<$ on the integers[434]?

[433] The definition of *wf* is:

$$wf(r) \equiv (\forall P.(\forall x.(\forall y.(y,x) \in r \to P(y)) \to P(x)) \to (\forall x.P(x)))$$

Let's instantiate r to $\emptyset$.

$$wf(\emptyset) \equiv (\forall P.(\forall x.(\forall y.(y,x) \in \emptyset \to P(y)) \to P(x)) \to (\forall x.P(x)))$$
$$wf(\emptyset) \equiv (\forall P.(\forall x.(\forall y.\mathit{False} \to P(y)) \to P(x)) \to (\forall x.P(x)))$$
$$wf(\emptyset) \equiv (\forall P.(\forall x.(\forall y.\mathit{True}) \to P(x)) \to (\forall x.P(x)))$$
$$wf(\emptyset) \equiv (\forall P.(\forall x.\mathit{True} \to P(x)) \to (\forall x.P(x))) \qquad (*)$$
$$wf(\emptyset) \equiv (\forall P.\mathit{True})$$
$$wf(\emptyset) \equiv \mathit{True}$$

So the empty set is well-founded.

Note the line marked $(*)$. Note that the well-foundedness of $\emptyset$ is useless for proving any P, because the induction step degenerates to the proof obligation $\forall x.P(x)$.

[434] Let us check (in an intuitive way) whether $<$ on the in-

Intuition of Well-Foundedness

Intuition of *wf*: All descending chains are finite.

---

tegers is well-founded. So we must check whether

$$(\forall P.(\forall x.(\forall y.y < x \to P(y)) \to P(x)) \to (\forall x.P(x)))$$

holds. Instantiating P to $\lambda x.\mathit{False}$ we obtain

$$(\forall x.(\forall y.y < x \to \mathit{False}) \to \mathit{False}) \to (\mathit{False})$$

Now since for every x there exists a y with $y < x$, it follows that $(\forall y.y < x \to \mathit{False})$ is equivalent to *False* and hence we obtain

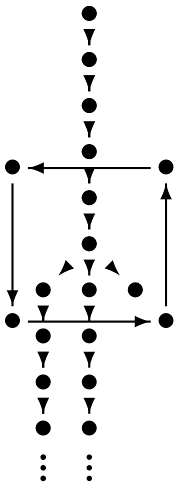
$$(\forall x.\mathit{False} \to \mathit{False}) \to (\mathit{False})$$

and thus

$$\mathit{False}$$

Thus, assuming that $<$ on the integers is well-founded, we derived a contradiction. You might think of $(\forall y.y < x \to \mathit{False})$ as being a conjunction containing infinitely many *False*s, and such a non-empty conjunction is *False*.

But: Cannot express infinity; must look for alternatives[435].



- Not symmetric: $(x,y) \in r \to (y,x) \notin r$?
- No cycles: $(x,x) \notin r^+$ (➜ p.499)?
- r has minimal element: $\exists x.\forall y.(y,x) \notin r$? Note: Trivial for $r = \emptyset$.
- Any subrelation must have minimal element: $\forall p.p \subseteq r \to \exists x.\forall y.(y,x) \notin p$? "Minimal element" badly formalized[436] (already in previous point).

What is different when we assume $<$ on the natural numbers? The difference is that it is not the case that for all x, we have that $(\forall y.y < x \to \mathit{False})$ is equivalent to *False*. Namely, for $x = 0$, we have $(\forall y.y < 0 \to \mathit{False})$ is equivalent to *True* because $y < 0$ is always *False*. Compared to the previous case, we have a conjunction consisting of only *True*s.

It turns out that when we do a proof using well-founded recursion on the natural numbers, for 0 there will be a non-trivial proof obligation, i.e., we will have to show $P(0)$.

[435] We will now try some ideas, work out their formalization as a formula, and then illustrate why the condition is either too weak or too strong, using an example. Finally, we will give the correct condition.

[436] In this attempt, we formalized the "minimal element in p" as an x such that there is no y with $(x,y) \in p$. But this is a bad formalization since an isolated element, i.e., one that is completely unrelated to p, or even to r, would meet the

A Characterization

All these attempts are just necessary but not sufficient conditions for well-foundedness.

The following theorem `wf_eq_minimal` gives a characterization of well-foundedness[437].:

$$wf\ r = (\forall Q\, x.\, x \in Q \to (\exists z \in Q. \forall y. (y, z) \in r \to y \notin Q))$$

Proof uses split =[438], `wf_def` (➜ p.504), rest routine.

Ergo: Definition of `wf` (➜ p.504) meets textbook definitions "every non-empty set Q has a minimal element in r".

---

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).

[437] The final condition

$$(\forall Q\, .\, x \in Q \to (\exists z \in Q. \forall y. (y, z) \in r \to y \notin Q))$$

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" (➜ p.504) definition is somewhat indirect since it defines well-foundedness by what one can do with it (➜ p.504).

[438] By this we simply mean to split a proof of $\phi = \psi$ into two proofs $\phi \Longrightarrow \psi$ and $\psi \Longrightarrow \phi$.

Alternative Characterization

Here is an alternative characterization (exercise):

$$(\forall r.r \neq \{\} \wedge r \subseteq p \rightarrow (\exists x \in \mathit{Domain}\, r.\forall y.(y, x) \notin r))$$

Let's see some theorems to confirm our intuition, including the characterization attempts just seen.

A Theorem[439] on the Empty Set

`wf_empty` $\mathit{wf}\,\{\}$

Proof sketch: `wf_empty`: substitute r into definition, simplify.

[439] The theorems (➜ p.438) 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.511) for Induction

By massage[440] of the definition of well-foundedness

$$\forall P.(\forall x.(\forall y.(y,x) \in r \to P\,y) \to P\,x) \to (\forall x.P\,x)$$

one obtains the theorem `wf_induct`

$$[\![wf\ r; \bigwedge x.\forall y.(y,x) \in r \to P\,y \Longrightarrow P\,x]\!] \Longrightarrow P\,a.$$

This is a form suitable for doing induction proofs in Isabelle.

[440] As far as the induction principle is concerned, `induct_wf` states the same as the very definition of `wf` (➜ p.504). All that happens is that some explicit universal object-level quantifiers are removed (➜ p.438) and the according variables are (implicitly) universally quantified on the meta-level, and some shifting (➜ p.438) from object-level implications to meta-level implications using `mp`. This is why we dare say "logical massage". See `Wellfounded_Recursion.ML` (➜ p.511).

Induction Theorem as Proof Rule

The Isabelle theorem `wf_induct` (➜ p.512)

$$[\![wf\;r; \bigwedge x. \forall y. (y,x) \in r \to P\,y \Longrightarrow P\,x]\!] \Longrightarrow P\,a.$$

as proof rule (➜ p.23):

$$\dfrac{wf\;r \qquad\qquad \begin{array}{c}[\forall y.(y,x) \in r \to P\,y]\\ \vdots \\ P\,x\end{array}}{P\,a}\;\texttt{wf\_induct}$$

A Theorem (➜ p.511) on Antisymmetry

`wf_not_sym` $\llbracket wf\ r; (a,x) \in r \rrbracket \Longrightarrow (x,a) \notin r$

Proof sketch:

$$\dfrac{wf\ r \qquad \begin{array}{c}[\forall y.(y,x) \in r \to (\forall z.(y,z) \in r \to (z,y) \notin r)]\\ \vdots \\ \forall z.(x,z) \in r \to (z,x) \notin r\end{array}}{\forall z.(a,z) \in r \to (z,a) \notin r}\ \texttt{wf\_induct}$$

The induction part needs classical reasoning (➜ p.**??**).

We will first give an intuitive proof.

The Induction Part Intuitively

Notation: Write $a < b$ instead of $(a, b) \in r$.

Hypothesis: for every $y < x$ have $\forall \rlap{z}w.\, y < \rlap{z}w \rightarrow \rlap{z}w \not< y$.

To show: It holds that $\forall z.\, x < z \rightarrow z \not< x$. Renaming.

We make a case distinction on z.

Case 1: $z \not< x$. Then trivially $x < z \rightarrow z \not< x$.

Case 2: $z < x$. Then setting $y := z$ and $w := x$ in the hypothesis, we get $z < x \rightarrow x \not< z$, which is equivalent to $x < z \rightarrow z \not< x$.

In both cases $x < z \rightarrow z \not< x$ holds, and thus $\forall z.\, x < z \rightarrow z \not< x$.

The Induction Part Formally

We will now give the induction part at a level of detail that shows the essential reasoning but hides all the swapping (➜ p.**??**) involved in the Isabelle proof.

A variation will be done as exercise.

The Induction Part in More Detail

$$\dfrac{\dfrac{\forall y.(y,x) \in r \to (\forall z.(y,z) \in r \to (z,y) \notin r)}{(w,x) \in r \to (\forall z.(w,z) \in r \to (z,w) \notin r)}\;\forall\text{-}E}{(w,x) \notin r \vee (\forall z.(w,z) \in r \to (z,w) \notin r)}\;\begin{matrix}(\mathsf{c})^{441}\\ \equiv \phi\end{matrix}$$

“(c)” stands for classical reasoning steps.

$$\dfrac{\dfrac{\phi \qquad \dfrac{[(w,x) \notin r]^1}{(x,w) \in r \to (w,x) \notin r}\;impI^2 \qquad \dfrac{\dfrac{[\forall z.(w,z) \in r \to (z,w) \notin r]^1}{\forall z.(z,w) \in r \to (w,z) \notin r}\;(\mathsf{c})^{442}}{(x,w) \in r \to (w,x) \notin r}\;\forall\text{-}E}{(x,w) \in r \to (w,x) \notin r}\;disjE^1}{\forall z.(x,z) \in r \to (z,x) \notin r}\;\forall\text{-}I$$

Theorems (➜ p.511) on Absence of Cycles

wf_not_refl	$\mathit{wf}\, r \Longrightarrow (a, a) \notin r$
wf_trancl	$\mathit{wf}\, r \Longrightarrow \mathit{wf}(r^+)$
wf_acyclic	$\mathit{wf}\, r \Longrightarrow \mathit{acyclic}\, r$

($\mathit{acyclic}\, r \equiv \forall x.(x, x) \notin r^+$ (➜ p.504))

	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* (➜ p.504) really meets our intuition of “no cycles”.

Another Theorem ("Exists Minimal Element")

`wf_minimal` $\mathit{wf}\ r \Longrightarrow \exists x.\forall y.(y,x) \notin r^+$

Proof sketch, writing $\phi \equiv (\exists x.\forall y.(y,x) \notin r^+)$:

$$\dfrac{\dfrac{\texttt{wf}(r)}{\texttt{wf}(r^+)}\bullet \qquad \dfrac{\overline{\phi \vee \neg\phi}\,\bullet \qquad [\phi]^2 \qquad \dfrac{\dfrac{\dfrac{[\neg\phi]^2}{\forall x.\exists y.(y,x) \in r^+}\cdots \qquad \dfrac{[\neg\phi]^2 \quad \left[\begin{array}{c}\forall w.(w,v)\\ \in r^+ \to \phi\end{array}\right]^1}{\neg\exists w.(w,v) \in r^+}\bullet^{443}}{\mathit{False}}\cdots}{\phi}\ \mathit{FalseE}\ (\text{➜ p.380})}{\phi}\ \mathit{disjE}\ (\text{➜ p.391})^2}{\phi}\ \texttt{wf\_minimalwf\_induct}\ (\text{➜ p.512})^1$$

[443] In the proof of $\exists x.\forall y.(y,x) \notin r^+$ we had the sub-proof

$$\dfrac{\neg\phi \quad \forall w.(w,v) \in r^+ \to \phi}{\neg\exists w.(w,v) \in r^+}$$

This sub-proof does not actually depend on ϕ, it would hold no matter what ϕ is (unlike the entire proof (➜ p.521))

In detail, the sub-proof looks as follows:

$$\dfrac{\dfrac{\neg\phi \qquad \dfrac{[\exists w.(w,v) \in r^+]^3 \qquad \dfrac{[(w,v) \in r^+]^4 \qquad \dfrac{\forall w.(w,v) \in r^+ \to \phi}{(w,v) \in r^+ \to \phi}\ \mathit{spec}}{\phi}\ \mathit{mp}}{\phi}\ \mathit{existsE}\ (\text{➜ p.386})^4}{\mathit{False}}\ \mathit{notE}\ (\text{➜ p.383})}{\neg\exists w.(w,v) \in r^+}\ \mathit{notI}\ (\text{➜ p.382})^3$$

Uses *mp* (➜ p.350), *spec* (➜ p.379)

This is what we must construct.

Note “special case”: w and v do not occur in ϕ!

This is `wf_trancl`.

We now try a proof by case distinction on ϕ.

Classical (➜ p.320) reasoning.

Using some elementary equivalences[444].

This subproof works for any ϕ. Think semantically or check (5 rule applications)!

It is routine to derive *False*.

This completes the proof by case distinction …

… and the proof by induction.

See (➜ p.518) and (➜ p.391).

Remarks on the Proof

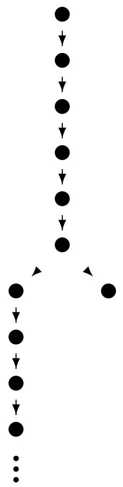
We used an instance of `wf_induct` (➜ p.512), where we instantiated x by v, y by w, and P by $\lambda w.(\exists x.\forall y.(y,x) \notin r^+)$. I.e., ϕ does not contain the “induction variables” w and v.

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 $(\forall w.(w,v) \in r^+ \to \phi) \not\equiv (\forall w.(w,v) \in r^+) \to \phi)$ and side conditions (➜ p.80) are very subtle in this proof. Underlines the importance of machine-checked proofs.

Remarks on `wf_minimal`

Ergo: Definition of `wf` (➜ p.504) fulfills the condition corresponding to our first attempt (➜ p.507) 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 Theorem (➜ p.511) on Subsets

`wf_subset` $[\![wf\ r; p \subseteq r]\!] \Longrightarrow wf\ p$

Proof sketch: `wf_subset`: simplification tactic using `wf_eq_minimal` (➜ p.509).

A Theorem on Subrelations

`wf_subrel`

$$wf\, r \Longrightarrow \forall p.p \subseteq r \to \exists x.\forall y.(y, x) \notin p^+$$

Proof sketch:

Combine `wf_minimal` (➜ p.519) and `wf_subset` (➜ p.523).

This implies $wf\, r \Longrightarrow \forall p.p \subseteq r \to \exists x.\forall y.(x, y) \notin p$ (➜ p.507).

Ergo: Definition of `wf` (➜ p.504) fulfills the condition corresponding to our second attempt (➜ p.507) of characterizing well-foundedness using minimal elements.

However, this formalization still (➜ p.522) 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 (➜ p.461) $f = e$, e.g. (➜ p.461)

$$fac = (\lambda n.\ \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * fac(n-1))$$

Define $F = \lambda f.e$, e.g. (α-conversion (➜ p.158) of what you have seen (➜ p.462))

$$Fac = (\lambda fac.\, \lambda n.\ \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * fac(n-1))$$

We say: F is the functional defining f.

Recall (➜ p.461) that $Y\,F$ would solve $f = e$, but we don't have (➜ p.466) 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.

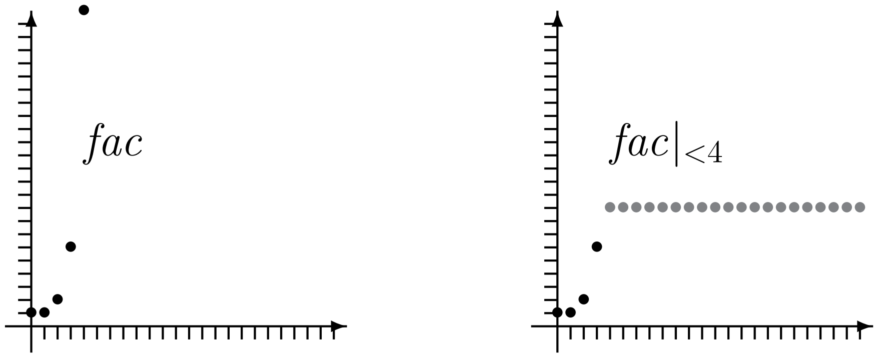
If F is coherent, then we would expect that for any a,

$$f\,a = (F\,f)\,a = (F\,f|_{<a})\,a. \tag{5}$$

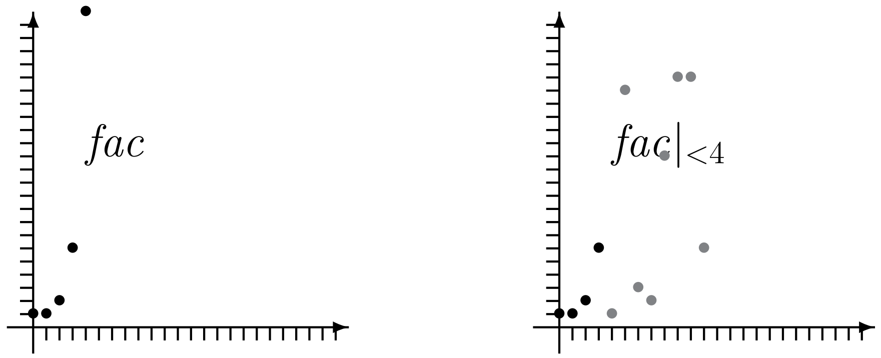
It's not that we are ultimately interested in constructing such a "bad" f, but our formalization of well-founded recursion defines coherence by the fact that one could use such a "bad" f, i.e., via (5).

“Bad” f’s: Example

Consider *fac* (➜ p.461). On the right-hand side, we show one possibility[445] for $fac|_{<4}$):



[445] For the construction we have in mind, it would be fine if $f|_{<a}$ was any 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 (➜ p.344) for HOL (with the extensions we consider here). The way that arbitrary elements are formalized in `HOL.thy` (➜ p.355), 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.504))

```
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.528) is defined in `HOL.thy` (➜ p.355).

The function `cut` f r x is unspecified for arguments y where $(y, x) \notin r$, but for each such argument, (`cut` f r x) y must be the same (in any particular model (➜ p.344)).

---

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 (➜ p.511) Involving `cut`

`cuts_eq`	$(cut\, f\, r\, x = cut\, g\, r\, x) =$ $(\forall y.(y, x) \in r \rightarrow f\, y = g\, y)$
`cut_apply`	$(x, a) \in r \Longrightarrow cut\, f\, r\, a\, x = f\, x$

Or, using the more intuitive notation:

`cuts_eq`	$(f\vert_{<x} = g\vert_{<x}) = (\forall y.y < x \rightarrow f\, y = g\, y)$
`cut_apply`	$x < a \Longrightarrow f\vert_{<a}\, x = f\, x$

wfrec_rel (in `Wellfounded_Recursion.thy` (➜ p.504))

Auxiliary construction: “approximate” f by a relation $\mathit{wfrec\_rel}\ R\ F$.

```
  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

$$\forall z.(z,x) \in R \to (z, g\, z) \in \mathit{wfrec\_rel}\, R\, F \Longrightarrow$$
$$(x, F\, g\, x) \in \mathit{wfrec\_rel}\, R\, F$$

- For R and F arbitrary, $\mathit{wfrec\_rel}\, R\, F$ is defined but we wouldn't want to know what it is.
- But if R is well-founded and F is coherent, $\mathit{wfrec\_rel}\, R\, F$ defines a recursive "function"[446].

Show that $(4, 24) \in (\mathit{wfrec\_rel}\,`<'\, \mathit{Fac})$!
Now let us really turn $\mathit{wfrec\_rel}\, R\, F$ into a function ...

[446] When we say that a binary relation $r : \tau \times \sigma$ is in fact a function, we mean that for $t : \tau$, there is exactly one $s : \sigma$ such that $(t, s) \in r$.

wfrec (in `Wellfounded_Recursion.thy` (➜ p.504))

```
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` $x.P\,x$[447] picks the unique a such that $P\,a$ holds, if it exists. We don't care what it does otherwise (see `HOL.thy` (➜ p.355)).

[447] The operator `THE` is similar to the Hilbert operator (➜ p.332), 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

$$\begin{array}{l} \mathit{wfrec}\,R\,F \equiv \\ \lambda x.\mathtt{THE}\,y.(x,y) \in \mathit{wfrec\_rel}\,R\,(\lambda f\,x.F\,(\mathit{cut}\ f\ R\,x)\,x) \end{array}$$

We don't care what this means for arbitrary R and F.

But if R is well-founded and F is coherent, then $F\,(\mathit{cut}\ f\ R\,x)\,x = F\ f\ x$ (by (5)), and so $\lambda f\,x.F\,(\mathit{cut}\ f\ R\,x)\,x = F$, and so $\lambda x.\mathtt{THE}\,y.(x,y) \in \mathit{wfrec\_rel}\,R\,(\lambda f\,x.F\,(\mathit{cut}\ f\ R\,x)\,x)$ is the function defined by $\mathit{wfrec\_rel}\,R\,F$ in the obvious way.

`wfrec` $R\,F$ is the recursive function defined by functional F.

The “Fixpoint” Theorem (➜ p.511)

`wfrec` $wf\ r \Longrightarrow wfrec\ r\ H\ a = H(cut(wfrec\ r\ H)\ r\ a)\ a$

Note that `wfrec` is used here both as a name of a constant (defined above (➜ p.533)) 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.461).

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 (➜ p.554)).

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 argument[448].

```
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.

[448]

```
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 (➜ p.541), but conceptually, the above is a definition of the `add` function.

Note that `add` is a function of type *nat* → *nat* → *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 − (➜ p.537).

The functions are recursive in one argument (just like `add` (➜ p.537)).

By the definitions (of *wfrec* (➜ p.533) most importantly), this expression is a function of type $nat \to nat$, 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 (➜ p.511) of the Example

`wf_pred_nat`	$\mathit{wf}\ \mathit{pred\_nat}$
`mod_if`	$m\,\mathit{mod}\,n =$ $(\mathit{if}\ m < n\ \mathit{then}\,m\ \mathit{else}\,(m-n)\,\mathit{mod}\,n)$
`div_if`	$0 < n \Longrightarrow m\ \mathit{div}\,n =$ $(\mathit{if}\ m < n\ \mathit{then}\,0\ \mathit{else}\ \mathit{Suc}((m-n)\,\mathit{div}\,n))$

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 (➜ p.438) that can be used for rewriting just like in a functional programming language.

Isabelle Package for Primitive Recursion

For primitive recursion[449], finding a well-founded ordering is simple enough for automation[450]!

Examples (use `nat` (➜ p.554) and `case` (➜ p.560)-syntax):

...

[449] A function is primitive recursive if the recursion is based on the immediate predecessor w.r.t. the well-founded order used (e.g., the predecessor on the natural numbers, as opposed to any arbitrary smaller numbers).

This is not the same concept as used in the context of computation theory, where primitive recursive is in contrast to μ-recursive [LP81].

[450] The `primrec` syntax provides a convenient front-end for defining primitive recursive functions.

Isabelle will guess a well-founded ordering to use. E.g. for functions on the natural numbers, it will use the usual $<$ ordering.

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 (➜ p.466), but we have, by conservative extension (➜ p.404):

- Least fixpoints for defining sets (➜ p.467);
- well-founded orders for defining functions (➜ p.497).

Both concepts come with induction schemes (lfp induction (➜ p.484) and definition of well-foundedness (➜ p.504)) for proving properties of the defined objects.

Summary: Proof Support

The methodological overhead can be faced by powerful mechanical support in Isabelle, since many proof-tasks are routine.

24 Arithmetic

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library (➜ p.427).

- Orders (➜ p.431)
- Sets (➜ p.440)
- Functions (➜ p.451)
- (Least) fixpoints and induction (➜ p.467)
- (Well-founded) recursion (➜ p.497)
- Arithmetic
- Datatypes (➜ p.568)

Current Stage of our Course

- On the basis of conservative embeddings, set theory (➜ p.440) can be built safely.

- Inductive sets (➜ p.491) can be defined using least fixpoints (➜ p.467) and suitably supported by Isabelle (➜ p.491).

- Well-founded orderings (➜ p.497) can be defined without referring to infinity (➜ p.507). Recursive functions can be based on these. Needs inductive sets (➜ p.531) though. Support by Isabelle (➜ p.541) provided.

Next important topic: arithmetic.

Which Approach to Take?

- Purely definitional (➜ p.398)?

  Not possible with eight basic rules (➜ p.350) (cannot enforce infinity[451] of HOL model)!

- Heavily axiomatic? I.e., we state natural numbers by Peano axioms[452] and claim analogous axioms for any other number type?

  Danger of inconsistency!

- Minimally axiomatic? We construct an infinite set, and define numbers etc. as inductive subset (➜ p.467)?

  Yes. Finally use infinity (➜ p.334) axiom.

24.1 What is Infinity? Cantor's Hotel

[451] Our intuition/knowledge about arithmetics clearly requires that there are infinite sets, e.g., the set of infinite numbers. Technically, the HOL model of the set of natural numbers must be an infinite set, otherwise we would not be willing to say that we have "modeled" arithmetic.

[452] The Peano axioms are

– $0 \in nat$

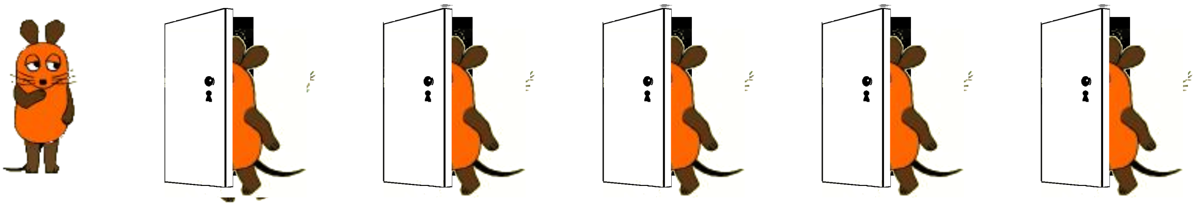
– $\forall x.x \in nat \to Suc(x) \in nat$

– $\forall x.Suc(x) \neq 0$

– $\forall x\ y.Suc(x) = Suc(y) \to x = y$

– $\forall P.(P(0) \wedge \forall n.(P(n) \to P(Suc(n)))) \to \forall n.P(n)$.

However, there are various ways of phrasing the Peano axioms.



Cantor's hotel has infinitely many rooms. New guest arrives.

The doors open, and all guests come out of their rooms. They move one room forward[453], the new guest walks towards the first room, they turn around, enter their new rooms. The doors close, all guests are accomodated.

[453] This means, there must be a successor function on rooms. To each room, it assigns the "next" room.

Axiom of Infinity

The axiomatic core[454] of numbers[455]:

$$\frac{}{\exists f :: (ind \to ind).\ injective\ f \wedge \neg surjective\ f}\ infty\ (\text{➜ p.352})$$

where

$$\begin{aligned} injective^{456}\ f &= \forall xy.\, f\,x = f\,y \to x = y \\ surjective\ f &= \forall y.\exists x.\, y = f\,x \end{aligned}$$

Forces *ind* to be "infinite type" (➜ p.334) (called "*I*" in [Chu40]).

We will see soon (➜ p.554) how this is done in Isabelle.

[454] Note that theoretically, it is not needed to add the infinity axiom (or some equivalent formulation (➜ p.554)) 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.

[455] The natural numbers can be built as an algebraic datatype by having a constant 0 and a term constructor *Suc* (for successor).

[456] These constants (actually called *inj* and *sur* (➜ p.552)) are defined in `Fun.thy` (➜ p.453).

24.2 Type-Closed Conservative Extensions

Why must conservative extensions be type-closed [GM93, page 221]?

Consider $H \equiv \exists f :: \alpha \Rightarrow \alpha.\ \mathit{injective}\ f \wedge \neg \mathit{surjective}\ f$

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)

$(H \equiv \exists f :: \alpha \Rightarrow \alpha.\ \mathit{injective}\ f \wedge \neg \mathit{surjective}\ f)$

$$\begin{array}{rl} & H = H \qquad \text{holds by } \mathit{refl} \text{ (➜ p.350)} \\ \Rightarrow & \exists f :: \mathit{bool} \Rightarrow \mathit{bool}.\mathit{inj}^{457}\ f \wedge \neg \mathit{sur}\ f = \\ & \quad \exists f :: \mathit{ind} \Rightarrow \mathit{ind}.\mathit{inj}\ f \wedge \neg \mathit{sur}\ f \\ \Rightarrow & \mathit{False} = \mathit{True} \\ \Rightarrow & \mathit{False} \end{array}$$

(unfolding H using two different type instantiations, and then using axiom of infinity (➜ p.352) and the fact that there are only finitely many functions on *bool*).

[457] We 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.407).

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 (➜ p.352) is formulated by defining a constant *Suc_Rep* having the two required properties.

inj (➜ p.552) is defined in `Fun.thy` (➜ p.453).

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 (➜ p.410) to the natural numbers. How?

By induction using the `inductive` syntax (➜ p.491):

```
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` (➜ p.422) syntax:

```
typedef (open Nat)
  nat = "Nat" by (rule exI, rule Nat.Zero_RepI)
```

After these two steps[458] we have the type *nat*.

[458] Note the two ingredients for defining the type `nat`:

- An inductively defined set (➜ p.484) `Nat`, i.e., a set defined as fixpoint of a monotone function. In Isabelle (`Nat.thy` (➜ p.558)), the `inductive` syntax (➜ p.491) is used for this purpose. This automatically generates an induction rule (➜ p.495) for the set.
- A type definition (➜ p.410) based on this set, defined using the `typedef` syntax (➜ p.422).

  Recall (➜ p.422) that this process automatically generates the two constants `Abs_Nat` (➜ p.421) and `Rep_Nat` (➜ p.421).

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, define[459]:

```
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.558).

[459] Based on the generic constants `Abs_Nat` (➜ p.421) and `Rep_Nat` (➜ p.421), we define all the constants that we need to work conveniently with `nat`, most importantly, 0 and `Suc`.

Some Theorems (➜ p.438) in `Nat.thy`[460]

`nat_induct` $[\![P\,0; \bigwedge n. P\,n \Longrightarrow P\,(Suc\,n)]\!] \Longrightarrow P\,n$

`diff_induct` $$[\![\bigwedge x. P\,x\,0; \bigwedge y. P\,0\,(Suc\,y);\\ \bigwedge xy. P\,x\,y \Longrightarrow P\,(Suc\,x)\,(Suc\,y)]\!]\\ \Longrightarrow P\,m\,n$$

We can now exploit that *nat* is defined based on a set (➜ p.488) defined using least fixpoints (➜ p.468). In particular, `nat_induct` follows (➜ p.556) from the `induct` (➜ p.484) theorem (➜ p.438) of `Lfp` (➜ p.470).

[460] 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/`

`Nat` (➜ p.558) and Well-Founded Orders

Examples of theorems (➜ p.438) involving well-founded orders (➜ p.497):

`wf_pred_nat`	$wf\ pred\_nat$
`less_linear`	$m < n \lor m = n \lor n < m$
`Suc_less_SucD`	$Suc\, m < Suc\, n \Longrightarrow m < n$

Using Primitive Recursion

`Nat.thy` (➜ p.558) defines rich theory on *nat*. Uses `primrec` (➜ p.541) syntax for defining recursive functions (➜ p.497), and `case`[461] 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)"
```

[461] The `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):

$$\texttt{case}\ z\ f\ n = \begin{cases} z & \text{if } n = 0 \\ f\ k & \text{if } n = \mathit{Suc}\ k \end{cases}$$

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.364)) of the original (“abstract”) syntax. In the original syntax it would read `case` $0\ (\lambda x.x)\ (n - m)$.

Some Theorems (➜ p.438) in `Nat` (➜ p.558)

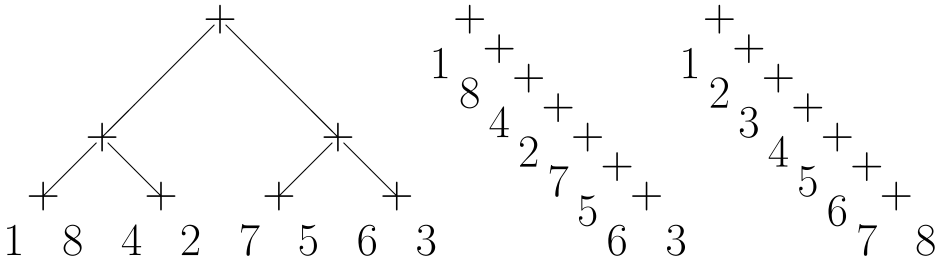
`add_0_right`	$m + 0 = m$
`add_ac`	$m + n + k = m + (n + k)$
	$m + n = n + m$
	$x + (y + z) = y + (x + z)$
`mult_ac`	$m * n * k = m * (n * k)$
	$m * n = n * m$
	$x * (y * z) = y * (x * z)$

Note third part[462] of `add_ac`, `mult_ac`, respectively.

Technically, `add_ac` and `mult_ac` are lists of `thm` (➜ p.293)'s.

[462] The theorems $x + (y + z) = y + (x + z)$ and $x * (y * z) = y * (x * z)$ are called left-commutation laws and are crucial for (ordered (➜ p.308)) rewriting (➜ p.299).

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 (➜ p.306), for an appropriately chosen term ordering (➜ p.308).



Proof of `add_0_right`

$$\dfrac{\dfrac{\text{add\_0}}{0+0=0} \qquad \dfrac{\dfrac{\dfrac{\text{add\_Suc}}{Suc\,n+0=Suc(n+0)}}{Suc(n+0)=Suc\,n+0}\,\mathit{sym} \qquad \dfrac{[n+0=n]^1}{Suc(n+0)=Suc\,n}\,\mathit{arg\_cong}}{Suc\,n+0=Suc\,n}\,\mathit{subst}}{m+0=m}\,\text{add\_0\_rightnat\_induct}^1$$

Note that $Suc\,n+0 = Suc(n+0)$ is an instance of $Suc\,m+n = Suc(m+n)$.

24.4 Integers

The integers are implemented[463] as equivalence classes[464] over $nat \times nat$.

```
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)
```

[463] The 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/
```

[464] Recall the general concept of an equivalence relation (➜ p.104). 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.438) in `IntArith`

`zminus_zadd_distrib`	$-(z+w) = -z + -w$
`zminus_zminus`	$-(-z) = z$
`zadd_ac`	$z1 + z2 + z3 = z1 + (z2 + z3)$
	$z + w = w + z$
	$x + (y + z) = y + (x + z)$
`zmult_ac`	$z1 * z2 * z3 = z1 * (z2 * z3)$
	$z * w = w * z$
	$z1 * (z2 * z3) = z2 * (z1 * z3)$

Compare to *nat* theorems (➜ p.561).

---

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.410) the set of all pairs of naturals (which corresponds to the `UNIV` (➜ p.445) constant on the type *nat* × *nat* (➜ p.420)) 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`[465], for fast computation)
- Rational Numbers (`Real/PRat.thy`[466])

[465] 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/`

[466] 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/`

- Reals[467] (`Real/PReal.thy`[468]: based on Dedekind-cuts of rationals [Fle00])
- Machine numbers (floats); see work for Intel's PentiumIV; built in HOL-light [Har98, Har00]
- ...[469]

[467] The reals have been axiomatized by Dedekind by stating that a set R is partitioned into two sets A and B such that $R = A \cup B$ and for all $a \in A$ and $b \in B$, we have $a < b$. Now there is a number s such that $a \leq s \leq b$ for all $a \in A$ and $b \in B$. The irrational numbers are characterised by the fact that there exists exactly one such s. This axiomatization has been used as a basis for formalizing real numbers in Isabelle/HOL.

[468] 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/`

[469] In non-standard analysis, one works with sequences that are not necessarily converging. This is a relatively new field in mathematics and Isabelle/HOL has been successfully applied in it [FP98].

This is the Isabelle file `Real/RealDef.thy` which should

24.6 Conclusion on Arithmetic

Using conservative extensions (➜ p.398) in HOL, we can build

- the naturals (➜ p.554) (as type definition (➜ p.410) based on *ind*), and
- higher number theories (➜ p.565) (via equivalence construction).

Potential for

- analysis of processor arithmetic units, and
- function analysis in HOL (combination with computer algebra systems such as Mathematica).

The methodological overhead can be tackled by powerful mechanical support, since many proof-tasks are routine.

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/`

We just mention this here to say that Isabelle/HOL is used for "cutting-edge" mathematics and not just toy examples.

25 Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library (➜ p.427).

- Orders (➜ p.431)
- Sets (➜ p.440)
- Functions (➜ p.451)
- (Least) fixpoints and induction (➜ p.467)
- (Well-founded) recursion (➜ p.497)
- Arithmetic (➜ p.545)
- Datatypes

What Are Datatypes?

We have seen types, but what are data[470]types?

- Order 0 (➜ p.219) (no → in type).
- Terms defined by finite set of term constructors (➜ p.300).
- Typically inductive definition.
- Term constructed by syntactic rule is unique.

[470] We have seen types, but what are datatypes?

First of all, a datatype must be of order 0 (➜ p.219), so it must be a non-functional type. Note that if we do not have polymorphism, this means that a datatype must be in $\mathcal{B}$ (➜ p.162). But if we have polymorphism, it just means that the type must not contain →. 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 (➜ p.300) that have τ as result type. At least one term constructor should just have type τ. E.g., $\mathit{Nil} : \alpha\ \mathit{list}$ and $\mathit{Cons} : \alpha \rightarrow (\alpha\ \mathit{list}) \rightarrow \alpha\ \mathit{list}$ 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.

Counterexample[471]: α *set* (➜ p.440).

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 (➜ p.550), 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 $Cons(1, Cons(1, Nil))$.

[471] To understand better the distinction of a datatype from another type, consider the following counterexample:

Datatypes: Motivation

We will now construct "datatypes (➜ p.570)" (as in ML [Pau96]). This construction is based on so-called S-expressions [Pau97].

Caveat: We will only sketch the construction and we will simplify, meaning that the technical details will not be

---

α *set* (➜ p.440). Sets are not a datatype:

1. While the type α *set* does not contain an $\rightarrow$, it is isomorphic (➜ p.415) to $\alpha \rightarrow$ *bool* which does contain an $\rightarrow$.

2. The most basic way of defining "what a set is" is: if f is of type $\tau \rightarrow$ *bool*, then $\mathit{Abs}_{set}\ f$ (➜ p.416) (alternatively: *Collect* f (➜ p.417)) 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 $\mathit{Insert}(h, S)$ is a set. But then $\mathit{Insert}(1, \{\})$ would be different from $\mathit{Insert}(1, \mathit{Insert}(1, \{\}))$, 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-

strictly correct! See `Datatype_Universe.thy`[472] and [Wen99].

tradiction to the inductive definition (➜ p.488) 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.

[472] 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/`

S-Expressions as Basis

In the end we want to have datatypes such as lists (➜ p.300) 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 do not have the concept of datatype yet, we must first represent S-expressions using constructs we already have.

25.1 S-Expressions

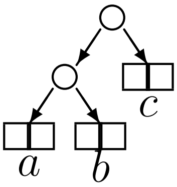
LISP-like S-expressions[473] are a kind of of binary trees. We call the type α *dtree*. This uses $\alpha + nat$ (➜ p.424).

[473] The datastructure we have in mind here consists of binary trees where the inner nodes are not labeled, and the leaves are labeled

- either with a term of arbitrary type, in which case the leaf would be an actual "piece of content" in the datastructure,
- or with a natural number, in which case the leaf serves special purposes for organizing our datastructure, as we will see later.

I.e., such binary trees have a type parametrized by a type variable α, the type of the former kind of leaves. Let us call the type of such trees α *dtree*.

As always with parametric polymorphism (➜ p.179), 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 (➜ p.179) two lists.



This is encoded as a set of "leaves"[474] (defined by their path from the root and a value), e.g.:

$$\{(\langle 0,0\rangle, a), (\langle 0,1\rangle, b), (\langle 1\rangle, c)\}$$

The type definition (➜ p.410) of α *dtree* uses such an encoding.

Of course, α could, by coincidence, be instantiated to type *nat*.

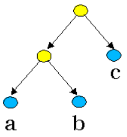
Think of a label of the first kind as content label and a label of the second kind as administration label.

Technically, if something is either of this type or of that type, we are talking about a sum type (➜ p.424). So a leaf label has type $\alpha + nat$ (written (α, nat) *sum* (➜ p.424) before), and it has the form either *Inl* (➜ p.424)(a) for some $a :: \alpha$, or *Inr* (➜ p.424)(n) for some $n :: nat$.

[474] The set

$$\{(\langle 0,0\rangle, a), (\langle 0,1\rangle, b), (\langle 1\rangle, c)\}$$

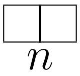
represents the tree



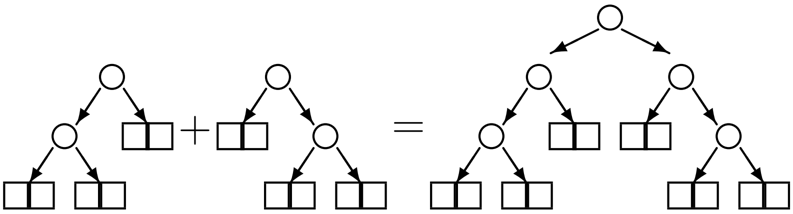
The path $\langle 0,0\rangle$ means: from the root take left subtree, then again left subtree. The path $\langle 1\rangle$ means: take right subtree.

Building Trees

- $Atom(n)$[475]



- $Scons\ X\ Y$[476]



How can a path $\langle p_0, \ldots, p_n \rangle$ be represented? One idea is to use the function $f :: nat \Rightarrow nat$ defined by

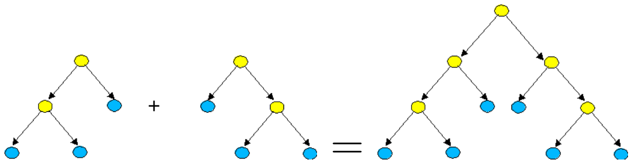
$$f\ i = \begin{cases} p_i & \text{if } i \leq n \\ 2 & \text{otherwise} \end{cases}$$

as representation of $\langle p_0, \ldots, p_n \rangle$.

[475] *Atom* takes a leaf label (➜ p.575) and turns it into a (simplest possible) S-expression (➜ p.574) (tree).

So it has type $\alpha + nat \Rightarrow \alpha\ dtree$.

[476] *Scons* takes two S-expressions (➜ p.574) and creates a new S-expression as illustrated below:

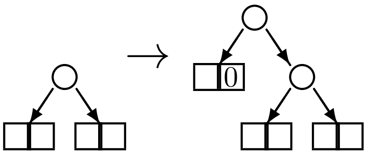


So it has type $[\alpha\ dtree, \alpha\ dtree]$ (➜ p.185) $\Rightarrow \alpha\ dtree$.

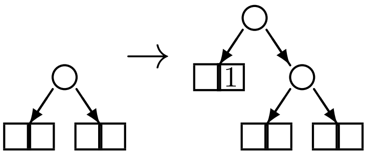
Tagging Trees

We want to tag an S-expression by either 0 or 1. This can be done by "*Scons*" (➜ p.576)-ing it with an S-expression consisting of an administration label (➜ p.575). By convention, the tag is to the left.

- `In0_def` $\mathit{In0}(X) \equiv \mathit{Scons}\,\mathit{Atom}(\mathit{Inr}$ (➜ p.424)$(0))X$



- `In1_def` $\mathit{In1}(X) \equiv \mathit{Scons}\,\mathit{Atom}(\mathit{Inr}$ (➜ p.424)$(1))X$



Products and Sums on Sets of S-Expressions

Product of two sets A and B of S-expressions: All *Scons* (➜ p.576)-trees where left subtree from A, right subtree from B.

$$\texttt{uprod\_def} \quad \mathit{uprod}\,A\,B \equiv \bigcup_{x\in A}\bigcup_{y\in B}\{(\mathit{Scons}\,x\,y)\}$$

Sum of two sets A and B of S-expressions: union of A and B after S-expressions in A have been tagged 0 (➜ p.577) and S-expressions in B have been tagged 1 (➜ p.577), so that one can tell where they come from.

$$\texttt{usum\_def} \quad \mathit{usum}\,A\,B \equiv \mathit{In0}\,\text{‘}^{477}A \cup \mathit{In1}\,\text{‘}\,B$$

[477] Recall that ‘ denotes the image (➜ p.447) of a function applied to a set.

Some Properties of Trees and Tree Sets

- *Atom*, *In0*, *In1*, *Scons* are[478] injective (➜ p.352).
- *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 $[(\alpha\ \mathit{dtree})\ \mathit{set}, (\alpha\ \mathit{dtree})\ \mathit{set}]$ (➜ p.185) $\Rightarrow (\alpha\ \mathit{dtree})\ \mathit{set}$.
- *uprod* and *usum* are monotone (➜ p.469).
- Tree sets represent a universe that is closed under products and sums[479] combined with arbitrary applications of *lfp* (➜ p.468).

Reminder: we simplified!

[478] This means that any of *Atom*, *In0*, *In1*, *Scons* applied to different S-expressions will return different S-expressions.

Moreover, a term with root *Scons* is definitely different from a term with root *Atom*, and a term with root *In0* is definitely different from a term with root *In1*.

Why is this important? It is an inherent characteristic of a datatype (➜ p.570). A datatype consists of terms constructed using term constructors (➜ p.300) and is uniquely defined by what it is syntactically (one also says that terms are generated freely using the constructors). For example, (➜ p.550) injectivity of *Suc* and pairwise-distinctness of 0 and *Suc* mean for any two numbers m and n, the terms $\underbrace{Suc(\ldots Suc}_{m \text{ times}}(0)\ldots)$ and $\underbrace{Suc(\ldots Suc}_{n \text{ times}}(0)\ldots)$ are different.

[479] Given a set T of trees (S-expressions), the closure of T under *Atom*, *In0*, *In1*, *Scons*, *usum*, *uprod* is the smallest set T' such that $T \subseteq T'$ and given any tree (or two trees,

25.2 Lists in Isabelle

Similar to the construction of *nat* (➜ p.556), we first construct a set of S-expressions having the "structure of lists". We start by defining "provisional" (➜ p.554) list constructors:

```
constdefs
  NIL :: 'a dtree
  "NIL == In0(Atom(Inr(0)))"
  CONS ::  ['a dtree, 'a dtree] => 'a dtree
  "CONS M N == In1(Scons M N)"
```

What type do you expect[480] *Cons* to have, and how does *CONS* compare? Must wrap list elements by *Atom* ∘ *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 (➜ p.467), 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 (➜ p.469), and note as well that slight complications arise from the fact that *usum* and *uprod* have type $[(\alpha\ dtree)\ set, (\alpha\ dtree)\ set]$ (➜ p.185) $\Rightarrow (\alpha\ dtree)\ set$ rather than $(\alpha\ dtree)\ set \Rightarrow (\alpha\ dtree)\ set$.

[480] *Cons* should have the polymorphic type $[\alpha, \alpha\ list] \Rightarrow \alpha\ list$. 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 $\tau\ list$.

In contrast, *CONS* is of type $[(\alpha\ dtree), (\alpha\ dtree)] \Rightarrow \alpha\ dtree$.

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:

Nil[481]	$[]$
$\quad \mathit{In0}(\mathit{Atom}(\mathit{Inr}\ \mathit{0}))$	
$\mathit{Cons}(7, \mathit{Nil})$	$[7]$
$\quad \mathit{CONS}\ (\mathit{Atom}(\mathit{Inl}\, 7))\ \mathit{In0}(\mathit{Atom}(\mathit{Inr}\ \mathit{0}))$	
$\mathit{Cons}(5, \mathit{Cons}(7, \mathit{Nil}))$	$[5, 7]$
$\quad \mathit{CONS}\ (\mathit{Atom}(\mathit{Inl}\, 5))$ $\qquad (\mathit{CONS}\ (\mathit{Atom}(\mathit{Inl}\, 7))\ \mathit{In0}(\mathit{Atom}(\mathit{Inr}\ \mathit{0})))$	

Now let's construct the S-expressions having this form.

expression) and a "list element", we must first wrap the list element by $\mathit{Atom} \circ \mathit{Inl}$, so that it becomes an S-expression.

[481] Nil, $\mathit{Cons}(7, \mathit{Nil})$, $\mathit{Cons}(5, \mathit{Cons}(7, \mathit{Nil}))$ 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 $A :: (\alpha\ dtree)\ set$ be the set of all "wrapped" elements, e.g. for $\alpha = nat$, the set $\{(Atom\ Inl\ 0), (Atom\ Inl\ 1), \ldots\}$. 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`[482] for how it's really done!

[482] 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/`

Defining the "Real" List Type

We now apply the type definition mechanism (➜ p.410) using the **typedef** (➜ p.422) syntax. How do we define A formally?

```
typedef (List)
  'a list =
   "list(range (Atom o Inl)) :: 'a dtree set"
  by ...
```

Choosing A as $range\,(Atom \circ Inl)$ 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 (➜ p.179) 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 (➜ p.422), Isabelle provides the `datatype` syntax to support the construction (➜ p.410) 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.

Question: Why didn't we use this package to define *nat*[483]?

[483] The `datatype` syntax is very convenient since the complex construction we have seen today is transparent to the normal user.

In particular, proofs of the induction theorem are automated. This is in contrast to the construction of *nat* where this theorem was not generated automatically (➜ p.556).

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* (➜ p.575) 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.427):

- Orders (➜ p.431)
- Sets (➜ p.440)
- Functions (➜ p.451)
- (Least) fixpoints and induction (➜ p.467)
- (Well-founded) recursion (➜ p.497)
- Arithmetic (➜ p.545)
- Datatypes (➜ p.568)

Summary (Cont.)

We conclude: HOL is a logical framework for 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;
- a package for induction;
- a package for recursion;
- a package for datatypes.

Outline

We could now look at how various formalisms (specification and programming languages) can be embedded in HOL:

- the specification language Z
- Imperative languages (➜ p.590)
- Denotational semantics and functional languages
- Object-oriented languages (Java-Light . . .)

In previous years, a varying choice of these topics has been presented. Now, we do imperative languages in the lecture and something resembling functional languages in the labs.

27 IMP

27.1 IMP: Introduction

IMP is a small imperative programming language. We study how its syntax and semantics are represented in HOL.

Semantics come in different flavors[484]:

- operational,
- denotational,
- axiomatic (Hoare-logic).

For operational semantics (➜ p.597), the idea is that our machine is always in some state, essentially consisting of the values of the program variables. The instructions of a program transform a state into a new state. Operational semantics are useful for compiler construction.

For denotational semantics (➜ p.607), the idea is that the meaning of a particular program is a relation between "input" states and "output" states.

Axiomatic semantics (➜ p.612) consist of a calculus for constructing proof obligations. This allows us to state the desired behavior of a program as a logic formula and check it.

[484] One distinguishes

- operational,
- denotational,
- axiomatic

semantics.

Imperative Languages in the Isabelle/HOL Library

There are several embeddings of imperative languages in Isabelle/HOL [Nip02]:

• Hoare:	shallowish, good examples
• IMP:	deepish, good theory
• IMPP:	extends IMP with procedures
• MicroJava:	complex, powerful, state-of-the-art

Explanations of Hoare[485] and shallow/deep[486].

We choose IMP to learn a bit about “good ole imperative languages”.

[485] You should find directories for each mentioned imperative language in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

`http://isabelle.in.tum.de/library/`

[486] The notions shallow and deep refer to the way the imperative language at hand is encoded in Isabelle.

In a deep embedding, the syntax of the analyzed programming language is modeled by (an) explicit datatype(s).

In a shallow embedding, the syntax of the analyzed programming language is implicit in the notation for operators on the semantic domain.

Semantics Provided for IMP

IMP offers:

- operational (➜ p.597) semantics;
  - natural semantics (➜ p.598);
  - transition semantics (➜ p.602);
- denotational semantics (➜ p.607);
- axiomatic semantics (➜ p.612) (Hoare logic);
- equivalence proofs[487];
- weakest preconditions (➜ p.643) and verification condition generator (➜ p.648).

It closely follows the standard textbook [Win96].

[487] Summarizing, we have the following equivalence results:
– natural vs. transition semantics (➜ p.606)
– denotational vs. natural semantics (➜ p.611).

An Imperative Language Embedding

We will now define the syntax and various semantics of IMP, but in fact, we define those as Isabelle theories. We say that we embed IMP in Isabelle/HOL.

You will see that such an embedding is more abstract and less detailed than if we were really going to define IMP for use as a programming language, i.e., if we were going to define a compiler for it.

The Command Language (Syntax)

The (abstract) syntax is defined in `Com.thy`[488].

```
Com = Main +           datatype com =
types                   SKIP
 loc                    | ":==" loc aexp (infixl 60)
 val = nat (*e.g.*)     | Semi com com ("_ ; _" [60, 60] 10)
 state = loc => val     | Cond bexp com com
 aexp = state => val             ("IF _ THEN _ ELSE _" 60)
 bexp = state => bool   | While bexp com ("WHILE _ DO _" 60)
```

The type *loc* stands for locations[489].

The datatype *com* stands for command(sequence)s.

[488] This file defines the command syntax. An Isabelle term of type *com* is an IMP program.

You should find the files in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

`http://isabelle.in.tum.de/library/`

[489] We realize program variables via pointers (locations). The type of pointers is an abstract datatype (➜ p.295).

We take the type of values to be *nat* (➜ p.554), just to have something simple.

A state is a function taking a location to a value, i.e. intuitively, each program variable has a value in a state.

The Command Language (Syntax)

The (abstract) syntax is defined in `Com.thy`[490].

```
Com = Main +         datatype com =
types                 SKIP
 loc                  | ":==" loc aexp (infixl 60)
 val = nat (*e.g.*)   | Semi com com ("_ ; _" [60, 60] 10)
 state = loc => val   | Cond bexp com com
 aexp = state => val           ("IF _ THEN _ ELSE _" 60)
 bexp = state => bool | While bexp com ("WHILE _ DO _" 60)
```

Note the abstractness[491] of *aexp* and *bexp*.

We consider Boolean expressions, but not Boolean vari-

[490] This file defines the command syntax. An Isabelle term of type *com* is an IMP program.

You should find the files in your Isabelle distribution. Or, if you only have an Isabelle executable, you can find the sources here:

`http://isabelle.in.tum.de/library/`

[491] In a formalization of the syntax of an imperative language, there will usually be some grammar saying that (➜ p.16) 1, $x+1$ (provided that x is an arithmetic variable) etc. are arithmetic expressions and that *True*, $x == 1$ etc. are Boolean expressions. Such expressions can only be evaluated if the state, i.e. the value of the program variables, is given.

Now, our notion of expressions (as realized by the types *aexp* and *bexp*) is much more abstract than that. An expression is a function taking a state to a value or Boolean, as applicable.

ables[492].

The fact that IMP has no explicit expression language allows for simple and abstract proofs.

[492] We have Boolean expressions (➜ p.594), but we do not consider Boolean variables here. If we wanted to introduce Boolean variables, we would need to generalize our language *Com*: we would need another location type, say *boolloc*, and another “state component”, say *boolstate*, of type *boolloc* $\Rightarrow$ *bool*.

27.2 Operational Semantics: Two Kinds

Natural semantics [Plo81] (idea: a program relates states[493]):

$$\begin{tikzcd} & & \boxed{state'''} \\ \boxed{state} \arrow[r, "a :== b"] & \boxed{state'} \arrow[ur, "\texttt{WHILE} \ldots"] \arrow[dr, "\texttt{SKIP}"'] & \\ & & \boxed{state''} \end{tikzcd}$$

$evalc :: (com * state * state)\ set$

Transition semantics (idea: sequence of “configurations”[494]):

$$\begin{tikzcd} & & \boxed{X'', state''} \\ \boxed{a :== b; X, state} \arrow[r] & \boxed{X, state'} \arrow[ur] \arrow[dr] & \\ & & \boxed{X''', state'''} \end{tikzcd}$$

$evalc1 :: ((com * state) * (com * state))\ set$

[493] The idea of the natural semantics is that a program relates two states, the “input state” and the “output state”.

This may remind you of denotational (➜ p.590) semantics, and in fact, the natural semantics is a kind of hybrid between operational and denotational semantics.

The fact that the natural semantics just relates an “input state” and an “output state” means, so to say, that it does not record what happens in between, i.e. at the single steps of a computation. In that respect, it resembles denotational semantics.

But the way the meaning of a whole program is defined is still operational in nature. Essentially, it is defined (➜ p.599) in terms of the meaning of the first execution step and the meaning of the rest of the program.

[494]

Unlike the natural semantics, the transition semantics records the single steps of the computation. A configuration is a pair consisting of a program and a state, and one step

27.3 Embedding of the Natural Semantics

The natural semantics encoding in Isabelle is given by an inductive definition. We first declare its type and define a paraphrasing using an arrow symbol for readability:

$$\texttt{consts}\ \mathit{evalc} :: \ \text{"}(\mathit{com} * \mathit{state} * \mathit{state})\ \mathit{set}\text{"}$$
$$\texttt{translations}\ \ \text{"}\langle c, s_0\rangle \xrightarrow{c} s_1\text{"} \equiv \text{"}(c, s_0, s_1) \in \mathit{evalc}\text{"}$$

Note that $\xrightarrow{c}$ (in ASCII: `-c->`) is one fixed (➜ p.605) arrow symbol.

We now start giving the actual inductive definition. It defines the $\xrightarrow{c}$ transitions (implicit: these are the only $\xrightarrow{c}$ transitions) . . .

---

reaches a new program and a new state.

Why "reaching a new program"? This realizes a program counter (➜ p.603). For example, if the first line of the program is an assignment, then the new program is obtained by removing that line from the old program.

Inductive Definition: Skip and Assignment

`inductive` evalc
 `intrs`
  `Skip:` $\langle \texttt{SKIP}, s\rangle \xrightarrow{c} s$
  `Assign:` $\langle x :== a, s\rangle \xrightarrow{c} s[x ::= (a\ s)]$

`Skip` and `Assign` are just names for the clauses of the inductive definition.

$s[x ::= v]$ is short for <u>*update s x v*</u>, where

$$\mathit{update}\ s\ x\ v \equiv \lambda y.\ \mathit{if}\ y = x\ \mathit{then}\ v\ \mathit{else}\ (s\ y)$$

Note that a is of type *aexp* (➜ p.595).

Inductive Definition: Semicolon

$$\texttt{Semi} : \llbracket \langle c_0, s\rangle \xrightarrow{c} s_1 ; \langle c_1, s_1\rangle \xrightarrow{c} s_2 \rrbracket$$
$$\Longrightarrow \langle c_0; c_1, s\rangle \xrightarrow{c} s_2$$

The rationale of natural semantics: To figure out the meaning of a program consisting of a “first instruction” c_0 and a “rest” c_1, starting from state s, you have to show two subgoals: c_0 starting from state s goes to some state s_1, and c_1 starting in state s_1 goes to some state s_2.

Note that by the definition of `Semi` (➜ p.594), c_0 does not have to be “atomic” (whatever this means).

Inductive Definition: Control

$\texttt{IfTrue}$: $\llbracket b\ s; \langle c_0, s\rangle \xrightarrow{c} s_1\rrbracket$
$\Longrightarrow \langle \texttt{IF}\ b\ \texttt{THEN}\ c_0\ \texttt{ELSE}\ c_1, s\rangle \xrightarrow{c} s_1$

$\texttt{IfFalse}$: $\llbracket \neg b\ s; \langle c_1, s\rangle \xrightarrow{c} s_1\rrbracket$
$\Longrightarrow \langle \texttt{IF}\ b\ \texttt{THEN}\ c_0\ \texttt{ELSE}\ c_1, s\rangle \xrightarrow{c} s_1$

$\texttt{WhileFalse}$: $\llbracket \neg b\ s\rrbracket \Longrightarrow \langle \texttt{WHILE}\ b\ \texttt{DO}\ c, s\rangle \xrightarrow{c} s$

$\texttt{WhileTrue}$: $\llbracket b\ s; \langle c, s\rangle \xrightarrow{c} s_1; \langle \texttt{WHILE}\ b\ \texttt{DO}\ c, s_1\rangle \xrightarrow{c} s_2\rrbracket$
$\Longrightarrow \langle \texttt{WHILE}\ b\ \texttt{DO}\ c, s\rangle \xrightarrow{c} s_2$

Note the termination problem in `WhileTrue`! Simplest example: $b \equiv \lambda x.\mathit{True}$. Then, no proof is possible and no s_2 can effectively be computed.

27.4 Embedding of the Transition Semantics

The transition semantics encoding in Isabelle is also (➜ p.598) given by an inductive definition. We first declare its type and define a paraphrasing, as before (➜ p.598):

consts $\mathit{evalc1} :: \;\text{"}((\mathit{com} * \mathit{state}) * (\mathit{com} * \mathit{state}))\ \mathit{set}\text{"}$

translations $\text{"}cs_0 \xrightarrow{1} cs_1\text{"} \equiv \text{"}(cs_0, cs_1) \in \mathit{evalc1}\text{"}$

Note that $\xrightarrow{1}$ is one fixed (➜ p.605) arrow symbol.
We now start giving the actual inductive definition . . .

Inductive Definition

`inductive` evalc1

`intrs`

`Assign:` ”$(x :== a, s) \xrightarrow{1} (\texttt{SKIP}, s[x ::= (a\ s)])$”

`Semi1:` ”$(\texttt{SKIP}; c, s) \xrightarrow{1} (c, s)$”

`Semi2:` ”$(c_0, s) \xrightarrow{1} (c_0', s') \Longrightarrow (c_0; c_1, s) \xrightarrow{1} (c_0'; c_1, s')$”

So far, we see that the component of *com* type in the configuration corresponds to a program stack (built by ”;” (➜ p.594)), which represents a program counter (➜ p.598).

Inductive Definition: Control

$\texttt{IfTrue:}$ $\quad$ ”$b\ s \Longrightarrow (\texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2, s) \xrightarrow{1} (c_1, s)$”

$\texttt{IfFalse:}$ $\quad$ ”$\neg b\ s \Longrightarrow (\texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2, s) \xrightarrow{1} (c_2, s)$”

$\texttt{WhileFalse:}$ $\quad$ ”$\neg b\ s \Longrightarrow (\texttt{WHILE}\ b\ \texttt{DO}\ c, s) \xrightarrow{1} (\texttt{SKIP}, s)$”

$\texttt{WhileTrue:}$ $\quad$ ”$b\ s \Longrightarrow (\texttt{WHILE}\ b\ \texttt{DO}\ c, s) \xrightarrow{1} (c; \texttt{WHILE}\ b\ \texttt{DO}\ c, s)$”

Termination problem as before (➜ p.601), but somehow less disturbing: we cannot be shocked about the fact that some computations are infinite, and at least, the transition semantics assigns a meaning to any finite prefix of an infinite computation.

Generalizations to more than one Step

n-step semantics:

$$\text{"}cs_0 \xrightarrow{n} cs_1\text{"} \equiv \text{"}(cs_0, cs_1) \in \textit{evalc1}^n\text{"}$$

Unlike $\xrightarrow{c}$ (➜ p.598) and $\xrightarrow{1}$ (➜ p.602), $\xrightarrow{n}$ is not a fixed arrow symbol, but meta-notation: for any number n, there is the paraphrasing[495] $\xrightarrow{n}$ defined as above. Here, $\textit{evalc1}^n$ (ASCII: `^n`) is defined in `Relation_Power.thy`[496].

multistep-semantics:

$$\text{"}cs_0 \xrightarrow{*} cs_1\text{"} \equiv \text{"}(cs_0, cs_1) \in \textit{evalc1}^*\text{"}$$

$\xrightarrow{*}$ is a fixed arrow symbol.

[495] As you see, paraphrasing in Isabelle is very powerful. One can think of $\xrightarrow{c}$ (➜ p.598) and $\xrightarrow{1}$ (➜ p.602) as infix symbols (➜ p.65). But $\xrightarrow{n}$ is by no means one single symbol. In fact the term $cs_0 \xrightarrow{n} cs_1$ is a paraphrasing of $(cs_0, cs_1) \in \textit{evalc1}^n$.

[496] 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/`

Equivalence of Semantics

Natural semantics vs. transition semantics.

Theorem (`evalc1_eq_evalc`):

$$(c, s) \xrightarrow{*} (\texttt{SKIP}, t) = (\langle c, s\rangle \xrightarrow{c} t)$$

The proof is by induction on the structure of programs.

27.5 Embedding of the Denotational Semantics

Domain: A semantics relates states (similar to natural (➜ p.597) semantics)

$$com\_den = (state * state)\ set$$

Semantic <u>function</u>: assigns semantics to a program

$$consts\ C :: com \Rightarrow com\_den$$

Before (➜ p.597), semantics were <u>relations</u>.

Characteristics of Denotational Semantics

A denotational semantics is a function (here: C) assigning a meaning to a program. More precisely, the meaning of a program is some "mathematical" function of the meanings of its components.

This is in contrast to the operational view where computation order ("first do this, then that…") and logical reasoning using proof rules ("if (…) computes (…) then (…) computes (…)") are focused.

The "mathematics" uses the *lfp* (➜ p.468) operator.

The Recursive Definition

The semantics C is defined recursively[497]:

```
primrec
  C_skip     "$C(\texttt{SKIP}) = Id$"
  C_assign   "$C(x :== a) = \{(s,t) \mid t = s[x ::= (a\ s)]\}$"
  C_comp     "$C(c_0; c_1) = C(c_1) \circ C(c_0)$"
  C_if       "$C(\texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2) =$
             $\{(s,t) \mid (s,t) \in C(c_1) \wedge b(s)\} \cup$
             $\{(s,t) \mid (s,t) \in C(c_2) \wedge \neg b(s)\}$"
  C_while    "$C(\texttt{WHILE}\ b\ \texttt{DO}\ c) = lfp(\Gamma\ b\ (C\ c))$"
```

where[498] "$\Gamma\ b\ cd \;\equiv\; (\lambda\phi.\{(s,t) \mid (s,t) \in (\phi \circ cd) \wedge b(s)\} \cup \{(s,t) \mid s = t \wedge \neg b(s)\})$"

[497] Recall (➜ p.541) that the `primrec` syntax is used for defining functions recursively. Here, the argument type of the function C is the datatype *com* (➜ p.594). It is characteristic for the definition of a datatype that its elements are defined by (structural) induction, i.e., its elements are syntactic terms formed from previously generated syntactic forms using a specific set of term constructors. For datatypes, it is clear that the subterm relation is a well-founded order. Hence it is legitimate to define C using recursion.

Equivalence of Programs

We have seen an equivalence result relating different semantics (➜ p.606).

The following is an equivalence relating program fragments.

Theorem (`C_While_If`):
$C(\texttt{WHILE}\ b\ \texttt{DO}\ c) = C(\texttt{IF}\ b\ \texttt{THEN}\ (c; \texttt{WHILE}\ b\ \texttt{DO}\ c)\ \texttt{ELSE}\ \texttt{SKIP})$

Such a result is important because it justifies a program transformation (the two fragments have the same semantics and so they are interchangeable).

Equivalence of Semantics

We have already suggested that the natural semantics is a hybrid (➜ p.597) between operational and denotational semantics. In fact, there is a simple equivalence relationship between the two:

Theorem (denotational is natural):

$((s,t) \in C\ c) = (\langle c, s\rangle \xrightarrow{c} t)$

27.6 Axiomatic (Hoare) Semantics

Idea: we relate "legal states" before and after a program execution. A set of legal states is modeled as "assertion":

$$\texttt{types}\ \mathit{assn} = \mathit{state} \Rightarrow \mathit{bool}$$

So rather than reasoning about single states, we reason about properties or sets of states. This is what we really need for verification of programs.

Semantics called axiomatic for historic reasons[499]. It is also called Hoare semantics.

[499] In terms of Isabelle/HOL, the semantics is not defined by axioms, but is an inductive definition (➜ p.491).

Embedding of the Hoare Semantics

The Hoare semantics encoding in Isabelle is also (➜ p.598) given by an inductive definition. We first declare its type and a paraphrasing:

`consts` hoare :: $''(\mathit{assn} * \mathit{com} * \mathit{assn})\ \mathit{set}''$
`translations` $'' \vdash \{P\}\, c\, \{Q\}'' \equiv ''(P, c, Q) \in \mathit{hoare}''$

An object of the form $\{P\}\, c\, \{Q\}$ is called a Hoare-triple. We now start giving the actual inductive definition . . .

Inductive Definition: `SKIP`

```
inductive hoare
 intrs
   skip  "⊢ {P} SKIP {P}"
```

No surprise here.

The Inductive Definition

`ass` $'' \vdash \{\lambda s.P(s[x ::= (a\ s)])\}\ x :== a\ \{P\}''$

This may be counter-intuitive, why not the other way round?

Consider an example: $a \equiv \lambda s.1$ and $P \equiv \lambda s.\ s\ x = 1$

$\{\lambda s.\underline{(\lambda s.s\ x = 1)(s[x ::= 1])}\}\ x :== \lambda s.1\ \{\lambda s.s\ x = 1\} \longrightarrow_\beta$

$\{\lambda s.\underline{(s[x ::= 1])\ x} = 1\}\ x :== \lambda s.1\ \{\lambda s.s\ x = 1\} \longrightarrow_\beta$

$\{\lambda s.(1 = 1)\}\ x :== \lambda s.1\ \{\lambda s.s\ x = 1\} \longrightarrow_\beta$

$\{\lambda s.\mathit{True}\}\ x :== \lambda s.1\ \{\lambda s.s\ x = 1\}$

What do we see? (You might also check the types[500].)

The *ass* rule is such that it relates the pre-state *True* with the post-state $\lambda s.\ s\ x = 1$, which is what we expect[501].

[500] Things are getting a bit complicated, maybe it helps to recall the types of the terms occurring in

$$ass \qquad '' \vdash \{\lambda s.P(s[x ::= (a\ s)])\}\ x :== a\ \{P\}''$$

P has type *assn*, which is (➜ p.612) *state* $\Rightarrow$ *bool*. In turn , *state* is (➜ p.594) *loc* $\Rightarrow$ *val*.

x has type *loc* (➜ p.594).

a has type *aexp*, which is (➜ p.594) *state* $\Rightarrow$ *val*.

s has type *state*.

[501] You can also argue a bit more generally. Let Q be an arbitrary assertion, and let

$$P \equiv \lambda s.\ \exists s'.\ s = s'[x ::= (a\, s')] \wedge Q\ s'$$

Intuitively: P is an assertion allowing any state obtained from a state allowed by Q by updating that state at location x with the expression a. Now consider the rule for assignment:

$$ass \qquad '' \vdash \{\lambda s.P(s[x ::= (a\ s)])\}\ x :== a\ \{P\}''$$

Inductive Definition: *Semi* and IF − THEN − ELSE

`semi` "$[\![\vdash \{P\}\, c\, \{Q\}; \vdash \{Q\}\, d\, \{R\}]\!] \Longrightarrow \vdash \{P\}\, c; d\, \{R\}$"
`If` "$[\![\vdash \{\lambda s. P\ s \wedge b\ s\}\, c\, \{Q\}; \vdash \{\lambda s. P\ s \wedge \neg b\ s\}\, d\, \{Q\}]\!]$
$\Longrightarrow \vdash \{P\}$ `IF` b `THEN` c `ELSE` $d\, \{Q\}$"

Since we are reasoning about sets of states, $b\, s$ may sometimes be true and sometimes false, and so we have two premises for those two cases. It turns out that if $b\ s$ is trivially true or trivially false, then one of the premises will be trivial to prove.

---

in particular the assertion on the left-hand side. It reduces as follows:

$\lambda s.\ \underline{P(s[x ::= (a\, s)])} \longrightarrow_\beta$
$\lambda s.\ (\exists s'.\ Q\ s' \wedge s[x ::= (a\, s)] = s'[x ::= (a\, s')]) \longrightarrow_\beta \ldots$
$\lambda s.\ (\exists s'.\ Q\ s' \wedge s = s') \longrightarrow_\beta \ldots \lambda s.\ (Q\ s) \longrightarrow_\eta Q$

So you see that any pre-state Q will be related to a post-state P as given above.

By this argument, we have only shown which post-states <u>are</u> possible given an arbitrary pre-state, not which post-states <u>are not</u>. Such an argument is more complicated.

Inductive Definition: WHILE

`While` $"\vdash \{\lambda s. P\ s \wedge b\ s\}\ c\ \{P\} \Longrightarrow$
$\vdash \{P\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \{\lambda s. P\ s \wedge \neg b\ s\}"$

This has a flavor of loop invariants: in the pre-state, $b\ s$ holds, in the post-state, $b\ s$ does not hold, and P holds all the time.

Inductive Definition: Weakening and Strengthening

`conseq` "$[\![\forall s. P's \rightarrow P\ s; \vdash \{P\}\, c\, \{Q\}; \forall s. Q\ s \rightarrow Q'\ s]\!]$
$\Longrightarrow \vdash \{P'\}\, c\, \{Q'\}$"

One can always strengthen the pre-condition or weaken the post-condition.

The Rules at a Glance

`inductive` hoare

`intrs`

`skip` $\ ''\vdash \{P\}\,\texttt{SKIP}\,\{P\}''$

`ass` $\ ''\vdash \{\lambda s.P(s[x ::= a\ s])\}\ x :== a\ \{P\}''$

`semi` $\ ''[\![\vdash \{P\}\ c\ \{Q\}; \vdash \{Q\}\ d\ \{R\}]\!] \Longrightarrow \vdash \{P\}\ c; d\ \{R\}''$

`If` $\ ''[\![\vdash \{\lambda s.P\ s \wedge b\ s\}\ c\ \{Q\}; \vdash \{\lambda s.P\ s \wedge \neg b\ s\}\ d\ \{Q\}]\!] \Longrightarrow$
$\vdash \{P\}\ \texttt{IF}\ b\ \texttt{THEN}\ c\ \texttt{ELSE}\ d\ \{Q\}''$

`While` $\ ''\vdash \{\lambda s.P\ s \wedge b\ s\}\ c\ \{P\} \Longrightarrow$
$\vdash \{P\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \{\lambda s.P\ s \wedge \neg b\ s\}''$

`conseq` $\ ''[\![\forall s.P's \to P\ s; \vdash \{P\}\ c\ \{Q\}; \forall s.Q\ s \to Q'\ s]\!] \Longrightarrow$
$\vdash \{P'\}\ c\ \{Q'\}''$

Validity Relation

We define a validity relation:

$$\models \{P\}\, c\, \{Q\} \equiv \forall s\, t.(s,t) \in C(c) \rightarrow (P\ s) \rightarrow (Q\ t)$$

A Hoare triple $\{P\}c\{Q\}$ is valid if it relates a set of input states and a set of output states correctly w.r.t. the denotational (or equivalently (➜ p.611), operational) semantics: for any input state s and output state t related by the denotational semantics (➜ p.607), if P holds for s, then Q must hold for t.

Why[502] do we raise the issue of a semantics being valid, why don't we just say "it's defined like this, full stop"?

[502] You may wonder: Why do we raise the issue of a semantics being valid, why don't we just say "it's defined like this, full stop"? After all, we didn't question the operational and denotational semantics in the same way. So why do we take the denotational semantics as the real semantics of a program that another semantics such as the Hoare semantics has to be somehow equivalent to in order to be correct? Couldn't we do it the other way round?

First: If you want to accept anything as the real semantics of a program, it would be the transition semantics (➜ p.602), since we believe that by the transition semantics, we have modeled what the compiler of the programming language actually does. The transition semantics records the actual computation steps (➜ p.597).

Secondly, we have shown that the transition semantics is equivalent to the natural semantics (➜ p.606), which in turn is equivalent to the denotational semantics (➜ p.611).

Thirdly, someone might claim that the Hoare semantics

Relating Hoare and Denotational Semantics

Theorem (Hoare soundness):

$$\vdash \{P\}\, c\, \{Q\} \Longrightarrow \models \{P\}\, c\, \{Q\}$$

Theorem (Hoare relative completeness):

$$\models \{P\}\, c\, \{Q\} \Longrightarrow \vdash \{P\}\, c\, \{Q\}$$

Why relative[503]?

So the Hoare relation is in fact compatible with the denotational semantics of IMP.

"obviously" reflects the real semantics of a program, but that would seem quite far-fetched, because the semantics speaks about properties of states rather than about states directly.

Together this explains why we call a Hoare triple valid if it is correct w.r.t. the denotational semantics.

[503] We will not give any details here, but the completeness result is restricted in the same way that the completeness of HOL (➜ p.354) is restricted to general models, as opposed to standard models.

27.7 Example Program

```
$tm :== \lambda x.1;$
$sum :== \lambda x.1;$
$i :== \lambda x.0;$
WHILE $\lambda s.(s\ sum) <= (s\ a)$ DO
   $(i :== \lambda s.(s\ i) + 1;$
   $tm :== \lambda s.(s\ tm) + 2;$
   $sum :== \lambda s.(s\ tm) + (s\ sum))$
```

What does this program do?

Try $a = 1$, $a = 2$, …, and look at i![504] (Note that a is of type *nat* and hence $a \geq 0$.)

[504] a is not modified anywhere. You should think of a as input of the program.

i counts the number of times the loop is entered, i.e. the final value of i is the number of times the loop was entered. This number depends on a. The following table shows that final values of i, *tm* and *sum* depending on the value of a:

	i	tm	sum
$0 \leq a < 1$	0	1	1
$1 \leq a < 4$	1	3	4
$4 \leq a < 9$	2	5	9
$9 \leq a < 16$	3	7	16
$16 \leq a < 25$	4	9	25
$25 \leq a < 36$	5	11	36
$36 \leq a < 49$	6	13	49

sum takes the values of all squares successively, computed by the famous binomial formula:

$$(i + 1)^2 = i^2 + 2i + 1$$

Since *tm* takes the value $2i+1$ for all i successively, it follows that $sum + tm$ always gives the next value of *sum*.

Square Root

Answer: The program computes the square root. Informally:

$$\begin{aligned} \mathit{Pre} &\equiv \text{”}\mathit{True}\text{”} \\ \mathit{Post} &\equiv \text{”}i^2 \leq a < (i+1)^{2}\text{”} \end{aligned}$$

Formally

$$\begin{aligned} \mathit{Pre} &\equiv \lambda s.\ \ \mathit{True} \\ \mathit{Post} &\equiv \lambda s.\ \ (s\, i)^{505} * (s\, i) \leq (s\, a) \wedge \\ & s\, a < (s\, i + 1) * (s\, i + 1) \end{aligned}$$

Proving $\{Pre\}\ldots\{Post\}$

We will now construct a proof tree showing that the program computes the square root.

Generally, the difficulty[506] is to know when to apply *conseq* (➜ p.619).

We try to illustrate the search for the proof tree by animation. Still you may not understand each choice immediately, but only in hindsight!

We use two metavariables: *Inv* for the loop invariant, *PW* for the enter condition of the loop. We instantiate later.

Abbreviation: $ExC \equiv \lambda s.Inv\ s \wedge \neg s\ sum \leq s\ a$ ("exit condition"). We omit $\vdash$!

[506] The *conseq* (➜ p.618) rule can always be applied. If one decides not to apply the *conseq* rule, then the choice of any other rule (➜ p.619) is deterministic.

Proof

$$\dfrac{\boxed{\mathcal{I}_1}^{508} \qquad \dfrac{\boxed{\mathcal{A}_1}^{510} \qquad \dfrac{\boxed{\mathcal{A}_2}^{513} \qquad \dfrac{\boxed{\mathcal{A}_3}^{516} \qquad \dfrac{\boxed{\mathcal{I}_3}^{518} \quad \{Inv\}\boxed{WH\ldots}\{ExC\} \quad \boxed{\mathcal{I}_4}^{519}}{\{PW\}\boxed{WH\ldots}^{517}\{ExC\}}\;\mathit{conseq}}{\{\lambda s.PW(s[''i'']^{514})\}\boxed{i\ldots}^{515}\{ExC\}}\;\mathit{semi}}{\{\lambda s.PW(s[''i,sum'']^{511})\}\boxed{sum\ldots}^{512}\{ExC\}}\;\mathit{semi}}{\{\lambda s.PW(s[''i,sum,tm'']^{509})\}\boxed{tm\ldots}\{ExC\}}\;\mathit{semi} \qquad \boxed{\mathcal{I}_2}^{520}}{\{Pre\}\boxed{tm\ldots}^{507}\{Post\}}\;\mathit{conseq}$$

This is what we want to prove.

Nothing happens after the loop, so intuition says (➜ p.624) that ExC must imply $Post$.

Apply *semi* three times. PW ("pre while") is just a sensible choice of name: we don't know yet what it is.

This application of *ass* (➜ p.615) will allow us to reconstruct the precondition in the line just below.

And likewise $\boxed{\mathcal{A}_2}$.

And likewise $\boxed{\mathcal{A}_1}$.

We now know (by the form of *conseq*) what $\boxed{\mathcal{I}_1}$ is.

Intuition says (➜ p.624) that PW must imply Inv.
Of course, we are not ready yet...

Completing the Proof

$\boxed{\mathcal{A}_1}$ (➜ p.625), $\boxed{\mathcal{A}_2}$ (➜ p.625) and $\boxed{\mathcal{A}_3}$ (➜ p.625) are complete, and $\boxed{\mathcal{I}_4}$ (➜ p.625) is trivial.

$\boxed{\mathcal{I}_1}$ (➜ p.625), $\boxed{\mathcal{I}_2}$ (➜ p.625), $\boxed{\mathcal{I}_3}$ (➜ p.625), and $\{Inv\}\boxed{WH \ldots}$ (➜ p.625)$\{ExC$ (➜ p.624)$\}$ remain to be shown.

This also involves the question of how the metavariables must be instantiated.

What is PW?

The metavariable PW (“precondition of `WHILE` ”) must fulfill (to show $\boxed{\mathcal{I}_1}$ (➜ p.625))

$\forall s.Pre$ (➜ p.623) $s \to PW(s[i ::= 0][sum ::= 1][tm ::= 1])$

where

$$\begin{array}{l} s[i ::= 0][sum ::= 1][tm ::= 1] \text{ (➜ p.599)} = \\ \quad \lambda y.\ \mathit{if}\ y = tm\ \mathit{then}\ 1\ \mathit{else} \\ \qquad (\mathit{if}\ y = sum\ \mathit{then}\ 1\ \mathit{else}(\mathit{if}\ y = i\ \mathit{then}\ 0\ \mathit{else}\ (s\ y))) \end{array}$$

Solution (recall (➜ p.623) that $Pre \equiv \lambda s.\mathit{True}$):

$$PW = \lambda s.s\ i = 0 \land s\ sum = 1 \land s\ tm = 1$$

What is *Inv*?

Continuing our proof tree construction:

$$\dfrac{\dfrac{\begin{array}{c}\{\lambda s.\mathit{Inv}\ s \wedge s\ \mathit{sum} \leq s\ a\} i :== \lambda s.s\ i + 1 \{P'\} \\ \{P'\} \mathit{tm} :== \lambda s.s\ \mathit{tm} + 2 \{P''\} \\ \{P''\} \mathit{sum} :== \lambda s.s\ \mathit{tm} + s\ \mathit{sum} \{\mathit{Inv}\}\end{array}}{\{\lambda s.\mathit{Inv}\ s \wedge s\ \mathit{sum} \leq s\ a\} \boxed{\text{"body"}}^{521} \{\mathit{Inv}\}}\ \mathit{semi}^2}{\{\mathit{Inv}\} \boxed{\mathit{WH}\ldots}\ (\text{➜ p.625}) \{\mathit{ExC}\ (\text{➜ p.624})\}}\ \mathit{While}\ (\text{➜ p.617})$$

Just blindly applying *semi* twice gives three formulas[522] to be proven using *ass* (➜ p.615), one for each assignment in the loop.

Now what are P' and P''? Have a look at rule *ass* (➜ p.615) first!

[522] Of course, these three formulas should be side by side in the proof tree, but this cannot be displayed.

Calculating P' and P'' (by Rule *ass*)

$$P'' = \lambda s.Inv(s[sum ::= s\ tm + s\ sum])$$

$$\begin{aligned}
&P' = \lambda s'.P''(s'[tm ::= s'\ tm + 2]) \qquad \text{(rule } \mathit{ass} \text{ (➜ p.615))}\\
&= \lambda s'.(\lambda s.Inv(s[sum ::= s\ tm + s\ sum]))\\
&\qquad\qquad (s'[tm ::= s'\ tm + 2])\\
&= \lambda s'.Inv((s'[tm ::= s'\ tm + 2])\\
&\qquad\qquad\qquad [sum ::= (s'[tm ::= s'\ tm + 2])\ tm +\\
&\qquad\qquad\qquad\qquad (s'[tm ::= s'\ tm + 2])\ sum])\\
&= \lambda s'.Inv(s'[tm ::= s'\ tm + 2]\\
&\qquad\qquad [sum ::= s'\ tm + 2 + s'\ sum]).
\end{aligned}$$

Applying ass to $i :== \lambda s.s\,i + 1$

Now treat $i :== \lambda s.s\ i + 1$ in the same way. Temporarily, let's write $\underline{P}$ for $\lambda s.Inv\ s \wedge s\,sum \leq s\,a$ (➜ p.629). Recall $P' =$ (➜ p.630)

λs (➜ p.158).$Inv(s$ (➜ p.158)$[tm ::= s$ (➜ p.158) $tm + 2][sum ::= s$ (➜ p.158) $tm + 2 + s$ (➜ p.158) $sum])$.

$$\begin{aligned}
&P = \lambda s'.P'(s'[i ::= s'\ i + 1]) \qquad \text{(by rule } ass \text{ (➜ p.615))}\\
&= \lambda s'.(\lambda s.Inv(s[tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum]))\\
&\qquad\qquad\qquad\qquad (s'[i ::= s'\ i + 1])\\
&= \lambda s'.Inv((s'[i ::= s'\ i + 1])\\
&\qquad\quad [tm ::= (s'[i ::= s'\ i + 1])\ tm + 2]\\
&\qquad\quad [sum ::= (s'[i ::= s'\ i + 1])\ tm + 2 + (s'[i ::= s'\ i + 1])\ sum]))\\
&= \lambda s\ \text{(➜ p.158)}.Inv(s[i ::= s\ i + 1][tm ::= s\ tm + 2]\\
&\qquad\qquad\qquad\qquad [sum ::= s\ tm + 2 + s\ sum]).
\end{aligned}$$

So *Inv* must solve[523] this equation.

[523] Recall (➜ p.629) that we had to prove the three formulas

$$\{\lambda s.Inv\ s \wedge s\ sum \leq s\ a\}i :== \lambda s.s\ i + 1\{P'\}$$
$$\{P'\}tm :== \lambda s.s\ tm + 2\{P''\}$$
$$\{P''\}sum :== \lambda s.s\ tm + s\ sum\{Inv\}$$

all by *ass* (➜ p.615). Dealing with the second and third formula using *ass*, we found that

$$P' = \lambda s'.Inv(s'[tm ::= s'\ tm{+}2][sum ::= s'\ tm + 2{+}s'\ sum]).$$

Therefore, to show

$$\{\lambda s.Inv\ s \wedge s\ sum \leq s\ a\}i :== \lambda s.s\ i + 1\{P'\}$$

as well, *Inv* must have such a form that the formula becomes an instance of *ass*.

Inv Must Fulfill the Equation

Inv must fulfill the equation

$$\begin{aligned}&\lambda\forall s.\mathit{Inv}\ s \land s\ \mathit{sum} \leq s\ a = \ \leftrightarrow\\ &\lambda\forall s.\mathit{Inv}(s[i ::= s\ i + 1][\mathit{tm} ::= s\ \mathit{tm} + 2]\\ &\qquad\qquad [\mathit{sum} ::= s\ \mathit{tm} + 2 + s\ \mathit{sum}])\end{aligned}$$

Don't think syntactically! We are in HOL (➜ p.373): $=$ means $\leftrightarrow$, and we can replace λ by $\forall$ (➜ p.362).

Guessing the right *Inv* is obviously difficult! Informally (➜ p.623)

$$\mathit{Inv} \ \equiv \ "(i+1)^2 = \mathit{sum} \ \land \ \mathit{tm} = (2 * i) + 1 \ \land \ i^2 \leq a"$$

Checking that *Inv* Fulfills Equation

$$
\begin{aligned}
s\,sum \leq s\,a &\quad \wedge \qquad (6)\\
(s\,i + 1)^2 = (s\,sum) &\quad \wedge \qquad (7)\\
s\,tm = (2 * (s\,i)) + 1 &\quad \wedge \qquad (8)\\
(s\,i)^2 \leq (s\,a) &\quad \wedge \qquad (9)\\
\text{(recall: = means } \leftrightarrow\text{)} &\quad = \qquad (10)\\
((s\,i + 1) + 1)^2 = (s\,sum) + (s\,tm) + 2 &\quad \wedge \qquad (11)\\
(s\,tm + 2) = (2 * (s\,i + 1)) + 1 &\quad \wedge \qquad (12)\\
(s\,i + 1)^2 \leq (s\,a) &\qquad\qquad (13)
\end{aligned}
$$

Proof Sketch

First show the “→”-direction:

(8) → (12) and (6) $\wedge$ (7) → (13) by simple arithmetic. (11) is shown as follows:

$$\begin{aligned}((s\,i+1)+1)^2 &= (s\,i+1)^2+2*(s\,i+1)+1\\ &\overset{(7)}{=} (s\;sum)+2(s\;i)+1+2\\ &\overset{(8)}{=} (s\;sum)+(s\;tm)+2\end{aligned}$$

Proof Sketch (Cont.)

Now show the "←"-direction:

(12) → (8) and (13) → (9) by simple arithmetic. (7) is shown as follows:

$$\begin{aligned}
(s\,i+1)^2 &= ((s\,i+1)+1)^2 - 2*(s\,i+1) - 1 \\
&\overset{(11)}{=} (s\;sum) + (s\;tm) + 2 - 2*(s\,i+1) - 1 \\
&\overset{(12)}{=} (s\;sum) + 2*(s\,i+1) + 1 \\
&\qquad\qquad -2*(s\,i+1) - 1 \\
&= s\;sum
\end{aligned}$$

Finally, (7) ∧ (13) → (6). So *Inv* (➜ p.633) is indeed an invariant!

The WHILE Loop: Remarks

We have shown (➜ p.633)

$$(\text{“enter condition”} \wedge \text{“invar. at entry”}) \leftrightarrow \text{“invar. at exit”}$$

One would definitely expect $\rightarrow$, but $\leftarrow$ is remarkable!

We can show this because our invariant is so strong: for showing $\rightarrow$, the weaker invariant (7) $\wedge$ (8), i.e.

$$"(i+1)^2 = sum \;\wedge\; tm = (2 * i) + 1$$

would do (check it!).

But the extra condition $i^2 \leq a$ is needed for showing *Post* (➜ p.623), which states what the program actually computes.

Taking Care of *Post*

We have shown $\boxed{\mathcal{I}_1}$ (➜ p.625) and $\{$*Inv* (➜ p.633)$\}$ $\boxed{\mathit{WH}\ldots}$ (➜ p.625)$\{$*ExC* (➜ p.624)$\}$. Now continue with $\boxed{\mathcal{I}_2}$ (➜ p.625).

Does *Post* (➜ p.623) s follow from *Inv* (➜ p.633) $s \wedge \neg s\;\mathit{sum} \leq s\;a$?

Yes!

$$\begin{array}{ll} (s\;i)^2 \leq (s\;a) & \text{follows from (9)} \\ (s\;a) < (s\;i+1)^2 & \text{follows from } \neg s\;\mathit{sum} \leq (s\;a) \text{ and (7).} \end{array}$$

The Final Missing Part

$\boxed{\mathcal{I}_3}$ (➜ p.625) remains to be shown, i.e.

$$\forall s. PW\ s \rightarrow Inv \text{ (➜ p.633) } s$$

or, expanding the solutions for PW (➜ p.628) and Inv (➜ p.633)

$$\begin{array}{ll}\forall s. & s\,i = 0 \wedge s\,sum = 1 \wedge s\,tm = 1 \rightarrow \\ & (s\,i + 1)^2 = s\,sum \wedge \\ & s\,tm = (2 * (s\,i)) + 1 \wedge \\ & (s\,i)^2 \leq (s\,a)\end{array}$$

This is easy to check.

An Alternative for Tackling the Loop Part

Recall that our loop invariant was “too strong” (➜ p.637). An alternative:

$$\dfrac{\dfrac{\begin{array}{l}\forall s.(\mathit{Inv}\ s\wedge\\ s\ \mathit{sum} \le s\ a)\rightarrow\\ \mathit{Inv}'\ s\end{array} \qquad \dfrac{\begin{array}{c}\{\mathit{Inv}'\}i :== \lambda s.s\ i+1\{P'\}\\ \{P'\}\mathit{tm} :== \lambda s.s\ \mathit{tm} + 2\{P''\}\\ \{P''\}\mathit{sum} :== \lambda s.s\ \mathit{tm} + s\ \mathit{sum}\{\mathit{Inv}\}\end{array}}{\{\mathit{Inv}'\}\boxed{\text{"body"}}\ (\text{➜ p.629})\{\mathit{Inv}\}}\ \mathit{semi}^2}{\{\lambda s.\mathit{Inv}\ s\wedge s\ \mathit{sum}\le s\ a\}\boxed{\text{"body"}}\ (\text{➜ p.629})\{\mathit{Inv}\}}\ \mathit{conseq}}{\{\mathit{Inv}\}\boxed{\mathit{WH}\ldots}\ (\text{➜ p.625})\{\mathit{ExC}\ (\text{➜ p.624})\}}\ \mathit{While}\ (\text{➜ p.617})$$

Alternative (Cont.)

Applying *ass* (➜ p.615) as before gives

$$\begin{aligned}Inv' = \lambda s.Inv(s[i ::= s\,i + 1][tm ::= s\,tm + 2]\\ [sum ::= s\,tm + 2 + s\,sum])\end{aligned}$$

We are left with the proof obligation

$$\begin{aligned}\forall s.(Inv\ s \wedge s\,sum \leq s\,a) \to Inv(s[i ::= s\,i + 1]\\ [tm ::= s\,tm + 2][sum ::= s\,tm + 2 + s\,sum])\end{aligned}$$

Just this could be shown setting weak $Inv \equiv$ (➜ p.637) (7) $\wedge$ (8), but for actually showing *Post* (➜ p.623), $i^2 \leq a$ is still needed.

27.8 Automating Hoare Proofs

In the example (➜ p.622), we have verified a program computing the square root.

But this was tedious, and parts of the task can be automated.

Weakest Liberal Preconditions

Observation: the Hoare relation is deterministic to a certain extent.

Idea: we use this fact for the generation of (weakest liberal) preconditions.

Weakest liberal preconditions are:

$$\texttt{constdefs}\ wp :: \ com \Rightarrow assn \Rightarrow assn$$
$$\text{''}wp\ c\ Q \equiv \ (\lambda s.\forall t.(s,t) \in C(c) \rightarrow Q\ t)\text{''}$$

So $wp\ c\ Q$ returns the set of states (➜ p.612) containing all states s such that if t is reached from s via c, then the post-condition Q holds for t. Computable? Not obvious.

Equivalence Proofs

Main results of the wp-generator are:

wp_SKIP:	$wp\ \texttt{SKIP}\ Q = Q$
wp_Ass:	$wp\ (x :== a)\ Q = (\lambda s.\ Q\ (s[x ::= a\ s]))$
wp_Semi:	$wp\ (c; d)\ Q = wp\ c\ (wp\ d\ Q)$
wp_If:	$wp\ (\texttt{IF}\ b\ \texttt{THEN}\ c\ \ \texttt{ELSE}\ d)\ Q = (\lambda s.(b\ s \to wp\ c\ Q\ s) \wedge (\neg b\ s \to wp\ d\ Q\ s))$
wp_While_True:	$b\ s \Longrightarrow wp\ (\texttt{WHILE}\ b\ \texttt{DO}\ c)\ Q\ s = wp\ (c; \texttt{WHILE}\ b\ \texttt{DO}\ c)\ Q\ s$
wp_While_False:	$\neg b\ s \Longrightarrow wp\ (\texttt{WHILE}\ b\ \texttt{DO}\ c)\ Q\ s = Q\ s$
wp_While_if:	$wp\ (\texttt{WHILE}\ b\ \texttt{DO}\ c)\ Q\ s = (\mathit{if}\ \ b\ s\ \mathit{then}\ wp(c; \texttt{WHILE}\ b\ \texttt{DO}\ c)\ Q\ s\ \mathit{else}\ Q\ s)$

Last case summarises the two before.

WP-Semantics

Except for termination problem due to *While*, (weakest liberal) precondition *wp* can be computed.

This fact can be used for further proof support by verification condition generation.

Verification Condition Generation

First, we must enrich the syntax by loop-invariants:

```
datatype acom =
    Askip
  | Aass loc aexp
  | Asemi acom acom
  | Aif bexp acom acom
  | Awhile bexp assn acom
```

Almost same as *com* (➜ p.594), but *While* gets an additional argument for asserting a loop invariant. Asserting this is the difficult, creative step to be done by a human.

Computing a Weakest Liberal Precondition

We define a function that computes a wp (➜ p.643):

```
primrec
    "$awp\ Askip\ Q = Q$"
    "$awp\ (Aass\ x\ a)\ Q = (\lambda s.Q(s[x ::= as]))$"
    "$awp\ (Asemi\ c\ d)\ Q = awp\ c\ (awp\ d\ Q)$"
    "$awp\ (Aif\ b\ c\ d)\ Q = (\lambda s.(b\ s \to awp\ c\ Q\ s) \wedge$
        $(\neg b\ s \to awp\ d\ Q\ s))$"
    "$awp\ (Awhile\ b\ Inv\ c)\ Q = Inv$"
```

Idea: for all statements, the exact wp (➜ p.643) is computed, except for *While*, where the assertion provided by the user is taken as approximation. Proof obligation: show that such an assertion is compatible with the program and the desired property ...

A Verification Condition

Construct a formula $vc\ c\ Q\ s$ with the intuitive reading: as far as the invariant assertions are concerned, s is a good pre-state for reaching desired post-property Q using annotated program (➜ p.646) c.

This is not about distinguishing good pre-states from bad pre-states! It is about formalising well-chosen invariants. For an annotated program with well-chosen invariants, $\underline{\forall s}.vc\ c\ Q\ s$ holds, i.e. $vc\ c\ Q \equiv \lambda s.\mathit{True}$.

The Definition of vc

Roughly, an annotated programm has well-chosen invariants if its components have well-chosen invariants, so most of the definition is saying just that:

```
primrec
    ”$vc\ Askip\ Q = (\lambda s.True)$”
    ”$vc\ (Aass\ x\ a)\ Q = (\lambda s.True)$”
    ”$vc\ (Asemi\ c\ d)\ Q = (\lambda s.vc\ c\ (awp\ d\ Q)\ s \wedge vc\ d\ Q\ s)$”
    ”$vc\ (Aif\ b\ c\ d)\ Q = (\lambda s.vc\ c\ Q\ s \wedge vc\ d\ Q\ s)$”
    ”$vc\ (Awhile\ b\ Inv\ c)\ Q = (\lambda s.(Inv\ s \wedge \neg b\ s \to Q\ s) \wedge$
      $(Inv\ s \wedge b\ s \to awp\ c\ Inv\ s) \wedge vc\ c\ Inv\ s)$”
```

Only the case for *While* is non-trivial ...

vc: The *While* case

$$
\begin{aligned}
\text{"}vc\ (Awhile\ b\ Inv\ c) Q = \ & (\lambda s.(Inv\ s \land \neg b\ s \to Q\ s)\land \\
& (Inv\ s \land b\ s \to awp\ c\ Inv\ s)\land \\
& vc\ c\ Inv\ s)\text{"}
\end{aligned}
$$

Why is *Inv* a well-chosen invariant?

- *Inv* + exit condition imply Q: $Inv\ s \land \neg(b\ s) \to Q\ s$;
- *Inv* + loop condition imply precondition of *Inv* (so that *Inv* will hold after one execution of c): $Inv\ s \land (b\ s) \to awp\ c\ Inv\ s$.
- $vc\ c\ Inv\ s$ is in the spirit of the rest of the definition of vc: call vc recursively for the component.

Results of the wp-Generator

`vc_sound:` $\forall Q.(\forall s.vc\ ac\ Q\ s) \rightarrow$
$\vdash \{awp\ ac\ Q\}\ astrip$[524] $ac\ \{Q\}$

`vc_complete:` $\vdash \{P\}\ c\ \{Q\} \Longrightarrow \exists ac.astrip\ ac = c \wedge$
$(\forall s.vc\ ac\ Q\ s) \wedge (\forall s.P\ s \rightarrow awp\ ac\ Q\ s)$

To prove that c has property Q after execution, annotate (➜ p.646) it with loop invariants (ac) and show $\forall s.\ vc\ ac\ Q\ s$. This implies that a Hoare proof exists, for the computable precondition $awp\ ac\ Q$. For good (robust) programs, $awp\ ac\ Q = \lambda s.True$.

Summary

IMP closely follows the standard textbook [Win96].

Isabelle/HOL is a powerful framework for embedding imperative languages.

Isabelle/HOL is also a framework for state-of-the-art languages like JAVA including interfaces, inheritance, dynamic methods.

It works in theory and for non-trivial problems in practice.

28 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 (➜ p.658);
- red-black trees (➜ p.663).

This is just to stimulate you to look for more applications on your own!

28.1 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 illegal[525] 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.

[525] By "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 limited[526].
- 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.

[526] The memory on smart cards is limited. A full-fledged bytecode verifier would be too large/slow. One approach to tackling this problem is to work with bytecode programs with type annotations. Checking if a bytecode program is consistent with its type annotations is a much simpler task than computing these type annotations, which is what a bytecode verifier is supposed to do. The task can therefore be performed on a smart card more easily than full bytecode verification.

28.2 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 <u>format</u> is characterized by maximal exponent E, precision p, and exponent offset ("ulpscale") N. The set of <u>real</u> numbers representable by a triple is:

$$\begin{array}{l} \mathit{format}\ (E, p, N) = \\ \{x \mid \exists s\, e\, k.\ s < 2 \land e < E \land k < 2^p \land x = (-1)^s \cdot 2^e \cdot k/2^N\} \end{array}$$

Rounding

Rounding takes a real to a representable real nearby. E.g. rounding up:

$$\begin{aligned} \mathit{round}\ \mathit{fmt}\ x = \epsilon a.\ & a \in \mathit{format}\ \mathit{fmt} \wedge a \leq x \wedge \\ & \forall b \in \mathit{format}\ \mathit{fmt}.\ b \leq x \to b \leq a \end{aligned}$$

Formalization of the Standard [IEE85].

Useful lemmas such as:

$$x \leq y \Longrightarrow \mathit{round}\ \mathit{fmt}\ x \leq \mathit{round}\ \mathit{fmt}\ y$$

$$a \in \mathit{format}\ \mathit{fmt} \wedge b \in \mathit{format}\ \mathit{fmt} \wedge 0.5 \leq \tfrac{a}{b} \leq 2 \Longrightarrow \\ \quad (b - a) \in \mathit{format}\ \mathit{fmt}$$

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.

28.3 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 Kimmig[527] 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.

[527] Angelika Kimmig is a student who took this course in Wintersemester 02/03 in Freiburg. She then continued working with Isabelle in a Studienarbeit (a project required by computer science students in Freiburg).

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

[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.

[Ber91] Paul Bernays. Axiomatic Set Theory. Dover Publications, 1991.

[BM00] David A. Basin and Seàn Matthews. Structuring metatheory on inductive definitions. Information and Computation, 162(1-2):80–95, 2000. Download.

[BN98] Franz Baader and Tobias Nipkow. Term Rewriting and All That. Cambridge University Press, 1998.

[Can18] Georg Cantor. ?? ??, 18??

[Chu40] Alonzo Church. A formulation of the simple theory of types. Journal of Symbolic Logic, 5:56–68, 1940.

[dB80] Nicolaas G. de Bruijn. A survey of the project AUTOMATH. In Essays in Combinatory Logic, Lambda Calculus, and Formalism. Academic Press, 1980.

[Des16] Rene Descartes. ?? ??, 16??

[Dev93] Keith Devlin. The Joy of Sets. Fundamentals of Contemporary Set Theory. Undergraduate Texts in Mathematics. Springer-Verlag, 1993.

[Ebb94] Heinz-Dieter Ebbinghaus. Einführung in die Mengenlehre. BI-Wissenschaftsverlag, 1994.

[Fle00] Jacques D. Fleuriot. On the mechanization of real analysis in isabelle/hol. In Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics, volume 1869 of Lecture Notes in Computer Science, pages 145–161. Springer, 2000.

[FP98] Jacques D. Fleuriot and Lawrence C. Paulson. A combination of nonstandard analysis and geometry 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.

[Frä22] Adolf Fränkel. Zu den Grundlagen der Cantor-Zermeloschen Mengenlehre. Mathematische Annalen, 86:230–237, 1922. See [vH67].

[Fre93] Gottlob Frege. Grundgesetze der Arithmetik, volume I. Verlag Hermann Pohle, 1893. Translated in part in [**?**].

[Fre03] Gottlob Frege. Grundgesetze der Arithmetik, volume II. Verlag Hermann Pohle, 1903. Translated in part in [**?**].

[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 Systeme. Monatshefte für Mathematik und Physik, 38:173–198, 1931.

[Har98] John Harrison. Theorem Proving with the Real Numbers. Springer-Verlag, 1998.

[Har99] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, Proceedings of the 12th TPHOLs, volume 1690 of LNCS, pages 113–130. Springer-Verlag, 1999.

[Har00] John Harrison. Formal verification of the IA/64 division algorithms. In Mark Aagaard and John Harrison, editors, Proceedings of the 13th TPHOLs, volume 1869 of LNCS, pages 233–251. Springer-Verlag, 2000.

[HC68] George E. Hughes and Maxwell John Cresswell.

An Introduction to Modal Logic. Muthuen and Co. Ltd, London, 1968.

[Hen50] Henkin. Completeness in the theory of types. Journal of Symbolic Logic, 15(2):81–91, 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.

[HR04] Michael Huth and Mark Ryan. Logic in Computer Science. Modelling and Reasoning about Systems. Cambridge University Press, 2nd edition edition, 2004.

[HS90] J. Roger Hindley and Jonathan P. Seldin. Introduction to Combinators and λ-Calculus. Cambridge University Press, 1990.

[Hué] Gerard Huét. ?? ??, ??

[IEE85] The Institute of Electrical and Electronic Engineers, Inc. IEEE. Standard for binary floating point arithmetic. ANSI/IEEE Standard 754-1985, 1985.

[Kim03] Angelika Kimmig. Red-black trees of slmnj. Studienarbeit at Universität Freiburg, Download, 2003.

[Klo93] Jan Willem Klop. Handbook of Logic in Computer Science, chapter "Term Rewriting Systems". Oxford: Clarendon Press, 1993.

[KN03] Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. Theoretical Computer Science, 3(298):583–626, 2003.

[LP81] Harry R. Lewis and Christos H. Papadimitriou. 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.

[Mil92] Dale Miller. Logic, higher-order. In Stuart C. Shapiro, editor, Encyclopedia of Artificial Intelligence. John Wiley & Sons, 2 edition, 1992.

[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 [**?**].

[Nip98] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics. Formal Aspects of Computing, 10(2):171–186, 1998.

[Nip02] Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, Proof and System-Reliability, pages 341–367. Kluwer, 2002.

[Nip03] Tobias Nipkow. Java bytecode verification. 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. Mechanizing coinduction and corecursion in higher-order logic. Journal of Logic and Computation, 7(2):175–204, 1997. Download.

[Pau05] Lawrence C. Paulson. The Isabelle Reference Manual. Computer Laboratory, University of Cambridge, October 2005.

[Pea18] Guiseppe Peano. ?? ??, 18??

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

[PM68] Dag Prawitz and Per-Erik Malmnäs. A survey of some connections between classical, intuitionistic and minimal logic. In A. Schmidt and H. Schütte, editors, Contributions to

Mathematical Logic, pages 215–229. North-Holland, 1968.

[Pra65] Dag Prawitz. *Natural Deduction: A proof theoretical study*. Almqvist and Wiksell, 1965.

[Pra71] Dag Prawitz. Ideas and results in proof theory. In Jens Erik Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 235–308. North-Holland, 1971.

[SH84] Peter Schroeder-Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49(4):1284–1300, 1984.

[Sza69] M. E. Szabo. *The Collected Papers of Gerhard Gentzen*. North-Holland, 1969.

[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. Contains translations of original works by David Hilbert and Adolf Fraenkel and Ernst Zermelo.

[vL16] Gottfried Wilhelm von Leibniz. ?? ??, 16??

[WB89] Phillip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In Conference Record of the 16th ACM Symposium on Principles of Programming Languages, pages 60–76, 1989.

[Wen99] Markus Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In Yves Bertot, Gilles Dowek, André Hirschowitz, and and Laurent Théry C. Paulin, editors, Proceedings of TPHOLs, volume 1690 of LNCS, pages 19–36. Springer-Verlag, 1999.

[Win96] Glynn Winskel. The Formal Semantics of Programming Languages – An Introduction. MIT Press, 1996. 3rd ed.

[WR25] Alfred N. Whitehead and Bertrand Russell. Principia Mathematica, volume 1. Cambridge University Press, 1925. 2nd edition.

[Zer07] Ernst Zermelo. Untersuchungen über die Grundlagen der Mengenlehre. Mathematische

Annalen, 65:261–281, 1907. See [vH67].